arch Documentation

Release 6.3.0

Kevin Sheppard

Jan 05, 2024

CONTENTS

1 Univariate Volatility Models **3**

- 1.1 Introduction to ARCH Models . . . 3
- 1.2 ARCH Modeling . . . 6
- 1.3 Forecasting . . . 20
- 1.4 Volatility Forecasting . . . 26
- 1.5 Value-at-Risk Forecasting . . . 33
- 1.6 Volatility Scenarios . . . 36
- 1.7 Forecasting with Exogenous Regressors . . . 43
- 1.8 Mean Models . . . 51
- 1.9 Volatility Processes . . . 113
- 1.10 Using the Fixed Variance process . . . 200
- 1.11 Distributions . . . 205
- 1.12 Model Results . . . 232
- 1.13 Utilities . . . 248
- 1.14 Theoretical Background . . . 248

2 Bootstrapping **249**

- 2.1 Bootstrap Examples . . . 249
- 2.2 Confidence Intervals . . . 256
- 2.3 Covariance Estimation . . . 261
- 2.4 Low-level Interfaces . . . 262
- 2.5 Semiparametric Bootstraps . . . 264
- 2.6 Parametric Bootstraps . . . 265
- 2.7 Independent, Identical Distributed Data (i.i.d.) . . . 266
- 2.8 Independent Samples . . . 277
- 2.9 Time-series Bootstraps . . . 288
- 2.10 References . . . 321

3 Multiple Comparison Procedures **323**

- 3.1 Multiple Comparisons . . . 323
- 3.2 Module Reference . . . 330
- 3.3 References . . . 338

4 Unit Root Testing **339**

- 4.1 Introduction . . . 339
- 4.2 Unit Root Testing . . . 340
- 4.3 The Unit Root Tests . . . 350

5 Cointegration Analysis **373**

- 5.1 Cointegration Testing . . . 373

5.2 Cointegration Tests 380
5.3 Cointegrating Vector Estimation 384

6 Long-run Covariance Estimation **405**
6.1 Long-run Covariance Estimators 405
6.2 Results 448

7 API Reference **451**
7.1 Volatility Modeling 451
7.2 Unit Root Testing 452
7.3 Cointegration Testing 452
7.4 Cointegrating Relationship Estimation 452
7.5 Bootstraps 453
7.6 Testing with Multiple-Comparison 453
7.7 Long-run Covariance (HAC) Estimation 453

8 Change Logs **455**
8.1 Version 6 455
8.2 Past Releases 456

9 Citation **463**

10 Index **465**

Bibliography **467**

Python Module Index **469**

Index **471**

Note

Stable documentation for the latest release is located at doc. Documentation for recent developments is located at devel.

The ARCH toolbox contains routines for:

- Univariate volatility models;
- Bootstrapping;
- Multiple comparison procedures;
- Unit root tests;
- Cointegration Testing and Estimation; and
- Long-run covariance estimation.

Future plans are to continue to expand this toolbox to include additional routines relevant for the analysis of financial data.

CHAPTER

ONE

UNIVARIATE VOLATILITY MODELS

arch.univaraite provides both high-level (*arch_model()*) and low-level methods (see *Mean Models*) to specify models. All models can be used to produce forecasts either analytically (when tractable) or using simulation-based methods (Monte Carlo or residual Bootstrap).

1.1 Introduction to ARCH Models

ARCH models are a popular class of volatility models that use observed values of returns or residuals as volatility shocks. A basic GARCH model is specified as

$$r_t = \mu + \epsilon_t \tag{1.1}$$

$$\epsilon_t = \sigma_t e_t \tag{1.2}$$

$$\sigma_t^2 = \omega + \alpha\epsilon_{t-1}^2 + \beta\sigma_{t-1}^2 \tag{1.3}$$

A complete ARCH model is divided into three components:

- a *mean model*, e.g., a constant mean or an *ARX*;
- a *volatility process*, e.g., a *GARCH* or an *EGARCH* process; and
- a *distribution* for the standardized residuals.

In most applications, the simplest method to construct this model is to use the constructor function *arch_model()*

```
import datetime as dt

import pandas_datareader.data as web

from arch import arch_model

start = dt.datetime(2000, 1, 1)
end = dt.datetime(2014, 1, 1)
sp500 = web.DataReader('^GSPC', 'yahoo', start=start, end=end)
returns = 100 * sp500['Adj Close'].pct_change().dropna()
am = arch_model(returns)
```

Alternatively, the same model can be manually assembled from the building blocks of an ARCH model

```
from arch import ConstantMean, GARCH, Normal

am = ConstantMean(returns)
am.volatility = GARCH(1, 0, 1)
am.distribution = Normal()
```

In either case, model parameters are estimated using

```
res = am.fit()
```

with the following output

```
Iteration:      1,   Func. Count:      6,   Neg. LLF: 5159.58323938
Iteration:      2,   Func. Count:     16,   Neg. LLF: 5156.09760149
Iteration:      3,   Func. Count:     24,   Neg. LLF: 5152.29989336
Iteration:      4,   Func. Count:     31,   Neg. LLF: 5146.47531817
Iteration:      5,   Func. Count:     38,   Neg. LLF: 5143.86337547
Iteration:      6,   Func. Count:     45,   Neg. LLF: 5143.02096168
Iteration:      7,   Func. Count:     52,   Neg. LLF: 5142.24105141
Iteration:      8,   Func. Count:     60,   Neg. LLF: 5142.07138907
Iteration:      9,   Func. Count:     67,   Neg. LLF: 5141.416653
Iteration:     10,   Func. Count:     73,   Neg. LLF: 5141.39212288
Iteration:     11,   Func. Count:     79,   Neg. LLF: 5141.39023885
Iteration:     12,   Func. Count:     85,   Neg. LLF: 5141.39023359
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 5141.39023359
            Iterations: 12
            Function evaluations: 85
            Gradient evaluations: 12
```

```
print(res.summary())
```

yields

```
                     Constant Mean - GARCH Model Results
==============================================================================
Dep. Variable:                Adj Close   R-squared:                      -0.001
Mean Model:               Constant Mean   Adj. R-squared:                 -0.001
Vol Model:                        GARCH   Log-Likelihood:               -5141.39
Distribution:                    Normal   AIC:                           10290.8
Method:              Maximum Likelihood   BIC:                           10315.4
                                          No. Observations:                 3520
Date:                  Fri, Dec 02 2016   Df Residuals:                     3516
Time:                          22:22:28   Df Model:                            4
                                 Mean Model
==============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
------------------------------------------------------------------------------
mu             0.0531  1.487e-02      3.569  3.581e-04   [2.392e-02,8.220e-02]
                              Volatility Model
==============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
------------------------------------------------------------------------------
omega          0.0156  4.932e-03      3.155  1.606e-03   [5.892e-03,2.523e-02]
alpha[1]       0.0879  1.140e-02      7.710  1.260e-14     [6.554e-02,  0.110]
beta[1]        0.9014  1.183e-02     76.163      0.000     [  0.878,    0.925]
==============================================================================

Covariance estimator: robust
```

1.1.1 Model Constructor

While models can be carefully specified using the individual components, most common specifications can be specified using a simple model constructor.

arch.univariate.**arch_model**(y: *ndarray* | *DataFrame* | *Series* | *None*, x: *ndarray* | *DataFrame* | *None* = **None**, mean: *'Constant'* | *'Zero'* | *'LS'* | *'AR'* | *'ARX'* | *'HAR'* | *'HARX'* | *'constant'* | *'zero'* = `'Constant'`, lags: *None* | *int* | *list[int]* | *ndarray* = **0**, vol: *'GARCH'* | *'ARCH'* | *'EGARCH'* | *'FIGARCH'* | *'APARCH'* | *'HARCH'* = `'GARCH'`, p: *int* | *list[int]* = **1**, o: *int* = **0**, q: *int* = **1**, power: *float* = **2.0**, dist: *'normal'* | *'gaussian'* | *'t'* | *'studentst'* | *'skewstudent'* | *'skewt'* | *'ged'* | *'generalized error'* = `'normal'`, hold_back: *int* | *None* = **None**, rescale: *bool* | *None* = **None**) → *HARX*

Initialization of common ARCH model specifications

Parameters

y: ndarray | DataFrame | Series | None
The dependent variable

x: ndarray | DataFrame | None = None
Exogenous regressors. Ignored if model does not permit exogenous regressors.

mean: 'Constant' | 'Zero' | 'LS' | 'AR' | 'ARX' | 'HAR' | 'HARX' | 'constant' | 'zero' = 'Constant'
Name of the mean model. Currently supported options are: 'Constant', 'Zero', 'LS', 'AR', 'ARX', 'HAR' and 'HARX'

lags: None | int | list[int] | ndarray = 0
Either a scalar integer value indicating lag length or a list of integers specifying lag locations.

vol: 'GARCH' | 'ARCH' | 'EGARCH' | 'FIGARCH' | 'APARCH' | 'HARCH' = 'GARCH'
Name of the volatility model. Currently supported options are: 'GARCH' (default), 'ARCH', 'EGARCH', 'FIGARCH', 'APARCH' and 'HARCH'

p: int | list[int] = 1
Lag order of the symmetric innovation

o: int = 0
Lag order of the asymmetric innovation

q: int = 1
Lag order of lagged volatility or equivalent

power: float = 2.0
Power to use with GARCH and related models

dist: 'normal' | 'gaussian' | 't' | 'studentst' | 'skewstudent' | 'skewt' | 'ged' | 'generalized error' = 'normal'
Name of the error distribution. Currently supported options are:

- Normal: 'normal', 'gaussian' (default)
- Students's t: 't', 'studentst'
- Skewed Student's t: 'skewstudent', 'skewt'
- Generalized Error Distribution: 'ged', 'generalized error"

hold_back: int | None = None
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.

rescale: bool | None = None
Flag indicating whether to automatically rescale data if the scale of the data is likely to produce convergence issues when estimating model parameters. If False, the model is estimated on the data without transformation. If True, than y is rescaled and the new scale is reported in the estimation results.

Returns
model – Configured ARCH model

Return type
`ARCHModel`

Examples

```
>>> import datetime as dt
>>> import pandas_datareader.data as web
>>> djia = web.get_data_fred('DJIA')
>>> returns = 100 * djia['DJIA'].pct_change().dropna()
```

A basic GARCH(1,1) with a constant mean can be constructed using only the return data

```
>>> from arch.univariate import arch_model
>>> am = arch_model(returns)
```

Alternative mean and volatility processes can be directly specified

```
>>> am = arch_model(returns, mean='AR', lags=2, vol='harch', p=[1, 5, 22])
```

This example demonstrates the construction of a zero mean process with a TARCH volatility process and Student t error distribution

```
>>> am = arch_model(returns, mean='zero', p=1, o=1, q=1,
...                 power=1.0, dist='StudentsT')
```

Notes

Input that are not relevant for a particular specification, such as *lags* when *mean='zero'*, are silently ignored.

1.2 ARCH Modeling

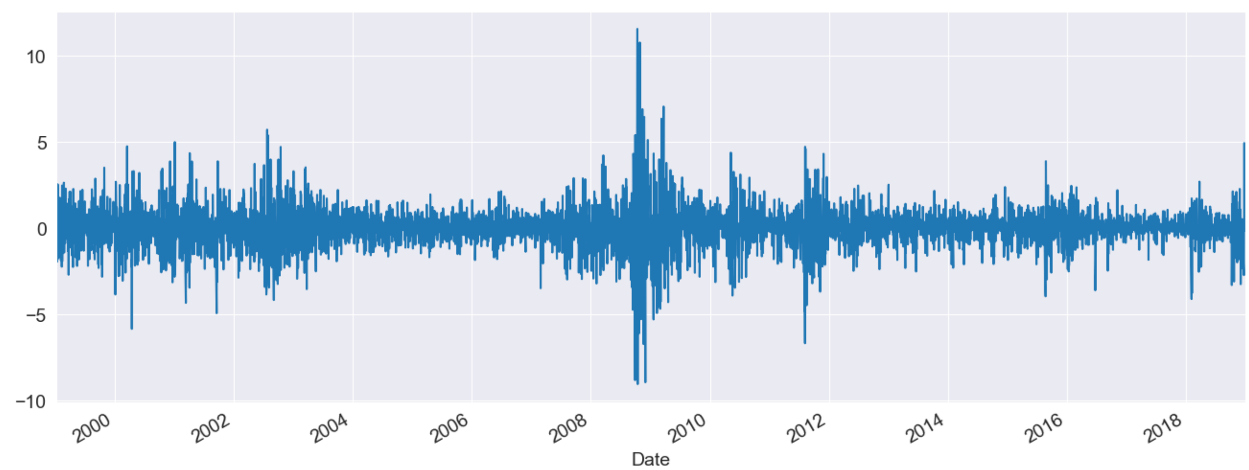
This setup code is required to run in an IPython notebook

```
[1]: %matplotlib inline
import matplotlib.pyplot as plt
import seaborn

seaborn.set_style("darkgrid")
plt.rc("figure", figsize=(16, 6))
plt.rc("savefig", dpi=90)
plt.rc("font", family="sans-serif")
plt.rc("font", size=14)
```

1.2.1 Setup

These examples will all make use of financial data from Yahoo! Finance. This data set can be loaded from arch.data.sp500.

[2]:
```
import datetime as dt

import arch.data.sp500

st = dt.datetime(1988, 1, 1)
en = dt.datetime(2018, 1, 1)
data = arch.data.sp500.load()
market = data["Adj Close"]
returns = 100 * market.pct_change().dropna()
ax = returns.plot()
xlim = ax.set_xlim(returns.index.min(), returns.index.max())
```



1.2.2 Specifying Common Models

The simplest way to specify a model is to use the model constructor arch.arch_model which can specify most common models. The simplest invocation of arch will return a model with a constant mean, GARCH(1,1) volatility process and normally distributed errors.

$$r_t = \mu + \epsilon_t$$

$$\sigma_t^2 = \omega + \alpha\epsilon_{t-1}^2 + \beta\sigma_{t-1}^2$$

$$\epsilon_t = \sigma_t e_t, \;\; e_t \sim N(0,1)$$

The model is estimated by calling fit. The optional inputs iter controls the frequency of output form the optimizer, and disp controls whether convergence information is returned. The results class returned offers direct access to the estimated parameters and related quantities, as well as a summary of the estimation results.

GARCH (with a Constant Mean)

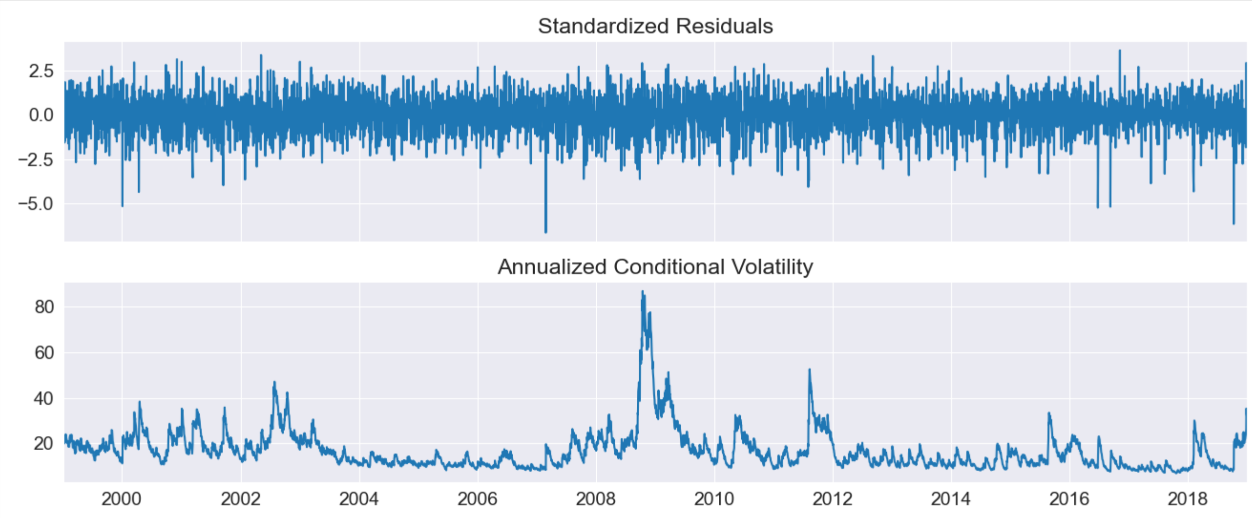
The default set of options produces a model with a constant mean, GARCH(1,1) conditional variance and normal errors.

[3]:
```
from arch import arch_model

am = arch_model(returns)
res = am.fit(update_freq=5)
print(res.summary())
```

```
Iteration:      5,   Func. Count:     35,   Neg. LLF: 6970.2779565773335
Iteration:     10,   Func. Count:     63,   Neg. LLF: 6936.718477482076
Optimization terminated successfully    (Exit mode 0)
            Current function value: 6936.718476988951
            Iterations: 11
            Function evaluations: 68
            Gradient evaluations: 11
                     Constant Mean - GARCH Model Results
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:             Constant Mean   Adj. R-squared:                  0.000
Vol Model:                      GARCH   Log-Likelihood:               -6936.72
Distribution:                  Normal   AIC:                           13881.4
Method:            Maximum Likelihood   BIC:                           13907.5
                                        No. Observations:                 5030
Date:                Tue, May 30 2023   Df Residuals:                     5029
Time:                        10:55:28   Df Model:                            1
                                 Mean Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
mu             0.0564  1.149e-02      4.906  9.302e-07 [3.384e-02,7.887e-02]
                              Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.0175  4.683e-03      3.738  1.854e-04 [8.328e-03,2.669e-02]
alpha[1]       0.1022  1.301e-02      7.852  4.105e-15   [7.665e-02,  0.128]
beta[1]        0.8852  1.380e-02     64.125      0.000     [  0.858,  0.912]
============================================================================

Covariance estimator: robust
```

`plot()` can be used to quickly visualize the standardized residuals and conditional volatility.

[4]:
```
fig = res.plot(annualize="D")
```



GJR-GARCH

Additional inputs can be used to construct other models. This example sets `o` to 1, which includes one lag of an asymmetric shock which transforms a GARCH model into a GJR-GARCH model with variance dynamics given by

$$\sigma_t^2 = \omega + \alpha\epsilon_{t-1}^2 + \gamma\epsilon_{t-1}^2 I_{[\epsilon_{t-1}<0]} + \beta\sigma_{t-1}^2$$

where I is an indicator function that takes the value 1 when its argument is true.

The log likelihood improves substantially with the introduction of an asymmetric term, and the parameter estimate is highly significant.

[5]:
```
am = arch_model(returns, p=1, o=1, q=1)
res = am.fit(update_freq=5, disp="off")
print(res.summary())
```

```
                   Constant Mean - GJR-GARCH Model Results
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:             Constant Mean   Adj. R-squared:                  0.000
Vol Model:                  GJR-GARCH   Log-Likelihood:               -6822.88
Distribution:                  Normal   AIC:                           13655.8
Method:            Maximum Likelihood   BIC:                           13688.4
                                        No. Observations:                 5030
Date:                Tue, May 30 2023   Df Residuals:                     5029
Time:                        10:55:29   Df Model:                            1
                                 Mean Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
mu             0.0175  1.145e-02      1.529      0.126 [-4.936e-03,3.995e-02]
                              Volatility Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
omega          0.0196  4.051e-03      4.830  1.362e-06   [1.163e-02,2.751e-02]
```

(continues on next page)

(continued from previous page)

```
alpha[1]   2.4027e-10  1.026e-02  2.341e-08      1.000 [-2.011e-02,2.011e-02]
gamma[1]       0.1831  2.266e-02      8.079  6.543e-16      [  0.139,  0.227]
beta[1]        0.8922  1.458e-02     61.200      0.000      [  0.864,  0.921]
============================================================================

Covariance estimator: robust
```

TARCH/ZARCH

TARCH (also known as ZARCH) model the *volatility* using absolute values. This model is specified using `power=1.0` since the default power, 2, corresponds to variance processes that evolve in squares.

The volatility process in a TARCH model is given by

$$\sigma_t = \omega + \alpha \left|\epsilon_{t-1}\right| + \gamma \left|\epsilon_{t-1}\right| I_{[\epsilon_{t-1}<0]} + \beta \sigma_{t-1}$$

More general models with other powers (κ) have volatility dynamics given by

$$\sigma_t^\kappa = \omega + \alpha \left|\epsilon_{t-1}\right|^\kappa + \gamma \left|\epsilon_{t-1}\right|^\kappa I_{[\epsilon_{t-1}<0]} + \beta \sigma_{t-1}^\kappa$$

where the conditional variance is $\left(\sigma_t^\kappa\right)^{2/\kappa}$.

The TARCH model also improves the fit, although the change in the log likelihood is less dramatic.

[6]:
```python
am = arch_model(returns, p=1, o=1, q=1, power=1.0)
res = am.fit(update_freq=5)
print(res.summary())
```

```
Iteration:      5,   Func. Count:     45,   Neg. LLF: 6829.197763035331
Iteration:     10,   Func. Count:     79,   Neg. LLF: 6799.178613053096
Optimization terminated successfully    (Exit mode 0)
            Current function value: 6799.178523172141
            Iterations: 14
            Function evaluations: 103
            Gradient evaluations: 13
                  Constant Mean - TARCH/ZARCH Model Results
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:             Constant Mean   Adj. R-squared:                  0.000
Vol Model:                TARCH/ZARCH   Log-Likelihood:               -6799.18
Distribution:                  Normal   AIC:                           13608.4
Method:            Maximum Likelihood   BIC:                           13641.0
                                        No. Observations:                 5030
Date:                Tue, May 30 2023   Df Residuals:                     5029
Time:                        10:55:29   Df Model:                            1
                                  Mean Model
============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------
mu             0.0143  1.091e-02      1.311      0.190 [-7.086e-03,3.570e-02]
                               Volatility Model
============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
```

(continues on next page)

(continued from previous page)

```
-----------------------------------------------------------------------------
omega          0.0258  4.100e-03      6.299  2.987e-10  [1.779e-02,3.386e-02]
alpha[1]   5.9528e-09  9.156e-03  6.502e-07      1.000 [-1.794e-02,1.794e-02]
gamma[1]       0.1707  1.601e-02     10.664  1.501e-26      [  0.139,  0.202]
beta[1]        0.9098  9.671e-03     94.067      0.000      [  0.891,  0.929]
=============================================================================

Covariance estimator: robust
```

Student's T Errors

Financial returns are often heavy tailed, and a Student's T distribution is a simple method to capture this feature. The call to `arch` changes the distribution from a Normal to a Students's T.

The standardized residuals appear to be heavy tailed with an estimated degree of freedom near 10. The log-likelihood also shows a large increase.

[7]:
```
am = arch_model(returns, p=1, o=1, q=1, power=1.0, dist="StudentsT")
res = am.fit(update_freq=5)
print(res.summary())
```

```
Iteration:      5,   Func. Count:     50,   Neg. LLF: 6728.995165345449
Iteration:     10,   Func. Count:     90,   Neg. LLF: 6722.15117736426
Optimization terminated successfully    (Exit mode 0)
            Current function value: 6722.1511833987915
            Iterations: 13
            Function evaluations: 110
            Gradient evaluations: 12
                     Constant Mean - TARCH/ZARCH Model Results
====================================================================================
Dep. Variable:                  Adj Close   R-squared:                       0.000
Mean Model:                 Constant Mean   Adj. R-squared:                  0.000
Vol Model:                    TARCH/ZARCH   Log-Likelihood:                -6722.15
Distribution:      Standardized Student's t   AIC:                           13456.3
Method:                Maximum Likelihood   BIC:                           13495.4
                                            No. Observations:                 5030
Date:                    Tue, May 30 2023   Df Residuals:                     5029
Time:                            10:55:29   Df Model:                            1
                                Mean Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
mu             0.0323  2.292e-03     14.077  5.292e-45 [2.777e-02,3.676e-02]
                              Volatility Model
=============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
-----------------------------------------------------------------------------
omega          0.0201  3.492e-03      5.745  9.179e-09  [1.322e-02,2.691e-02]
alpha[1]       0.0000  8.223e-03      0.000      1.000 [-1.612e-02,1.612e-02]
gamma[1]       0.1721  1.512e-02     11.386  4.925e-30      [  0.143,  0.202]
beta[1]        0.9139  9.578e-03     95.414      0.000      [  0.895,  0.933]
                              Distribution
```

(continues on next page)

(continued from previous page)

```
========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
nu             7.9549      0.880      9.036  1.620e-19 [  6.229,  9.680]
========================================================================

Covariance estimator: robust
```

1.2.3 Fixing Parameters

In some circumstances, fixed rather than estimated parameters might be of interest. A model-result-like class can be generated using the `fix()` method. The class returned is identical to the usual model result class except that information about inference (standard errors, t-stats, etc) is not available.

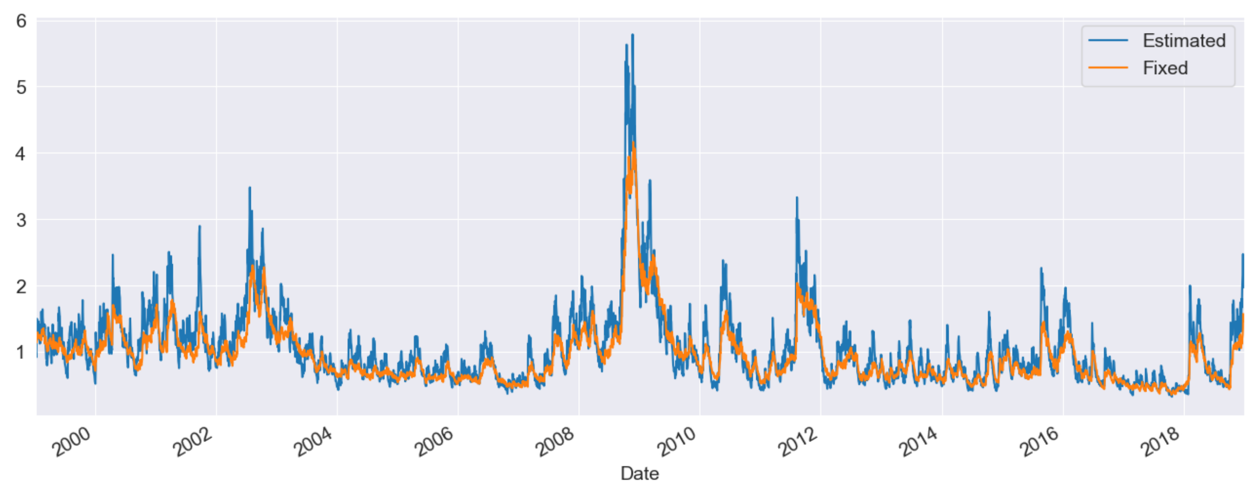
In the example, I fix the parameters to a symmetric version of the previously estimated model.

[8]:
```
fixed_res = am.fix([0.0235, 0.01, 0.06, 0.0, 0.9382, 8.0])
print(fixed_res.summary())
```

```
                      Constant Mean - TARCH/ZARCH Model Results
=====================================================================================
Dep. Variable:                   Adj Close   R-squared:                           --
Mean Model:                  Constant Mean   Adj. R-squared:                      --
Vol Model:                     TARCH/ZARCH   Log-Likelihood:                -6908.93
Distribution:      Standardized Student's t   AIC:                           13829.9
Method:          User-specified Parameters   BIC:                           13869.0
                                             No. Observations:                 5030
Date:                     Tue, May 30 2023
Time:                             10:55:29
      Mean Model
=====================
                 coef
---------------------
mu             0.0235
   Volatility Model
=====================
                 coef
---------------------
omega          0.0100
alpha[1]       0.0600
gamma[1]       0.0000
beta[1]        0.9382
     Distribution
=====================
                 coef
---------------------
nu             8.0000
=====================

Results generated with user-specified parameters.
Std. errors not available when the model is not estimated,
```

[9]:
```python
import pandas as pd

df = pd.concat([res.conditional_volatility, fixed_res.conditional_volatility], axis=1)
df.columns = ["Estimated", "Fixed"]
subplot = df.plot()
subplot.set_xlim(xlim)
```

[9]: (10596.0, 17896.0)



1.2.4 Building a Model From Components

Models can also be systematically assembled from the three model components:

- A mean model (`arch.mean`)
  - Zero mean (`ZeroMean`) - useful if using residuals from a model estimated separately
  - Constant mean (`ConstantMean`) - common for most liquid financial assets
  - Autoregressive (`ARX`) with optional exogenous regressors
  - Heterogeneous (`HARX`) autoregression with optional exogenous regressors
  - Exogenous regressors only (`LS`)
- A volatility process (`arch.volatility`)
  - ARCH (`ARCH`)
  - GARCH (`GARCH`)
  - GJR-GARCH (`GARCH` using `o` argument)
  - TARCH/ZARCH (`GARCH` using `power` argument set to 1)
  - Power GARCH and Asymmetric Power GARCH (`GARCH` using `power`)
  - Exponentially Weighted Moving Average Variance with estimated coefficient (`EWMAVariance`)
  - Heterogeneous ARCH (`HARCH`)
  - Parameterless Models
    - Exponentially Weighted Moving Average Variance, known as RiskMetrics (`EWMAVariance`)

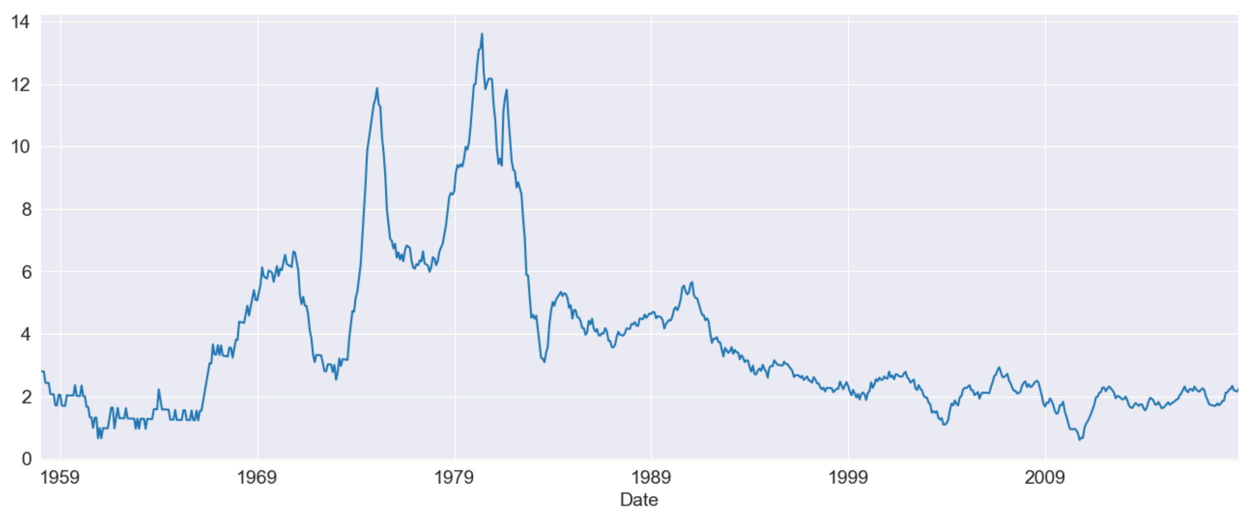
  * Weighted averages of EWMAs, known as the RiskMetrics 2006 methodology (`RiskMetrics2006`)

- A distribution (`arch.distribution`)
  - Normal (`Normal`)
  - Standardized Students's T (`StudentsT`)

Mean Models

The first choice is the mean model. For many liquid financial assets, a constant mean (or even zero) is adequate. For other series, such as inflation, a more complicated model may be required. These examples make use of Core CPI downloaded from the Federal Reserve Economic Data site.

[10]:
```
import arch.data.core_cpi

core_cpi = arch.data.core_cpi.load()
ann_inflation = 100 * core_cpi.CPILFESL.pct_change(12).dropna()
fig = ann_inflation.plot()
```



All mean models are initialized with constant variance and normal errors. For `ARX` models, the `lags` argument specifies the lags to include in the model.

[11]:
```
from arch.univariate import ARX

ar = ARX(100 * ann_inflation, lags=[1, 3, 12])
print(ar.fit().summary())
```

```
                     AR - Constant Variance Model Results
==============================================================================
Dep. Variable:                   CPILFESL   R-squared:                   0.991
Mean Model:                            AR   Adj. R-squared:              0.991
Vol Model:              Constant Variance   Log-Likelihood:           -3299.84
Distribution:                      Normal   AIC:                       6609.68
Method:                Maximum Likelihood   BIC:                       6632.57
                                            No. Observations:              719
Date:                    Tue, May 30 2023   Df Residuals:                  715
Time:                            10:55:30   Df Model:                        4
```

(continues on next page)

(continued from previous page)

```
                                  Mean Model
==============================================================================
                  coef    std err          t      P>|t|       95.0% Conf. Int.
------------------------------------------------------------------------------
Const           4.0216      2.030      1.981  4.762e-02    [4.218e-02,  8.001]
CPILFESL[1]     1.1921  3.475e-02     34.306 6.315e-258      [  1.124,  1.260]
CPILFESL[3]    -0.1798  4.076e-02     -4.411  1.030e-05   [ -0.260,-9.989e-02]
CPILFESL[12]   -0.0232  1.370e-02     -1.692  9.058e-02 [-5.002e-02,3.666e-03]
                             Volatility Model
===========================================================================
                coef    std err          t      P>|t|      95.0% Conf. Int.
---------------------------------------------------------------------------
sigma2      567.4180     64.487      8.799  1.381e-18 [4.410e+02,6.938e+02]
===========================================================================

Covariance estimator: White's Heteroskedasticity Consistent Estimator
```

Volatility Processes

Volatility processes can be added a a mean model using the `volatility` property. This example adds an ARCH(5) process to model volatility. The arguments `iter` and `disp` are used in `fit()` to suppress estimation output.

[12]:
```python
from arch.univariate import ARCH, GARCH

ar.volatility = ARCH(p=5)
res = ar.fit(update_freq=0, disp="off")
print(res.summary())
```

```
                           AR - ARCH Model Results
==============================================================================
Dep. Variable:               CPILFESL   R-squared:                       0.991
Mean Model:                        AR   Adj. R-squared:                  0.991
Vol Model:                       ARCH   Log-Likelihood:               -3174.60
Distribution:                  Normal   AIC:                           6369.19
Method:            Maximum Likelihood   BIC:                           6414.97
                                        No. Observations:                  719
Date:                Tue, May 30 2023   Df Residuals:                      715
Time:                        10:55:30   Df Model:                            4
                                  Mean Model
==============================================================================
                  coef    std err          t      P>|t|       95.0% Conf. Int.
------------------------------------------------------------------------------
Const           2.8500      1.883      1.513      0.130      [ -0.841,  6.541]
CPILFESL[1]     1.0859  3.534e-02     30.726 2.596e-207      [  1.017,  1.155]
CPILFESL[3]    -0.0788  3.855e-02     -2.045  4.085e-02   [ -0.154,-3.282e-03]
CPILFESL[12]   -0.0189  1.157e-02     -1.630      0.103 [-4.154e-02,3.820e-03]
                            Volatility Model
=========================================================================
                coef    std err          t      P>|t|    95.0% Conf. Int.
-------------------------------------------------------------------------
omega        76.8696     16.017      4.799  1.592e-06 [ 45.478,1.083e+02]
```

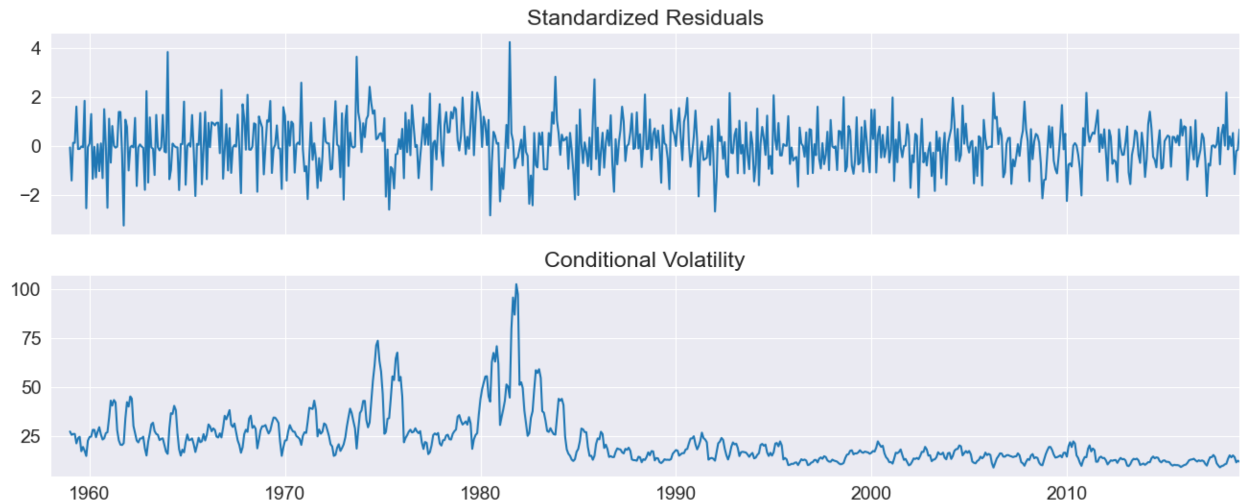
(continues on next page)

(continued from previous page)

```
alpha[1]        0.1345  4.003e-02      3.359  7.827e-04 [5.600e-02,  0.213]
alpha[2]        0.2280  6.284e-02      3.628  2.861e-04   [  0.105,  0.351]
alpha[3]        0.1838  6.802e-02      2.702  6.894e-03 [5.046e-02,  0.317]
alpha[4]        0.2538  7.826e-02      3.242  1.185e-03   [  0.100,  0.407]
alpha[5]        0.1954  7.091e-02      2.756  5.856e-03 [5.643e-02,  0.334]
===========================================================================

Covariance estimator: robust
```

Plotting the standardized residuals and the conditional volatility shows some large (in magnitude) errors, even when standardized.

[13]:
```
fig = res.plot()
```



Distributions

Finally the distribution can be changed from the default normal to a standardized Student's T using the `distribution` property of a mean model.

The Student's t distribution improves the model, and the degree of freedom is estimated to be near 8.

[14]:
```
from arch.univariate import StudentsT

ar.distribution = StudentsT()
res = ar.fit(update_freq=0, disp="off")
print(res.summary())
```

```
                              AR - ARCH Model Results
====================================================================================
Dep. Variable:                     CPILFESL   R-squared:                       0.991
Mean Model:                              AR   Adj. R-squared:                  0.991
Vol Model:                             ARCH   Log-Likelihood:               -3168.25
Distribution:      Standardized Student's t   AIC:                           6358.51
Method:                  Maximum Likelihood   BIC:                           6408.86
                                              No. Observations:                  719
Date:                      Tue, May 30 2023   Df Residuals:                      715
```

(continues on next page)

(continued from previous page)

```
Time:                            10:55:31   Df Model:                            4
                                Mean Model
===============================================================================
                  coef    std err          t      P>|t|       95.0% Conf. Int.
-------------------------------------------------------------------------------
Const           3.1220      1.861      1.678  9.343e-02      [ -0.526,  6.770]
CPILFESL[1]     1.0843  3.525e-02     30.762 8.350e-208      [  1.015,  1.153]
CPILFESL[3]    -0.0730  3.873e-02     -1.885  5.948e-02    [ -0.149,2.919e-03]
CPILFESL[12]   -0.0236  1.316e-02     -1.791  7.329e-02 [-4.935e-02,2.223e-03]
                            Volatility Model
==========================================================================
                coef    std err          t      P>|t|     95.0% Conf. Int.
--------------------------------------------------------------------------
omega        87.3503     20.625      4.235  2.283e-05 [ 46.927,1.278e+02]
alpha[1]      0.1715  5.064e-02      3.386  7.087e-04 [7.222e-02,  0.271]
alpha[2]      0.2202  6.394e-02      3.444  5.741e-04 [9.486e-02,  0.345]
alpha[3]      0.1547  6.327e-02      2.446  1.447e-02 [3.072e-02,  0.279]
alpha[4]      0.2117  7.287e-02      2.905  3.677e-03 [6.884e-02,  0.354]
alpha[5]      0.1959  7.853e-02      2.495  1.261e-02 [4.199e-02,  0.350]
                              Distribution
========================================================================
                coef    std err          t      P>|t|   95.0% Conf. Int.
------------------------------------------------------------------------
nu            9.0451      3.366      2.687  7.205e-03 [  2.448, 15.642]
========================================================================

Covariance estimator: robust
```

1.2.5 WTI Crude

The next example uses West Texas Intermediate Crude data from FRED. Three models are fit using alternative distributional assumptions. The results are printed, where we can see that the normal has a much lower log-likelihood than either the Standard Student's T or the Standardized Skew Student's T – however, these two are fairly close. The closeness of the T and the Skew T indicate that returns are not heavily skewed.

[15]:
```
from collections import OrderedDict

import arch.data.wti

crude = arch.data.wti.load()
crude_ret = 100 * crude.DCOILWTICO.dropna().pct_change().dropna()
res_normal = arch_model(crude_ret).fit(disp="off")
res_t = arch_model(crude_ret, dist="t").fit(disp="off")
res_skewt = arch_model(crude_ret, dist="skewt").fit(disp="off")
lls = pd.Series(
    OrderedDict(
        (
            ("normal", res_normal.loglikelihood),
            ("t", res_t.loglikelihood),
            ("skewt", res_skewt.loglikelihood),
        )
```

(continues on next page)

(continued from previous page)

```
    )
)
print(lls)
params = pd.DataFrame(
    OrderedDict(
        (
            ("normal", res_normal.params),
            ("t", res_t.params),
            ("skewt", res_skewt.params),
        )
    )
)
params
```

```
normal   -18165.858870
t        -17919.643916
skewt    -17916.669052
dtype: float64
```

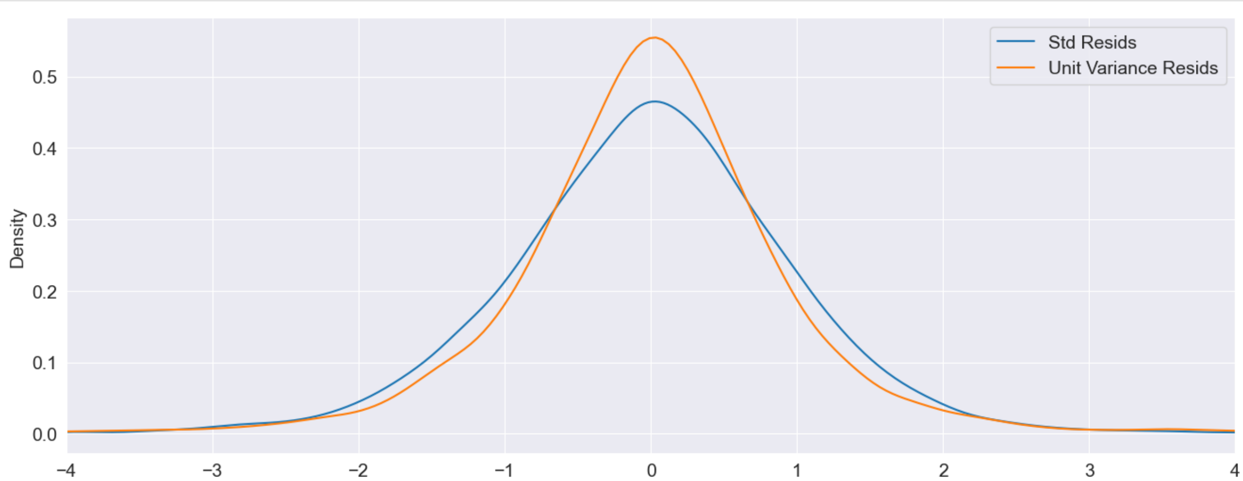
[15]:

```
              normal         t      skewt
alpha[1]   0.085627  0.064980  0.064889
beta[1]    0.909098  0.927950  0.928215
eta             NaN       NaN  6.186541
lambda          NaN       NaN -0.036986
mu         0.046682  0.056438  0.040928
nu              NaN  6.178598       NaN
omega      0.055806  0.048516  0.047683
```

The standardized residuals can be computed by dividing the residuals by the conditional volatility. These are plotted along with the (unstandardized, but scaled) residuals. The non-standardized residuals are more peaked in the center indicating that the distribution is somewhat more heavy tailed than that of the standardized residuals.

[16]:

```
std_resid = res_normal.resid / res_normal.conditional_volatility
unit_var_resid = res_normal.resid / res_normal.resid.std()
df = pd.concat([std_resid, unit_var_resid], axis=1)
df.columns = ["Std Resids", "Unit Variance Resids"]
subplot = df.plot(kind="kde", xlim=(-4, 4))
```



1.2.6 Simulation

All mean models expose a method to simulate returns from assuming the model is correctly specified. There are two required parameters, `params` which are the model parameters, and `nobs`, the number of observations to produce.

Below we simulate from a GJR-GARCH(1,1) with Skew-t errors using parameters estimated on the WTI series. The simulation returns a `DataFrame` with 3 columns:

- `data`: The simulated data, which includes any mean dynamics.
- `volatility`: The conditional volatility series
- `errors`: The simulated errors generated to produce the model. The errors are the difference between the data and its conditional mean, and can be transformed into the standardized errors by dividing by the volatility.

[17]:
```
res = arch_model(crude_ret, p=1, o=1, q=1, dist="skewt").fit(disp="off")
pd.DataFrame(res.params)
```

[17]:
```
            params
mu        0.029365
omega     0.044374
alpha[1]  0.044344
gamma[1]  0.036104
beta[1]   0.931280
eta       6.211281
lambda   -0.041616
```

[18]:
```
sim_mod = arch_model(None, p=1, o=1, q=1, dist="skewt")

sim_data = sim_mod.simulate(res.params, 1000)
sim_data.head()
```

[18]:
```
       data  volatility    errors
0  2.242949    2.325649  2.213584
1  0.404109    2.301873  0.374743
2 -0.106860    2.232734 -0.136226
3 -1.678962    2.165269 -1.708327
4 -1.388209    2.155309 -1.417574
```

Simulations can be reproduced using a NumPy `RandomState`. This requires constructing a model from components where the `RandomState` instance is passed into to the distribution when the model is created.

The cell below contains code that builds a model with a constant mean, GJR-GARCH volatility and Skew t errors initialized with a user-provided `RandomState`. Saving the initial state allows it to be restored later so that the simulation can be run with the same random values.

[19]:
```
import numpy as np
from arch.univariate import GARCH, ConstantMean, SkewStudent

rs = np.random.RandomState([892380934, 189201902, 129129894, 9890437])
# Save the initial state to reset later
state = rs.get_state()

dist = SkewStudent(seed=rs)
vol = GARCH(p=1, o=1, q=1)
repro_mod = ConstantMean(None, volatility=vol, distribution=dist)
```

(continues on next page)

(continued from previous page)

```
repro_mod.simulate(res.params, 1000).head()
```

[19]:

```
       data  volatility    errors
0  1.616836    4.787697  1.587470
1  4.106780    4.637129  4.077415
2  4.530200    4.561457  4.500834
3  2.284833    4.507739  2.255468
4  3.378519    4.381016  3.349153
```

Resetting the state using `set_state` shows that calling `simulate` using the same underlying state in the `RandomState` produces the same objects.

[20]:

```
# Reset the state to the initial state
rs.set_state(state)
repro_mod.simulate(res.params, 1000).head()
```

[20]:

```
       data  volatility    errors
0  1.616836    4.787697  1.587470
1  4.106780    4.637129  4.077415
2  4.530200    4.561457  4.500834
3  2.284833    4.507739  2.255468
4  3.378519    4.381016  3.349153
```

1.3 Forecasting

Multi-period forecasts can be easily produced for ARCH-type models using forward recursion, with some caveats. In particular, models that are non-linear in the sense that they do not evolve using squares or residuals do not normally have analytically tractable multi-period forecasts available.

All models support three methods of forecasting:

- Analytical: analytical forecasts are always available for the 1-step ahead forecast due to the structure of ARCH-type models. Multi-step analytical forecasts are only available for model which are linear in the square of the residual, such as GARCH or HARCH.
- Simulation: simulation-based forecasts are always available for any horizon, although they are only useful for horizons larger than 1 since the first out-of-sample forecast from an ARCH-type model is always fixed. Simulation-based forecasts make use of the structure of an ARCH-type model to forward simulate using the assumed distribution of residuals, e.g., a Normal or Student's t.
- Bootstrap: bootstrap-based forecasts are similar to simulation based forecasts except that they make use of the standardized residuals from the actual data used in the estimation rather than assuming a specific distribution. Like simulation-base forecasts, bootstrap-based forecasts are only useful for horizons larger than 1. Additionally, the bootstrap forecasting method requires a minimal amount of in-sample data to use prior to producing the forecasts.

This document will use a standard GARCH(1,1) with a constant mean to explain the choices available for forecasting. The model can be described as

$$r_t = \mu + \epsilon_t \tag{1.4}$$

$$\epsilon_t = \sigma_t e_t \tag{1.5}$$

$$\sigma_t^2 = \omega + \alpha\epsilon_{t-1}^2 + \beta\sigma_{t-1}^2 \tag{1.6}$$

$$e_t \sim N(0,1) \tag{1.7}$$

In code this model can be constructed using data from the S&P 500 using

```
from arch import arch_model
import datetime as dt
import pandas_datareader.data as web
start = dt.datetime(2000,1,1)
end = dt.datetime(2014,1,1)
sp500 = web.get_data_yahoo('^GSPC', start=start, end=end)
returns = 100 * sp500['Adj Close'].pct_change().dropna()
am = arch_model(returns, vol='Garch', p=1, o=0, q=1, dist='Normal')
```

The model will be estimated using the first 10 years to estimate parameters and then forecasts will be produced for the final 5.

```
split_date = dt.datetime(2010,1,1)
res = am.fit(last_obs=split_date)
```

1.3.1 Analytical Forecasts

Analytical forecasts are available for most models that evolve in terms of the squares of the model residuals, e.g., GARCH, HARCH, etc. These forecasts exploit the relationship $E_t[\epsilon^2_{t+1}] = \sigma^2_{t+1}$ to recursively compute forecasts.

Variance forecasts are constructed for the conditional variances as

$$\begin{aligned}
\sigma^2_{t+1} &= \omega + \alpha\epsilon^2_t + \beta\sigma^2_t && (1.8)\\
\sigma^2_{t+h} &= \omega + \alpha E_t[\epsilon^2_{t+h-1}] + \beta E_t[\sigma^2_{t+h-1}]\, h \geq 2 && (1.9)\\
&= \omega + (\alpha + \beta)\, E_t[\sigma^2_{t+h-1}]\, h \geq 2 && (1.10)
\end{aligned}$$

```
forecasts = res.forecast(horizon=5, start=split_date)
forecasts.variance[split_date:].plot()
```

1.3.2 Simulation Forecasts

Simulation-based forecasts use the model random number generator to simulate draws of the standardized residuals, e_{t+h}. These are used to generate a pre-specified number of paths of the variances which are then averaged to produce the forecasts. In models like GARCH which evolve in the squares of the residuals, there are few advantages to simulation-based forecasting. These methods are more valuable when producing multi-step forecasts from models that do not have closed form multi-step forecasts such as EGARCH models.

Assume there are B simulated paths. A single simulated path is generated using

$$\begin{aligned}
\sigma^2_{t+h,b} &= \omega + \alpha\epsilon^2_{t+h-1,b} + \beta\sigma^2_{t+h-1,b} && (1.11)\\
\epsilon_{t+h,b} &= e_{t+h,b}\sqrt{\sigma^2_{t+h,b}} && (1.12)
\end{aligned}$$

where the simulated shocks are $e_{t+1,b}, e_{t+2,b}, \ldots, e_{t+h,b}$ where b is included to indicate that the simulations are independent across paths. Note that the first residual, ϵ_t, is in-sample and so is not simulated.

The final variance forecasts are then computed using the B simulations

$$E_t[\epsilon^2_{t+h}] = \sigma^2_{t+h} = B^{-1}\sum_{b=1}^{B}\sigma^2_{t+h,b}. \quad (1.13)$$

```
forecasts = res.forecast(horizon=5, start=split_date, method='simulation')
```

1.3.3 Bootstrap Forecasts

Bootstrap-based forecasts are virtually identical to simulation-based forecasts except that the standardized residuals are generated by the model. These standardized residuals are generated using the observed data and the estimated parameters as

$$\hat{e}_t = \frac{r_t - \hat{\mu}}{\hat{\sigma}_t} \tag{1.14}$$

The generation scheme is identical to the simulation-based method except that the simulated shocks are drawn (i.i.d., with replacement) from $\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_t$. so that only data available at time t are used to simulate the paths.

1.3.4 Forecasting Options

The *forecast()* method is attached to a model fit result.`

- `params` - The model parameters used to forecast the mean and variance. If not specified, the parameters estimated during the call to `fit` the produced the result are used.
- `horizon` - A positive integer value indicating the maximum horizon to produce forecasts.
- `start` - A positive integer or, if the input to the mode is a DataFrame, a date (string, datetime, datetime64 or Timestamp). Forecasts are produced from `start` until the end of the sample. If not provided, `start` is set to the length of the input data minus 1 so that only 1 forecast is produced.
- `align` - One of 'origin' (default) or 'target' that describes how the forecasts aligned in the output. Origin aligns forecasts to the last observation used in producing the forecast, while target aligns forecasts to the observation index that is being forecast.
- `method` - One of 'analytic' (default), 'simulation' or 'bootstrap' that describes the method used to produce the forecasts. Not all methods are available for all horizons.
- `simulations` - A non-negative integer indicating the number of simulation to use when `method` is 'simulation' or 'bootstrap'

1.3.5 Understanding Forecast Output

Any call to *forecast()* returns a *ARCHModelForecast* object with has 3 core attributes and 1 which may be useful when using simulation- or bootstrap-based forecasts.

The three core attributes are

- `mean` - The forecast conditional mean.
- `variance` - The forecast conditional variance.
- `residual_variance` - The forecast conditional variance of residuals. This will differ from `variance` whenever the model has dynamics (e.g. an AR model) for horizons larger than 1.

Each attribute contains a `DataFrame` with a common structure.

```
print(forecasts.variance.tail())
```

which returns

```
                h.1       h.2       h.3       h.4       h.5
Date
2013-12-24  0.489534  0.495875  0.501122  0.509194  0.518614
2013-12-26  0.474691  0.480416  0.483664  0.491932  0.502419
2013-12-27  0.447054  0.454875  0.462167  0.467515  0.475632
2013-12-30  0.421528  0.430024  0.439856  0.448282  0.457368
2013-12-31  0.407544  0.415616  0.422848  0.430246  0.439451
```

The values in the columns `h.1` are one-step ahead forecast, while values in `h.2`, . . . , `h.5` are 2, . . . , 5-observation ahead forecasts. The output is aligned so that the Date column is the final data used to generate the forecast, so that `h.1` in row `2013-12-31` is the one-step ahead forecast made using data **up to and including** December 31, 2013.

By default forecasts are only produced for observations after the final observation used to estimate the model.

```
day = dt.timedelta(1)
print(forecasts.variance[split_date - 5 * day:split_date + 5 * day])
```

which produces

```
               h.1       h.2       h.3       h.4       h.5
Date
2009-12-28       NaN       NaN       NaN       NaN       NaN
2009-12-29       NaN       NaN       NaN       NaN       NaN
2009-12-30       NaN       NaN       NaN       NaN       NaN
2009-12-31       NaN       NaN       NaN       NaN       NaN
2010-01-04  0.739303  0.741100  0.744529  0.746940  0.752688
2010-01-05  0.695349  0.702488  0.706812  0.713342  0.721629
2010-01-06  0.649343  0.654048  0.664055  0.672742  0.681263
```

The output will always have as many rows as the data input. Values that are not forecast are `nan` filled.

1.3.6 Output Classes

ARCHModelForecast(index, start_index, mean, ...)	Container for forecasts from an ARCH Model
ARCHModelForecastSimulation(index, values, ...)	Container for a simulation or bootstrap-based forecasts from an ARCH Model

arch.univariate.base.ARCHModelForecast

class arch.univariate.base.**ARCHModelForecast**(index: *list[Label] | pd.Index*, start_index: *int*, mean: *Float64Array*, variance: *Float64Array*, residual_variance: *Float64Array*, simulated_paths: *Float64Array | None* = **None**, simulated_variances: *Float64Array | None* = **None**, simulated_residual_variances: *Float64Array | None* = **None**, simulated_residuals: *Float64Array | None* = **None**, align: *'origin' | 'target'* = `'origin'`, *, reindex: *bool* = ***False***)

Container for forecasts from an ARCH Model

Parameters

index: list[Label] | pd.Index

mean: Float64Array

variance: Float64Array

residual_variance: Float64Array

simulated_paths: Float64Array | None = None

simulated_variances: Float64Array | None = None

simulated_residual_variances: Float64Array | None = None

simulated_residuals: Float64Array | None = None

align: 'origin' | 'target' = 'origin'

Methods

Properties

mean	Forecast values for the conditional mean of the process
residual_variance	Forecast values for the conditional variance of the residuals
simulations	Detailed simulation results if using a simulation-based method
variance	Forecast values for the conditional variance of the process

arch.univariate.base.ARCHModelForecast.mean

property ARCHModelForecast.**mean** : pd.DataFrame

Forecast values for the conditional mean of the process

arch.univariate.base.ARCHModelForecast.residual_variance

property ARCHModelForecast.**residual_variance** : pd.DataFrame

Forecast values for the conditional variance of the residuals

arch.univariate.base.ARCHModelForecast.simulations

property ARCHModelForecast.**simulations** : *ARCHModelForecastSimulation*

Detailed simulation results if using a simulation-based method

Returns

Container for simulation results

Return type

ARCHModelForecastSimulation

arch.univariate.base.ARCHModelForecast.variance

property ARCHModelForecast.**variance** : pd.DataFrame

Forecast values for the conditional variance of the process

arch.univariate.base.ARCHModelForecastSimulation

class arch.univariate.base.**ARCHModelForecastSimulation**(index: *list[Label] | pd.Index*, values: *Float64Array | None*, residuals: *Float64Array | None*, variances: *Float64Array | None*, residual_variances: *Float64Array | None*)

Container for a simulation or bootstrap-based forecasts from an ARCH Model

Parameters

index: list[Label] | pd.Index

values: Float64Array | None

residuals: Float64Array | None

variances: Float64Array | None

residual_variances: Float64Array | None

Methods

Properties

index	The index aligned to dimension 0 of the simulation paths
residual_variances	Simulated variance of the residuals
residuals	Simulated residuals used to produce the values
values	The values of the process
variances	Simulated variances of the values

arch.univariate.base.ARCHModelForecastSimulation.index

property ARCHModelForecastSimulation.**index** : pd.Index

The index aligned to dimension 0 of the simulation paths

arch.univariate.base.ARCHModelForecastSimulation.residual_variances

property ARCHModelForecastSimulation.**residual_variances** : ndarray | None

Simulated variance of the residuals

arch.univariate.base.ARCHModelForecastSimulation.residuals

property ARCHModelForecastSimulation.**residuals** : ndarray | None

Simulated residuals used to produce the values

arch.univariate.base.ARCHModelForecastSimulation.values

property ARCHModelForecastSimulation.**values** : ndarray | None

The values of the process

arch.univariate.base.ARCHModelForecastSimulation.variances

property ARCHModelForecastSimulation.**variances** : ndarray | None

Simulated variances of the values

1.4 Volatility Forecasting

This setup code is required to run in an IPython notebook

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style("darkgrid")
plt.rc("figure", figsize=(16, 6))
plt.rc("savefig", dpi=90)
plt.rc("font", family="sans-serif")
plt.rc("font", size=14)
```

1.4.1 Data

These examples make use of S&P 500 data from Yahoo! that is available from `arch.data.sp500`.

[2]:
```
import datetime as dt
import sys

import arch.data.sp500
import numpy as np
import pandas as pd
from arch import arch_model
```

(continues on next page)

(continued from previous page)

```
data = arch.data.sp500.load()
market = data["Adj Close"]
returns = 100 * market.pct_change().dropna()
```

1.4.2 Basic Forecasting

Forecasts can be generated for standard GARCH(p,q) processes using any of the three forecast generation methods:

- Analytical
- Simulation-based
- Bootstrap-based

Be default forecasts will only be produced for the final observation in the sample so that they are out-of-sample.

Forecasts start with specifying the model and estimating parameters.

[3]:
```
am = arch_model(returns, vol="Garch", p=1, o=0, q=1, dist="Normal")
res = am.fit(update_freq=5)
```

```
Iteration:      5,   Func. Count:     35,   Neg. LLF: 6970.2779565773335
Iteration:     10,   Func. Count:     63,   Neg. LLF: 6936.718477482076
Optimization terminated successfully    (Exit mode 0)
            Current function value: 6936.718476988951
            Iterations: 11
            Function evaluations: 68
            Gradient evaluations: 11
```

[4]:
```
forecasts = res.forecast()
```

Forecasts are contained in an `ARCHModelForecast` object which has 4 attributes:

- `mean` - The forecast means
- `residual_variance` - The forecast residual variances, that is $E_t[\epsilon_{t+h}^2]$
- `variance` - The forecast variance of the process, $E_t[r_{t+h}^2]$. The variance will differ from the residual variance whenever the model has mean dynamics, e.g., in an AR process.
- `simulations` - An object that contains detailed information about the simulations used to generate forecasts. Only used if the forecast `method` is set to `'simulation'` or `'bootstrap'`. If using `'analytical'` (the default), this is `None`.

The three main outputs are all returned in `DataFrame`s with columns of the form `h.#` where `#` is the number of steps ahead. That is, `h.1` corresponds to one-step ahead forecasts while `h.10` corresponds to 10-steps ahead.

The default forecast only produces 1-step ahead forecasts.

[5]:
```
print(forecasts.mean.iloc[-3:])
print(forecasts.residual_variance.iloc[-3:])
print(forecasts.variance.iloc[-3:])
```

```
                 h.1
Date
2018-12-31  0.056353
                h.1
```

(continues on next page)

(continued from previous page)

```
Date
2018-12-31  3.59647
                h.1
Date
2018-12-31  3.59647
```

Longer horizon forecasts can be computed by passing the parameter `horizon`.

[6]:
```
forecasts = res.forecast(horizon=5)
print(forecasts.residual_variance.iloc[-3:])
```

```
                h.1       h.2       h.3       h.4       h.5
Date
2018-12-31  3.59647  3.568502  3.540887  3.513621  3.486701
```

1.4.3 Alternative Forecast Generation Schemes

Fixed Window Forecasting

Fixed-windows forecasting uses data up to a specified date to generate all forecasts after that date. This can be implemented by passing the entire data in when initializing the model and then using `last_obs` when calling `fit`. `forecast()` will, by default, produce forecasts after this final date.

Note `last_obs` follow Python sequence rules so that the actual date in `last_obs` is not in the sample.

[7]:
```
res = am.fit(last_obs="2011-1-1", update_freq=5)
forecasts = res.forecast(horizon=5)
print(forecasts.variance.dropna().head())
```

```
Iteration:      5,   Func. Count:     34,   Neg. LLF: 4578.713295409127
Iteration:     10,   Func. Count:     63,   Neg. LLF: 4555.338451419905
Optimization terminated successfully    (Exit mode 0)
            Current function value: 4555.285110045323
            Iterations: 14
            Function evaluations: 83
            Gradient evaluations: 14
                 h.1       h.2       h.3       h.4       h.5
Date
2010-12-31  0.381757  0.390905  0.399988  0.409008  0.417964
2011-01-03  0.451724  0.460381  0.468976  0.477512  0.485987
2011-01-04  0.428416  0.437236  0.445994  0.454691  0.463326
2011-01-05  0.420554  0.429429  0.438242  0.446993  0.455683
2011-01-06  0.402483  0.411486  0.420425  0.429301  0.438115
```

Rolling Window Forecasting

Rolling window forecasts use a fixed sample length and then produce one-step from the final observation. These can be implemented using `first_obs` and `last_obs`.

[8]:
```
index = returns.index
start_loc = 0
end_loc = np.where(index >= "2010-1-1")[0].min()
forecasts = {}
for i in range(20):
    sys.stdout.write(".")
    sys.stdout.flush()
    res = am.fit(first_obs=i, last_obs=i + end_loc, disp="off")
    temp = res.forecast(horizon=3).variance
    fcast = temp.iloc[0]
    forecasts[fcast.name] = fcast
print()
print(pd.DataFrame(forecasts).T)
```

```
...
                 h.1       h.2       h.3
2009-12-31  0.615314  0.621743  0.628133
2010-01-04  0.751747  0.757343  0.762905
2010-01-05  0.710453  0.716315  0.722142
2010-01-06  0.666244  0.672346  0.678411
2010-01-07  0.634424  0.640706  0.646949
2010-01-08  0.600109  0.606595  0.613040
2010-01-11  0.565514  0.572212  0.578869
2010-01-12  0.599561  0.606051  0.612501
2010-01-13  0.608309  0.614748  0.621148
2010-01-14  0.575065  0.581756  0.588406
2010-01-15  0.629890  0.636245  0.642561
2010-01-19  0.695074  0.701042  0.706974
2010-01-20  0.737154  0.742908  0.748627
2010-01-21  0.954167  0.958725  0.963255
2010-01-22  1.253453  1.256401  1.259332
2010-01-25  1.178691  1.182043  1.185374
2010-01-26  1.112205  1.115886  1.119545
2010-01-27  1.051295  1.055327  1.059335
2010-01-28  1.085678  1.089512  1.093324
2010-01-29  1.085786  1.089594  1.093378
```

Recursive Forecast Generation

Recursive is similar to rolling except that the initial observation does not change. This can be easily implemented by dropping the `first_obs` input.

[9]:
```
import numpy as np
import pandas as pd

index = returns.index
start_loc = 0
end_loc = np.where(index >= "2010-1-1")[0].min()
```

(continues on next page)

(continued from previous page)

```
forecasts = {}
for i in range(20):
    sys.stdout.write(".")
    sys.stdout.flush()
    res = am.fit(last_obs=i + end_loc, disp="off")
    temp = res.forecast(horizon=3).variance
    fcast = temp.iloc[0]
    forecasts[fcast.name] = fcast
print()
print(pd.DataFrame(forecasts).T)
```

```
...
                 h.1       h.2       h.3
2009-12-31  0.615314  0.621743  0.628133
2010-01-04  0.751723  0.757321  0.762885
2010-01-05  0.709956  0.715791  0.721591
2010-01-06  0.666057  0.672146  0.678197
2010-01-07  0.634503  0.640776  0.647011
2010-01-08  0.600417  0.606893  0.613329
2010-01-11  0.565684  0.572369  0.579014
2010-01-12  0.599963  0.606438  0.612874
2010-01-13  0.608558  0.614982  0.621366
2010-01-14  0.575020  0.581639  0.588217
2010-01-15  0.629696  0.635989  0.642244
2010-01-19  0.694735  0.700656  0.706541
2010-01-20  0.736509  0.742193  0.747842
2010-01-21  0.952751  0.957246  0.961713
2010-01-22  1.251145  1.254050  1.256936
2010-01-25  1.176864  1.180162  1.183441
2010-01-26  1.110848  1.114497  1.118124
2010-01-27  1.050102  1.054077  1.058028
2010-01-28  1.084669  1.088454  1.092216
2010-01-29  1.085003  1.088783  1.092541
```

1.4.4 TARCH

Analytical Forecasts

All ARCH-type models have one-step analytical forecasts. Longer horizons only have closed forms for specific models. TARCH models do not have closed-form (analytical) forecasts for horizons larger than 1, and so simulation or bootstrapping is required. Attempting to produce forecasts for horizons larger than 1 using `method='analytical'` results in a `ValueError`.

[10]:
```
# TARCH specification
am = arch_model(returns, vol="GARCH", power=2.0, p=1, o=1, q=1)
res = am.fit(update_freq=5)
forecasts = res.forecast()
print(forecasts.variance.iloc[-1])
```

```
Iteration:      5,   Func. Count:     40,   Neg. LLF: 6846.48928243015
Iteration:     10,   Func. Count:     75,   Neg. LLF: 6822.883188650879
Optimization terminated successfully    (Exit mode 0)
```

(continues on next page)

(continued from previous page)

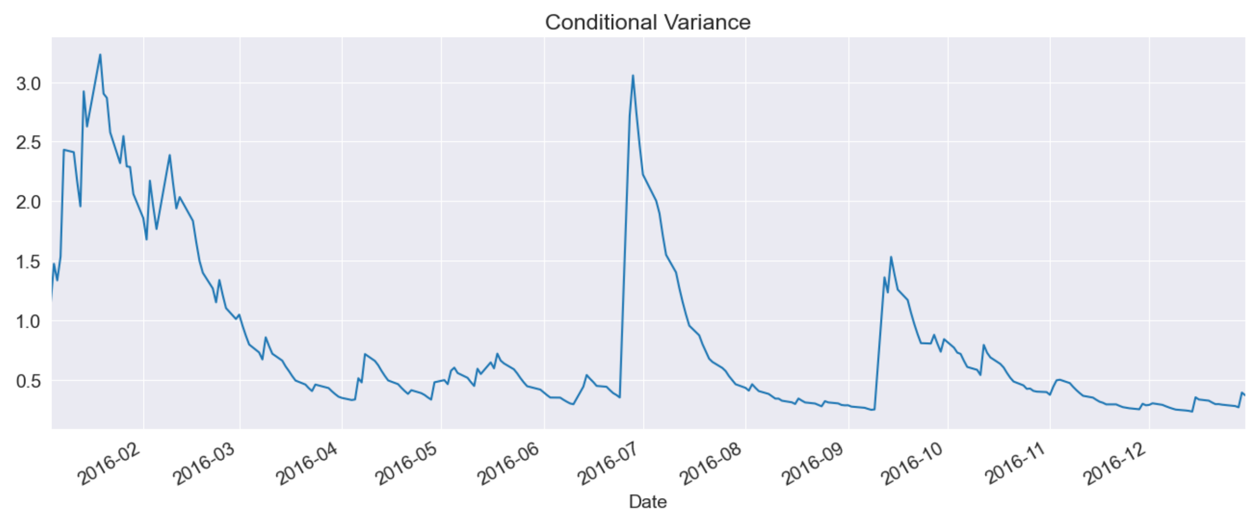
```
            Current function value: 6822.882823441358
            Iterations: 13
            Function evaluations: 93
            Gradient evaluations: 13
h.1    3.010188
Name: 2018-12-31 00:00:00, dtype: float64
```

Simulation Forecasts

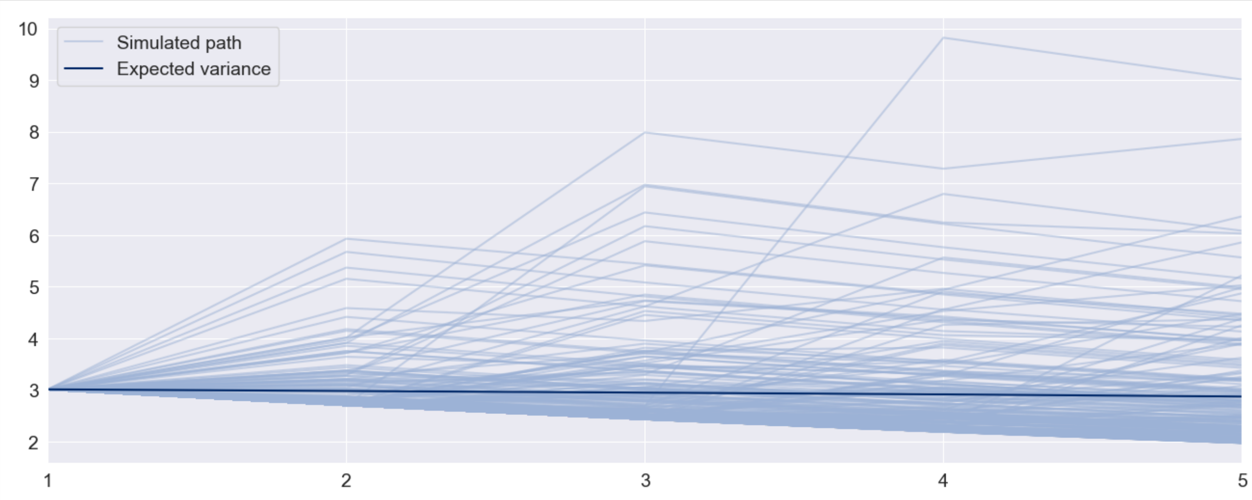
When using simulation- or bootstrap-based forecasts, an additional attribute of an `ARCHModelForecast` object is meaningful – `simulation`.

[11]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 1)
var_2016 = res.conditional_volatility["2016"] ** 2.0
subplot = var_2016.plot(ax=ax, title="Conditional Variance")
subplot.set_xlim(var_2016.index[0], var_2016.index[-1])
```

[11]:
```
(16804.0, 17165.0)
```

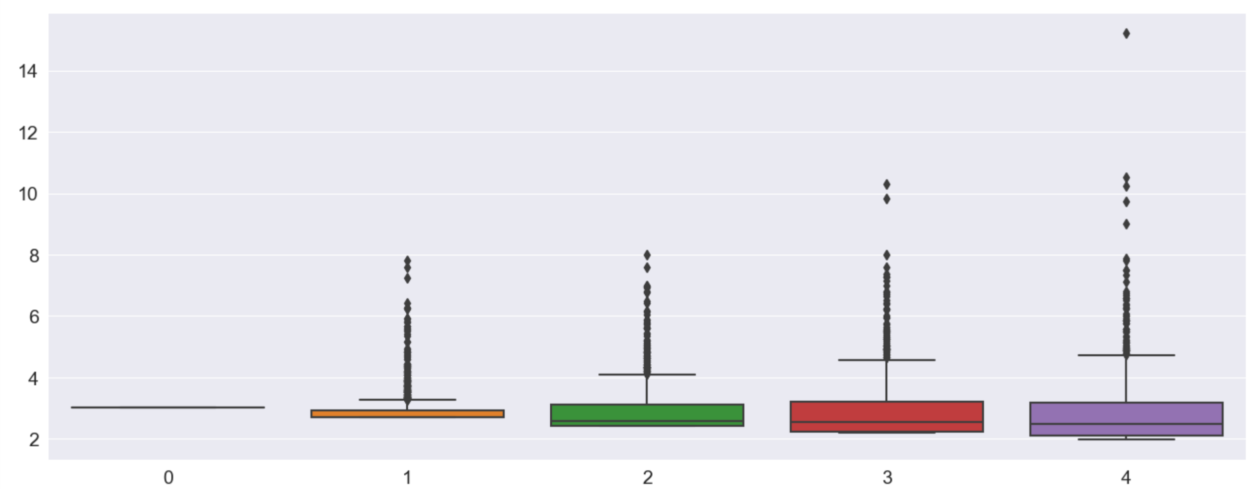


[12]:
```
forecasts = res.forecast(horizon=5, method="simulation")
sims = forecasts.simulations

x = np.arange(1, 6)
lines = plt.plot(x, sims.residual_variances[-1, ::5].T, color="#9cb2d6", alpha=0.5)
lines[0].set_label("Simulated path")
line = plt.plot(x, forecasts.variance.iloc[-1].values, color="#002868")
line[0].set_label("Expected variance")
plt.gca().set_xticks(x)
plt.gca().set_xlim(1, 5)
legend = plt.legend()
```




```
[13]: import seaborn as sns

sns.boxplot(data=sims.variances[-1])
```

```
[13]: <Axes: >
```



Bootstrap Forecasts

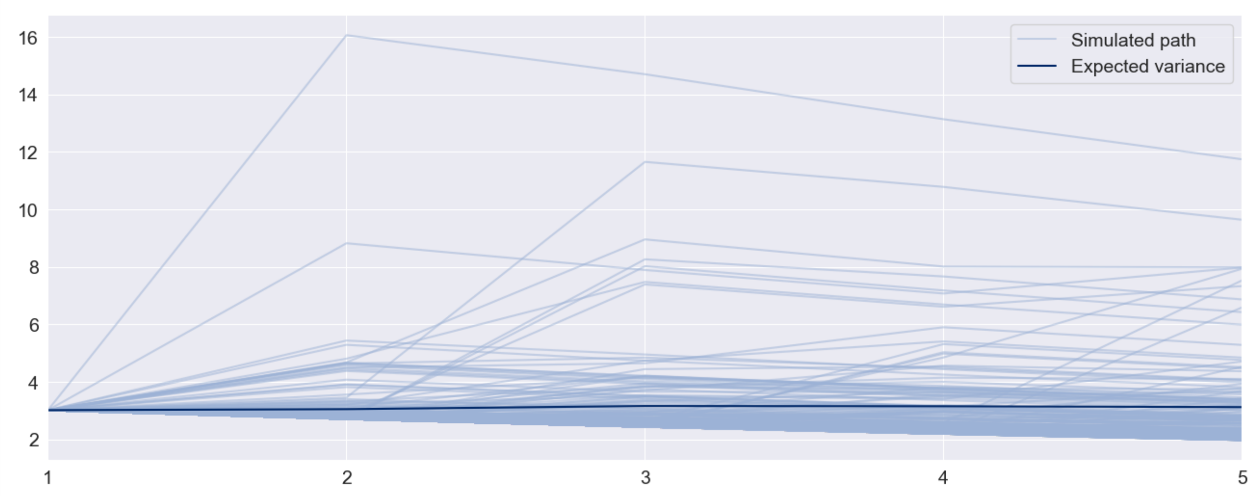
Bootstrap-based forecasts are nearly identical to simulation-based forecasts except that the values used to simulate the process are computed from historical data rather than using the assumed distribution of the residuals. Forecasts produced using this method also return an `ARCHModelForecastSimulation` containing information about the simulated paths.

```
[14]: forecasts = res.forecast(horizon=5, method="bootstrap")
sims = forecasts.simulations

lines = plt.plot(x, sims.residual_variances[-1, ::5].T, color="#9cb2d6", alpha=0.5)
lines[0].set_label("Simulated path")
line = plt.plot(x, forecasts.variance.iloc[-1].values, color="#002868")
```

(continues on next page)

(continued from previous page)

```
line[0].set_label("Expected variance")
plt.gca().set_xticks(x)
plt.gca().set_xlim(1, 5)
legend = plt.legend()
```



1.5 Value-at-Risk Forecasting

Value-at-Risk (VaR) forecasts from GARCH models depend on the conditional mean, the conditional volatility and the quantile of the standardized residuals,

$$VaR_{t+1|t} = -\mu_{t+1|t} - \sigma_{t+1|t} q_{\alpha}$$

where q_{α} is the α quantile of the standardized residuals, e.g., 5%.

The quantile can be either computed from the estimated model density or computed using the empirical distribution of the standardized residuals. The example below shows both methods.

[15]:
```
am = arch_model(returns, vol="Garch", p=1, o=0, q=1, dist="skewt")
res = am.fit(disp="off", last_obs="2017-12-31")
```

1.5.1 Parametric VaR

First, we use the model to estimate the VaR. The quantiles can be computed using the `ppf` method of the distribution attached to the model. The quantiles are printed below.

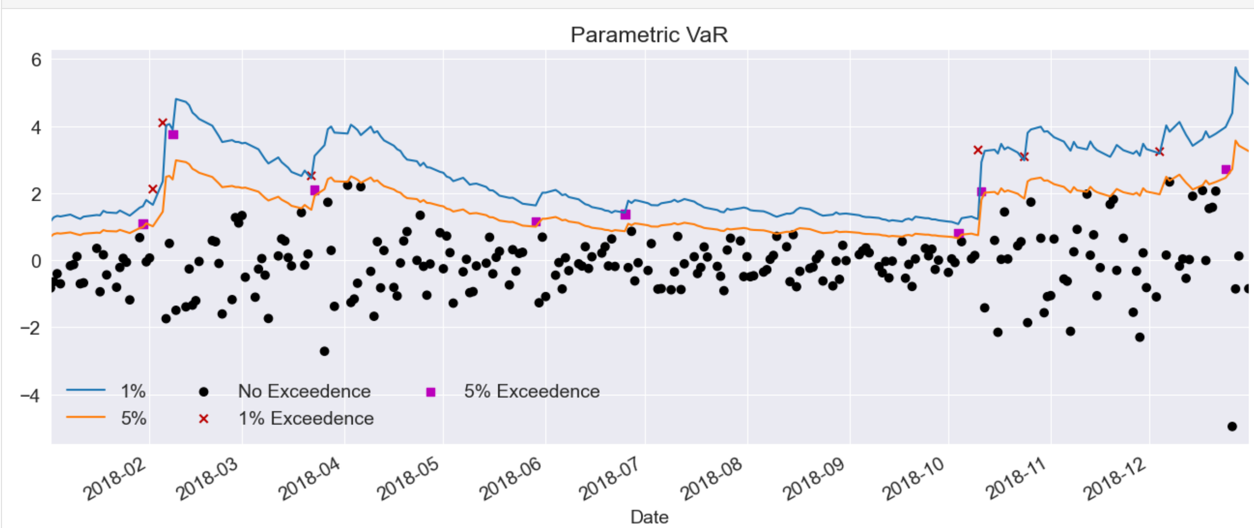
Note: `forecast` is called with `align="target"` so that the forecasts are already aligned with the target and so do not need further shifting.

[16]:
```
forecasts = res.forecast(align="target")
cond_mean = forecasts.mean["2018":].dropna()
cond_var = forecasts.variance["2018":].dropna()
q = am.distribution.ppf([0.01, 0.05], res.params[-2:])
print(q)
```

```
[-2.64484999 -1.64965918]
```

Next, we plot the two VaRs along with the returns. The returns that violate the VaR forecasts are highlighted.

[17]:
```
value_at_risk = -cond_mean.values - np.sqrt(cond_var).values * q[None, :]
value_at_risk = pd.DataFrame(value_at_risk, columns=["1%", "5%"], index=cond_var.index)
ax = value_at_risk.plot(legend=False)
xl = ax.set_xlim(value_at_risk.index[0], value_at_risk.index[-1])
rets_2018 = returns["2018":].copy()
rets_2018.name = "S&P 500 Return"
c = []
for idx in value_at_risk.index:
    if rets_2018[idx] > -value_at_risk.loc[idx, "5%"]:
        c.append("#000000")
    elif rets_2018[idx] < -value_at_risk.loc[idx, "1%"]:
        c.append("#BB0000")
    else:
        c.append("#BB00BB")
c = np.array(c, dtype="object")
labels = {
    "#BB0000": "1% Exceedence",
    "#BB00BB": "5% Exceedence",
    "#000000": "No Exceedence",
}
markers = {"#BB0000": "x", "#BB00BB": "s", "#000000": "o"}
for color in np.unique(c):
    sel = c == color
    ax.scatter(
        rets_2018.index[sel],
        -rets_2018.loc[sel],
        marker=markers[color],
        c=c[sel],
        label=labels[color],
    )
ax.set_title("Parametric VaR")
leg = ax.legend(frameon=False, ncol=3)
```



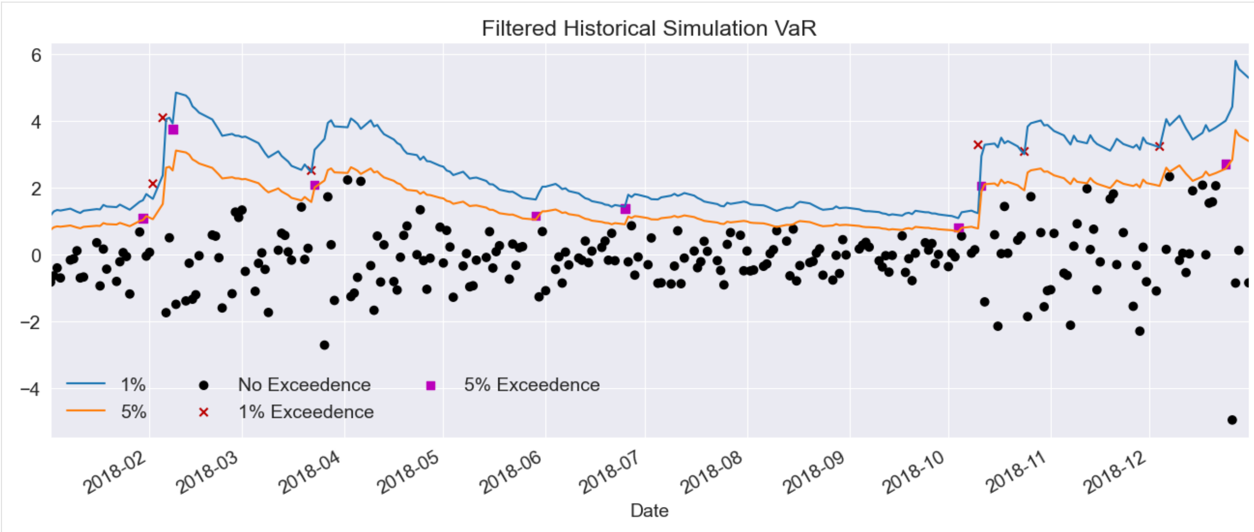
1.5.2 Filtered Historical Simulation

Next, we use the empirical distribution of the standardized residuals to estimate the quantiles. These values are very similar to those estimated using the assumed distribution. The plot below is identical except for the slightly different quantiles.

[18]:
```
std_rets = (returns[:"2017"] - res.params["mu"]) / res.conditional_volatility
std_rets = std_rets.dropna()
q = std_rets.quantile([0.01, 0.05])
print(q)
```

```
0.01   -2.668273
0.05   -1.723352
dtype: float64
```

[19]:
```
value_at_risk = -cond_mean.values - np.sqrt(cond_var).values * q.values[None, :]
value_at_risk = pd.DataFrame(value_at_risk, columns=["1%", "5%"], index=cond_var.index)
ax = value_at_risk.plot(legend=False)
xl = ax.set_xlim(value_at_risk.index[0], value_at_risk.index[-1])
rets_2018 = returns["2018":].copy()
rets_2018.name = "S&P 500 Return"
c = []
for idx in value_at_risk.index:
    if rets_2018[idx] > -value_at_risk.loc[idx, "5%"]:
        c.append("#000000")
    elif rets_2018[idx] < -value_at_risk.loc[idx, "1%"]:
        c.append("#BB0000")
    else:
        c.append("#BB00BB")
c = np.array(c, dtype="object")
for color in np.unique(c):
    sel = c == color
    ax.scatter(
        rets_2018.index[sel],
        -rets_2018.loc[sel],
        marker=markers[color],
        c=c[sel],
        label=labels[color],
    )
ax.set_title("Filtered Historical Simulation VaR")
leg = ax.legend(frameon=False, ncol=3)
```



1.6 Volatility Scenarios

Custom random-number generators can be used to implement scenarios where shock follow a particular pattern. For example, suppose you wanted to find out what would happen if there were 5 days of shocks that were larger than average. In most circumstances, the shocks in a GARCH model have unit variance. This could be changed so that the first 5 shocks have variance 4, or twice the standard deviation.

Another scenario would be to over sample a specific period for the shocks. When using the standard bootstrap method (filtered historical simulation) the shocks are drawn using iid sampling from the history. While this approach is standard and well-grounded, it might be desirable to sample from a specific period. This can be implemented using a custom random number generator. This strategy is precisely how the filtered historical simulation is implemented internally, only where the draws are uniformly sampled from the entire history.

First, some preliminaries

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from arch.univariate import GARCH, ConstantMean, Normal

sns.set_style("darkgrid")
plt.rc("figure", figsize=(16, 6))
plt.rc("savefig", dpi=90)
plt.rc("font", family="sans-serif")
plt.rc("font", size=14)
```

This example makes use of returns from the NASDAQ index. The scenario bootstrap will make use of returns in the run-up to and during the Financial Crisis of 2008.

[2]:
```
import arch.data.nasdaq

data = arch.data.nasdaq.load()
```

(continues on next page)

(continued from previous page)

```
nasdaq = data["Adj Close"]
print(nasdaq.head())
```

```
Date
1999-01-04    2208.050049
1999-01-05    2251.270020
1999-01-06    2320.860107
1999-01-07    2326.090088
1999-01-08    2344.409912
Name: Adj Close, dtype: float64
```

Next, the returns are computed and the model is constructed. The model is constructed from the building blocks. It is a standard model and could have been (almost) equivalently constructed using

```
mod = arch_model(rets, mean='constant', p=1, o=1, q=1)
```

The one advantage of constructing the model using the components is that the NumPy `RandomState` that is used to simulate from the model can be externally set. This allows the generator seed to be easily set and for the state to reset, if needed.

NOTE: It is always a good idea to scale return by 100 before estimating ARCH-type models. This helps the optimizer converse since the scale of the volatility intercept is much closer to the scale of the other parameters in the model.

[3]:
```
rets = 100 * nasdaq.pct_change().dropna()

# Build components to set the state for the distribution
random_state = np.random.RandomState(1)
dist = Normal(seed=random_state)
volatility = GARCH(1, 1, 1)

mod = ConstantMean(rets, volatility=volatility, distribution=dist)
```

Fitting the model is standard.

[4]:
```
res = mod.fit(disp="off")
res
```

[4]:
```
                   Constant Mean - GJR-GARCH Model Results
==============================================================================
Dep. Variable:              Adj Close   R-squared:                       0.000
Mean Model:             Constant Mean   Adj. R-squared:                  0.000
Vol Model:                  GJR-GARCH   Log-Likelihood:               -8196.75
Distribution:                  Normal   AIC:                           16403.5
Method:            Maximum Likelihood   BIC:                           16436.1
                                        No. Observations:                 5030
Date:                Tue, May 30 2023   Df Residuals:                     5029
Time:                        10:55:35   Df Model:                            1
                                 Mean Model
============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
----------------------------------------------------------------------------
mu             0.0376  1.476e-02      2.549  1.081e-02 [8.693e-03,6.656e-02]
                               Volatility Model
=============================================================================
```

(continues on next page)

(continued from previous page)

```
                  coef    std err          t      P>|t|      95.0% Conf. Int.
-----------------------------------------------------------------------------
omega           0.0214  5.001e-03      4.281  1.861e-05  [1.161e-02,3.121e-02]
alpha[1]        0.0152  8.442e-03      1.802  7.148e-02 [-1.330e-03,3.176e-02]
gamma[1]        0.1265  2.024e-02      6.250  4.109e-10    [8.684e-02,  0.166]
beta[1]         0.9100  1.107e-02     82.232      0.000      [  0.888,  0.932]
=============================================================================

Covariance estimator: robust
ARCHModelResult, id: 0x202821906a0
```

GJR-GARCH models support analytical forecasts, which is the default. The forecasts are produced for all of 2017 using the estimated model parameters.

All GARCH specification are complete models in the sense that they specify a distribution. This allows simulations to be produced using the assumptions in the model. The `forecast` function can be made to produce simulations using the assumed distribution by setting `method='simulation'`.

These forecasts are similar to the analytical forecasts above. As the number of simulation increases towards ∞, the simulation-based forecasts will converge to the analytical values above.

[5]:
```
sim_forecasts = res.forecast(start="1-1-2017", method="simulation", horizon=10)
print(sim_forecasts.residual_variance.dropna().head())
```

```
                h.01      h.02      h.03      h.04      h.05      h.06  \
Date
2017-01-03  0.623295  0.637251  0.647817  0.663746  0.673404  0.687952
2017-01-04  0.599455  0.617539  0.635838  0.649695  0.659733  0.667267
2017-01-05  0.567297  0.583415  0.597571  0.613065  0.621790  0.636180
2017-01-06  0.542506  0.555688  0.570280  0.585426  0.595551  0.608487
2017-01-09  0.515452  0.528771  0.542658  0.559684  0.580434  0.594855

                h.07      h.08      h.09      h.10
Date
2017-01-03  0.697221  0.707707  0.717701  0.729465
2017-01-04  0.686503  0.699708  0.707203  0.718560
2017-01-05  0.650287  0.663344  0.679835  0.692300
2017-01-06  0.619195  0.638180  0.653185  0.661366
2017-01-09  0.605136  0.621835  0.634091  0.653222
```

1.6.1 Custom Random Generators

`forecast` supports replacing the generator based on the assumed distribution of residuals in the model with any other generator. A shock generator should usually produce unit variance shocks. However, in this example the first 5 shocks generated have variance 2, and the remainder are standard normal. This scenario consists of a period of consistently surprising volatility where the volatility has shifted for some reason.

The forecast variances are much larger and grow faster than those from either method previously illustrated. This reflects the increase in volatility in the first 5 days.

[6]:
```
forecasts = res.forecast(start="1-1-2017", horizon=10)
print(forecasts.residual_variance.dropna().head())
```

```
                h.01      h.02      h.03      h.04      h.05      h.06  \
Date
2017-01-03  0.623295  0.637504  0.651549  0.665431  0.679154  0.692717
2017-01-04  0.599455  0.613940  0.628257  0.642408  0.656397  0.670223
2017-01-05  0.567297  0.582153  0.596837  0.611352  0.625699  0.639880
2017-01-06  0.542506  0.557649  0.572616  0.587410  0.602034  0.616488
2017-01-09  0.515452  0.530906  0.546183  0.561282  0.576208  0.590961

                h.07      h.08      h.09      h.10
Date
2017-01-03  0.706124  0.719376  0.732475  0.745423
2017-01-04  0.683890  0.697399  0.710752  0.723950
2017-01-05  0.653897  0.667753  0.681448  0.694985
2017-01-06  0.630776  0.644899  0.658858  0.672656
2017-01-09  0.605543  0.619957  0.634205  0.648288
```

[7]:
```python
import numpy as np

random_state = np.random.RandomState(1)


def scenario_rng(size):
    shocks = random_state.standard_normal(size)
    shocks[:, :5] *= np.sqrt(2)
    return shocks


scenario_forecasts = res.forecast(
    start="1-1-2017", method="simulation", horizon=10, rng=scenario_rng
)
print(scenario_forecasts.residual_variance.dropna().head())
```

```
                h.01      h.02      h.03      h.04      h.05      h.06  \
Date
2017-01-03  0.623295  0.685911  0.745202  0.821112  0.886289  0.966737
2017-01-04  0.599455  0.668181  0.743119  0.811486  0.877539  0.936587
2017-01-05  0.567297  0.629195  0.691225  0.758891  0.816663  0.893986
2017-01-06  0.542506  0.596301  0.656603  0.721505  0.778286  0.849680
2017-01-09  0.515452  0.567086  0.622224  0.689831  0.775048  0.845656

                h.07      h.08      h.09      h.10
Date
2017-01-03  0.970796  0.977504  0.982202  0.992547
2017-01-04  0.955295  0.965540  0.966432  0.974248
2017-01-05  0.905952  0.915208  0.930777  0.938636
2017-01-06  0.856175  0.873865  0.886221  0.890002
2017-01-09  0.851104  0.864591  0.874696  0.894397
```

1.6.2 Bootstrap Scenarios

`forecast` supports Filtered Historical Simulation (FHS) using `method='bootstrap'`. This is effectively a simulation method where the simulated shocks are generated using iid sampling from the history of the demeaned and standardized return data. Custom bootstraps are another application of `rng`. Here an object is used to hold the shocks. This object exposes a method (`rng`) the acts like a random number generator, except that it only returns values that were provided in the `shocks` parameter.

The internal implementation of the FHS uses a method almost identical to this where `shocks` contain the entire history.

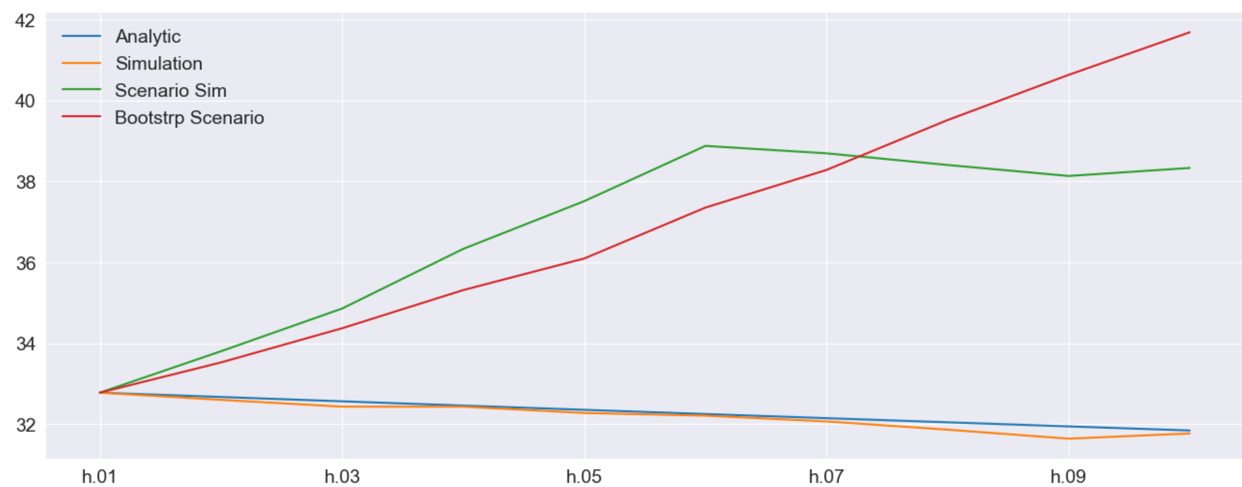
[8]:
```
class ScenarioBootstrapRNG(object):
    def __init__(self, shocks, random_state):
        self._shocks = np.asarray(shocks)  # 1d
        self._rs = random_state
        self.n = shocks.shape[0]

    def rng(self, size):
        idx = self._rs.randint(0, self.n, size=size)
        return self._shocks[idx]


random_state = np.random.RandomState(1)
std_shocks = res.resid / res.conditional_volatility
shocks = std_shocks["2008-08-01":"2008-11-10"]
scenario_bootstrap = ScenarioBootstrapRNG(shocks, random_state)
bs_forecasts = res.forecast(
    start="1-1-2017", method="simulation", horizon=10, rng=scenario_bootstrap.rng
)
print(bs_forecasts.residual_variance.dropna().head())
```

```
                h.01      h.02      h.03      h.04      h.05      h.06  \
Date
2017-01-03  0.623295  0.676081  0.734322  0.779325  0.828189  0.898202
2017-01-04  0.599455  0.645237  0.697133  0.750169  0.816280  0.888417
2017-01-05  0.567297  0.610493  0.665995  0.722954  0.777860  0.840369
2017-01-06  0.542506  0.597387  0.644534  0.691387  0.741206  0.783319
2017-01-09  0.515452  0.561312  0.611026  0.647824  0.700559  0.757398

                h.07      h.08      h.09      h.10
Date
2017-01-03  0.958215  1.043704  1.124684  1.203893
2017-01-04  0.945120  1.013400  1.084042  1.158148
2017-01-05  0.889032  0.961424  1.022412  1.097192
2017-01-06  0.840667  0.895559  0.957266  1.019497
2017-01-09  0.820788  0.887791  0.938708  1.028614
```

1.6.3 Visualizing the differences

The final forecast values are used to illustrate how these are different. The analytical and standard simulation are virtually identical. The simulated scenario grows rapidly for the first 5 periods and then more slowly. The bootstrap scenario grows quickly and consistently due to the magnitude of the shocks in the financial crisis.

[9]:
```
import pandas as pd

df = pd.concat(
    [
        forecasts.residual_variance.iloc[-1],
        sim_forecasts.residual_variance.iloc[-1],
        scenario_forecasts.residual_variance.iloc[-1],
        bs_forecasts.residual_variance.iloc[-1],
    ],
    axis=1,
)
df.columns = ["Analytic", "Simulation", "Scenario Sim", "Bootstrp Scenario"]
# Plot annualized vol
subplot = np.sqrt(252 * df).plot(legend=False)
legend = subplot.legend(frameon=False)
```



[10]:
```
subplot = np.sqrt(252 * df).plot
```

1.6.4 Comparing the paths

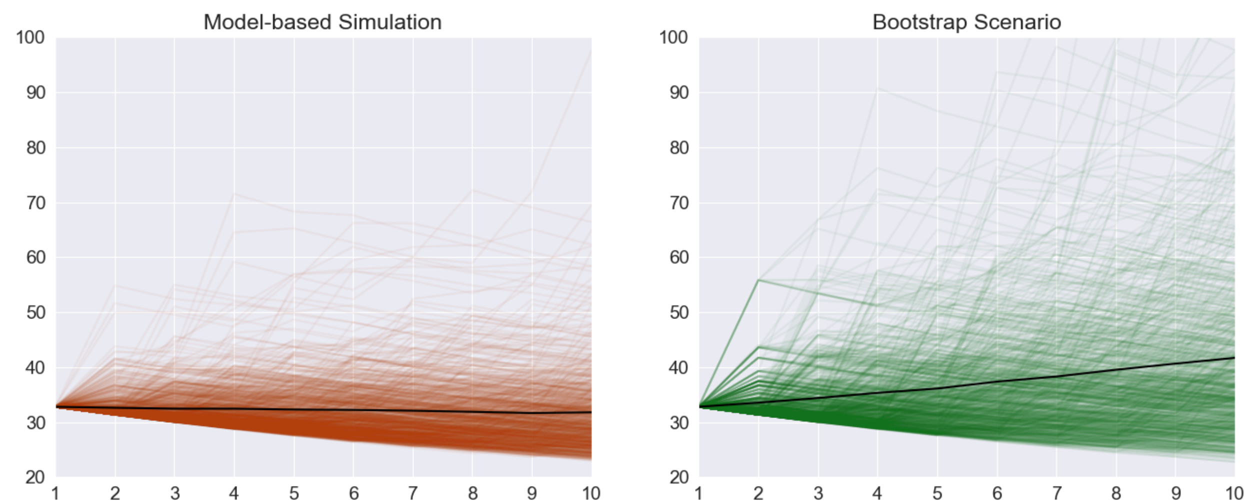
The paths are available on the attribute `simulations`. Plotting the paths shows important differences between the two scenarios beyond the average differences plotted above. Both start at the same point.

[11]:
```
fig, axes = plt.subplots(1, 2)
colors = sns.color_palette("dark")
# The paths for the final observation
sim_paths = sim_forecasts.simulations.residual_variances[-1].T
bs_paths = bs_forecasts.simulations.residual_variances[-1].T

x = np.arange(1, 11)
```

(continues on next page)

(continued from previous page)

```
# Plot the paths and the mean, set the axis to have the same limit
axes[0].plot(x, np.sqrt(252 * sim_paths), color=colors[1], alpha=0.05)
axes[0].plot(
    x, np.sqrt(252 * sim_forecasts.residual_variance.iloc[-1]), color="k", alpha=1
)
axes[0].set_title("Model-based Simulation")
axes[0].set_xticks(np.arange(1, 11))
axes[0].set_xlim(1, 10)
axes[0].set_ylim(20, 100)

axes[1].plot(x, np.sqrt(252 * bs_paths), color=colors[2], alpha=0.05)
axes[1].plot(
    x, np.sqrt(252 * bs_forecasts.residual_variance.iloc[-1]), color="k", alpha=1
)
axes[1].set_xticks(np.arange(1, 11))
axes[1].set_xlim(1, 10)
axes[1].set_ylim(20, 100)
title = axes[1].set_title("Bootstrap Scenario")
```



1.6.5 Comparing across the year

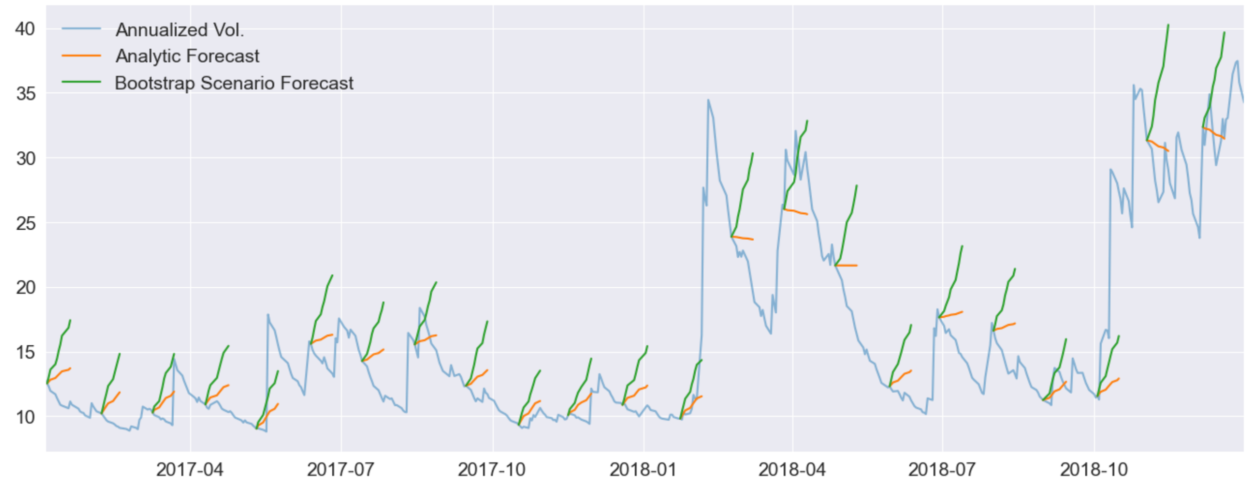
A hedgehog plot is useful for showing the differences between the two forecasting methods across the year, instead of a single day.

[12]:
```
analytic = forecasts.residual_variance.dropna()
bs = bs_forecasts.residual_variance.dropna()
fig, ax = plt.subplots(1, 1)
vol = res.conditional_volatility["2017-1-1":"2019-1-1"]
idx = vol.index
ax.plot(np.sqrt(252) * vol, alpha=0.5)
colors = sns.color_palette()
for i in range(0, len(vol), 22):
    a = analytic.iloc[i]
    b = bs.iloc[i]
```

(continues on next page)

(continued from previous page)

```
    loc = idx.get_loc(a.name)
    new_idx = idx[loc + 1 : loc + 11]
    a.index = new_idx
    b.index = new_idx
    ax.plot(np.sqrt(252 * a), color=colors[1])
    ax.plot(np.sqrt(252 * b), color=colors[2])
labels = ["Annualized Vol.", "Analytic Forecast", "Bootstrap Scenario Forecast"]
legend = ax.legend(labels, frameon=False)
xlim = ax.set_xlim(vol.index[0], vol.index[-1])
```



1.7 Forecasting with Exogenous Regressors

This notebook provides examples of the accepted data structures for passing the expected value of exogenous variables when these are included in the mean. For example, consider an AR(1) with 2 exogenous variables. The mean dynamics are

$$Y_t = \phi_0 + \phi_1 Y_{t-1} + \beta_0 X_{0,t} + \beta_1 X_{1,t} + \epsilon_t.$$

The h-step forecast, $E_T[Y_{t+h}]$, depends on the conditional expectation of $X_{0,T+h}$ and $X_{1,T+h}$,

$$E_T[Y_{T+h}] = \phi_0 + \phi_1 E_T[Y_{T+h-1}] + \beta_0 E_T[X_{0,T+h}] + \beta_1 E_T[X_{1,T+h}]$$

where $E_T[Y_{T+h-1}]$ has been recursively computed.

In order to construct forecasts up to some horizon h, it is necessary to pass $2 \times h$ values (h for each series). If using the features of `forecast` that allow many forecast to be specified, it necessary to supply $n \times 2 \times h$ values.

There are two general purpose data structures that can be used for any number of exogenous variables and any number steps ahead:

- `dict` - The values can be pass using a `dict` where the keys are the variable names and the values are 2-dimensional arrays. This is the most natural generalization of a pandas `DataFrame` to 3-dimensions.
- `array` - The vales can alternatively be passed as a 3-d NumPy `array` where dimension 0 tracks the regressor index, dimension 1 is the time period and dimension 2 is the horizon.

When a model contains a single exogenous regressor it is possible to use a 2-d array or `DataFrame` where dim0 tracks the time period where the forecast is generated and dimension 1 tracks the horizon.

In the special case where a model contains a single regressor *and* the horizon is 1, then a 1-d array of pandas Series can be used.

[1]:
```
# initial setup
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn

seaborn.set_style("darkgrid")
plt.rc("figure", figsize=(16, 6))
plt.rc("savefig", dpi=90)
plt.rc("font", family="sans-serif")
plt.rc("font", size=14)
```

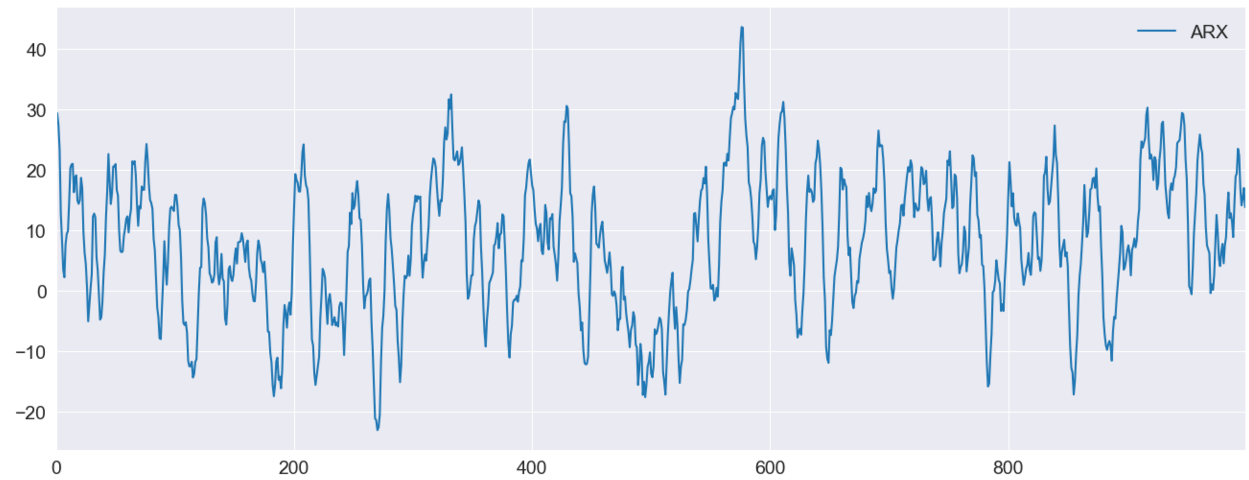
1.7.1 Simulating data

Two X variables are simulated and are assumed to follow independent AR(1) processes. The data is then assumed to follow an ARX(1) with 2 exogenous regressors and GARCH(1,1) errors.

[2]:
```
from arch.univariate import ARX, GARCH, ZeroMean, arch_model

burn = 250

x_mod = ARX(None, lags=1)
x0 = x_mod.simulate([1, 0.8, 1], nobs=1000 + burn).data
x1 = x_mod.simulate([2.5, 0.5, 1], nobs=1000 + burn).data

resid_mod = ZeroMean(volatility=GARCH())
resids = resid_mod.simulate([0.1, 0.1, 0.8], nobs=1000 + burn).data

phi1 = 0.7
phi0 = 3
y = 10 + resids.copy()
for i in range(1, y.shape[0]):
    y[i] = phi0 + phi1 * y[i - 1] + 2 * x0[i] - 2 * x1[i] + resids[i]

x0 = x0.iloc[-1000:]
x1 = x1.iloc[-1000:]
y = y.iloc[-1000:]
y.index = x0.index = x1.index = np.arange(1000)
```

1.7.2 Plotting the data

[3]:
```
ax = pd.DataFrame({"ARX": y}).plot(legend=False)
ax.legend(frameon=False)
_ = ax.set_xlim(0, 999)
```



1.7.3 Forecasting the X values

The forecasts of Y depend on forecasts of X_0 and X_1. Both of these follow simple AR(1), and so we can construct the forecasts for all time horizons. Note that the value in position `[i,j]` is the time-`i` forecast for horizon `j+1`.

[4]:
```
x0_oos = np.empty((1000, 10))
x1_oos = np.empty((1000, 10))
for i in range(10):
    if i == 0:
        last = x0
    else:
        last = x0_oos[:, i - 1]
    x0_oos[:, i] = 1 + 0.8 * last
    if i == 0:
        last = x1
    else:
        last = x1_oos[:, i - 1]
    x1_oos[:, i] = 2.5 + 0.5 * last

x1_oos[-1]
```

[4]:
```
array([5.65805541, 5.3290277 , 5.16451385, 5.08225693, 5.04112846,
       5.02056423, 5.01028212, 5.00514106, 5.00257053, 5.00128526])
```

1.7.4 Fitting the model

Next, the most is fit. The parameters are accurately estimated.

[5]:
```
exog = pd.DataFrame({"x0": x0, "x1": x1})
mod = arch_model(y, x=exog, mean="ARX", lags=1)
res = mod.fit(disp="off")
print(res.summary())
```

```
                          AR-X - GARCH Model Results
==============================================================================
Dep. Variable:                   data   R-squared:                       0.992
Mean Model:                      AR-X   Adj. R-squared:                  0.992
Vol Model:                      GARCH   Log-Likelihood:               -1391.04
Distribution:                  Normal   AIC:                           2796.08
Method:            Maximum Likelihood   BIC:                           2830.43
                                        No. Observations:                  999
Date:                Tue, May 30 2023   Df Residuals:                      995
Time:                        10:55:41   Df Model:                            4
                               Mean Model
========================================================================
                 coef    std err          t      P>|t|   95.0% Conf. Int.
------------------------------------------------------------------------
Const          2.9447      0.151     19.443  3.308e-84 [  2.648,  3.242]
data[1]        0.6947  3.807e-03    182.482      0.000 [  0.687,  0.702]
x0             1.9813  2.175e-02     91.078      0.000 [  1.939,  2.024]
x1            -1.9620  2.564e-02    -76.519      0.000 [ -2.012, -1.912]
                              Volatility Model
==========================================================================
                 coef    std err          t      P>|t|     95.0% Conf. Int.
--------------------------------------------------------------------------
omega          0.0946  3.606e-02      2.622  8.733e-03 [2.389e-02,  0.165]
alpha[1]       0.0974  2.496e-02      3.903  9.516e-05 [4.849e-02,  0.146]
beta[1]        0.8083  5.059e-02     15.977  1.839e-57   [  0.709,  0.907]
==========================================================================

Covariance estimator: robust
```

1.7.5 Using a `dict`

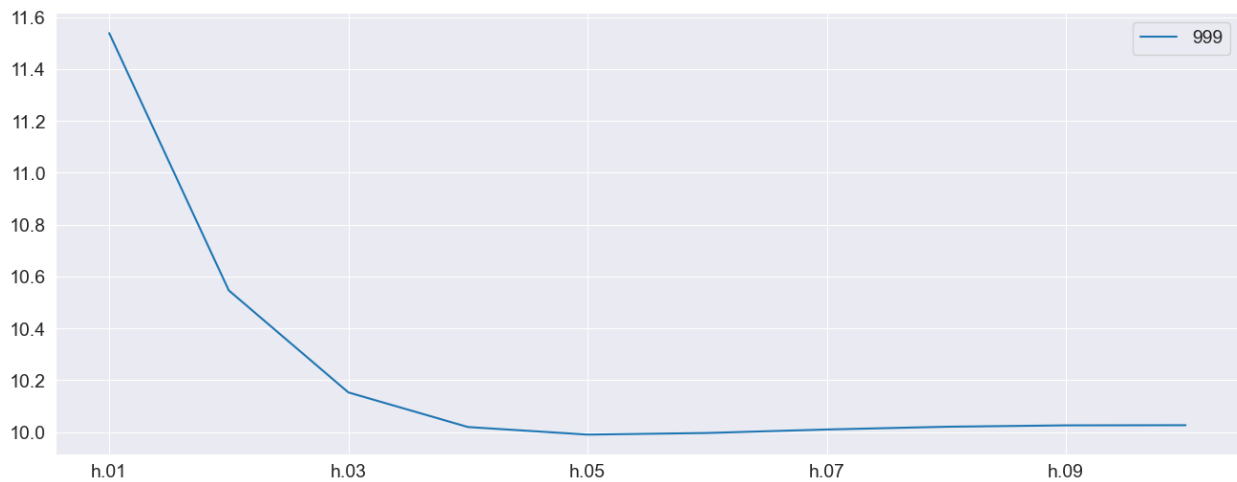
The first approach uses a dict to pass the two variables. The key consideration here is the the keys of the dictionary must **exactly** match the variable names (`x0` and `x1` here). The dictionary here contains only the final row of the forecast values since `forecast` will only make forecasts beginning from the final in-sample observation by default.

Using DataFrame

While these examples make use of NumPy arrays, these can be DataFrames. This allows the index to be used to track the forecast origination point, which can be a helpful device.

[6]:
```
exog_fcast = {"x0": x0_oos[-1:], "x1": x1_oos[-1:]}
forecasts = res.forecast(horizon=10, x=exog_fcast)
forecasts.mean.T.plot()
```

[6]:
```
<Axes: >
```



1.7.6 Using an array

An array can alternatively be used. This frees the restriction on matching the variable names although the order must match instead. The forecast values are 2 (variables) by 1 (forecast) by 10 (horizon).

[7]:
```
exog_fcast = np.array([x0_oos[-1:], x1_oos[-1:]])
print(f"The shape is {exog_fcast.shape}")
array_forecasts = res.forecast(horizon=10, x=exog_fcast)
print(array_forecasts.mean - forecasts.mean)
```

```
The shape is (2, 1, 10)
     h.01  h.02  h.03  h.04  h.05  h.06  h.07  h.08  h.09  h.10
999   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
```

1.7.7 Producing multiple forecasts

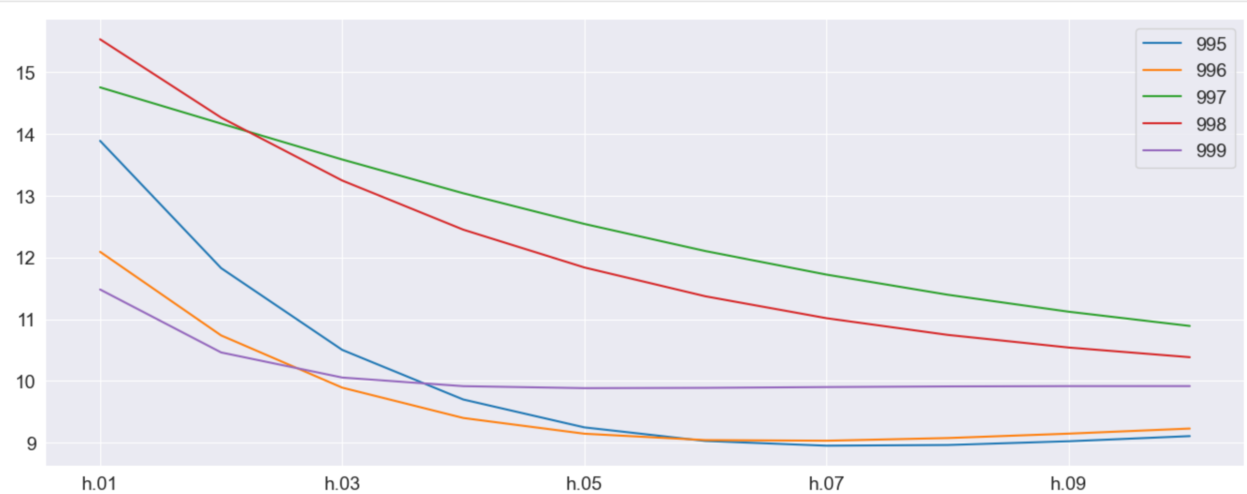
forecast can produce multiple forecasts using the same fit model. Here the model is fit to the first 500 observations and then forecasting for the remaining values are produced. It must be the case that the x values passed for forecast have the same number of rows as the table of forecasts produced.

[8]:
```
res = mod.fit(disp="off", last_obs=500)
exog_fcast = {"x0": x0_oos[-500:], "x1": x1_oos[-500:]}
multi_forecasts = res.forecast(start=500, horizon=10, x=exog_fcast)
multi_forecasts.mean.tail(10)
```

[8]:
```
          h.01       h.02       h.03       h.04       h.05       h.06  \
990  17.638678  19.057943  19.305712  18.876070  18.090794  17.153279
991  20.307394  20.095492  19.176459  18.008276  16.816890  15.707369
992  18.286380  16.896235  15.586099  14.454254  13.517502  12.759810
993  24.177049  22.846561  20.903832  18.944579  17.198290  15.728217
994  21.476395  19.945167  18.325031  16.830889  15.536242  14.450638
995  13.887372  11.827018  10.504285   9.701208   9.250338   9.029847
996  12.091224  10.738291   9.893515   9.402421   9.147641   9.044580
997  14.752402  14.165799  13.584948  13.038435  12.542245  12.103274
998  15.530046  14.262848  13.244627  12.449407  11.837882  11.371370
999  11.481761  10.462828  10.056773   9.917344   9.885242   9.890079

          h.07       h.08       h.09       h.10
990  16.187233  15.263806  14.420245  13.672511
991  14.722133  13.871672  13.150830  12.547444
992  12.154606  11.674450  11.294780  10.994971
993  14.527010  13.561360  12.791967  12.181810
994  13.557059  12.829543  12.241130  11.767149
995   8.954356   8.965773   9.025626   9.109140
996   9.034509   9.077894   9.148903   9.231197
997  11.722401  11.396828  11.121754  10.891488
998  11.016716  10.747264  10.542302  10.386034
999   9.902827   9.913039   9.918147   9.918574
```

The plot of the final 5 forecast paths shows the the mean reversion of the process.

[9]:
```
_ = multi_forecasts.mean.tail().T.plot()
```



The previous example made use of dictionaries where each of the values was a 500 (number of forecasts) by 10 (horizon) array. The alternative format can be used where x is a 3-d array with shape 2 (variables) by 500 (forecasts) by 10 (horizon).

[10]:
```
exog_fcast = np.array([x0_oos[-500:], x1_oos[-500:]])
print(exog_fcast.shape)
array_multi_forecasts = res.forecast(start=500, horizon=10, x=exog_fcast)
np.max(np.abs(array_multi_forecasts.mean - multi_forecasts.mean))
```

```
(2, 500, 10)
```

[10]:
```
0.0
```

1.7.8 x input array sizes

While the natural shape of the x data is the number of forecasts, it is also possible to pass an x that has the same shape as the y used to construct the model. The may simplify tracking the origin points of the forecast. Values are are not needed are ignored. In this example, the out-of-sample values are 2 by 1000 (original number of observations) by 10. Only the final 500 are used.

> WARNING
>
> Other sizes are not allowed. The size of the out-of-sample data must either match the original data size or the number of forecasts.

[11]:
```
exog_fcast = np.array([x0_oos, x1_oos])
print(exog_fcast.shape)
array_multi_forecasts = res.forecast(start=500, horizon=10, x=exog_fcast)
np.max(np.abs(array_multi_forecasts.mean - multi_forecasts.mean))
```

```
(2, 1000, 10)
```

[11]:
```
0.0
```

1.7.9 Special Cases with a single x variable

When a model consists of a single exogenous regressor, then x can be a 1-d or 2-d array (or Series or DataFrame).

[12]:
```
mod = arch_model(y, x=exog.iloc[:, :1], mean="ARX", lags=1)
res = mod.fit(disp="off")
print(res.summary())
```

```
                           AR-X - GARCH Model Results
==============================================================================
Dep. Variable:                   data   R-squared:                       0.955
Mean Model:                      AR-X   Adj. R-squared:                  0.955
Vol Model:                      GARCH   Log-Likelihood:               -2280.96
Distribution:                  Normal   AIC:                           4573.92
Method:            Maximum Likelihood   BIC:                           4603.36
                                        No. Observations:                  999
Date:                Tue, May 30 2023   Df Residuals:                      996
Time:                        10:55:42   Df Model:                            3
                                Mean Model
========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
Const         -6.4609      0.226    -28.561 2.064e-179 [ -6.904, -6.018]
data[1]        0.7695  8.995e-03     85.552      0.000 [  0.752,  0.787]
x0             1.7394  5.329e-02     32.643 1.021e-233 [  1.635,  1.844]
```

(continues on next page)

(continued from previous page)

```
                          Volatility Model
==============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
------------------------------------------------------------------------------
omega          0.1073  7.460e-02      1.438      0.150  [-3.895e-02,  0.253]
alpha[1]   8.6689e-03  1.513e-02      0.573      0.567 [-2.099e-02,3.832e-02]
beta[1]        0.9717  2.616e-02     37.149 4.549e-302      [  0.920,  1.023]
==============================================================================

Covariance estimator: robust
```

These two examples show that both formats can be used.

[13]:
```
forecast_1d = res.forecast(horizon=10, x=x0_oos[-1])
forecast_2d = res.forecast(horizon=10, x=x0_oos[-1:])
print(forecast_1d.mean - forecast_2d.mean)

## Simulation-forecasting

mod = arch_model(y, x=exog, mean="ARX", lags=1, power=1.0)
res = mod.fit(disp="off")
```

```
     h.01  h.02  h.03  h.04  h.05  h.06  h.07  h.08  h.09  h.10
999   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
```

1.7.10 Simulation

`forecast` supports simulating paths. When forecasting a model with exogenous variables, the same value is used to in all mean paths. If you wish to also simulate the paths of the `x` variables, these need to generated and then passed inside a loop.

Static out-of-sample x

This first example shows that variance of the paths when the same `x` values are used in the forecast. There is a sense the out-of-sample `x` are treated as deterministic.

[14]:
```
x = {"x0": x0_oos[-1], "x1": x1_oos[-1]}
sim_fixedx = res.forecast(horizon=10, x=x, method="simulation", simulations=100)
sim_fixedx.simulations.values.std(1)
```

[14]:
```
array([[1.04095499, 1.27798546, 1.30149937, 1.53498654, 1.60825322,
        1.53831759, 1.55948399, 1.49811376, 1.74794163, 1.54496175]])
```

Simulating the out-of-sample x

This example simulates distinct paths for the two exogenous variables and then simulates a single path. This is then repeated 100 times. We see that variance is much higher when we account for variation in the x data.

[15]:
```
from numpy.random import RandomState


def sim_ar1(params: np.ndarray, initial: float, horizon: int, rng: RandomState):
    out = np.zeros(horizon)
    shocks = rng.standard_normal(horizon)
    out[0] = params[0] + params[1] * initial + shocks[0]
    for i in range(1, horizon):
        out[i] = params[0] + params[1] * out[i - 1] + shocks[i]
    return out


simulations = []
rng = RandomState(20210301)
for i in range(100):
    x0_sim = sim_ar1(np.array([1, 0.8]), x0.iloc[-1], 10, rng)
    x1_sim = sim_ar1(np.array([2.5, 0.5]), x1.iloc[-1], 10, rng)
    x = {"x0": x0_sim, "x1": x1_sim}
    fcast = res.forecast(horizon=10, x=x, method="simulation", simulations=1)
    simulations.append(fcast.simulations.values)
```

Finally the standard deviation is quite a bit larger. This is a most accurate value fo the long-run variance of the forecast residuals which should account for dynamics in the model and any exogenous regressors.

[16]:
```
joined = np.concatenate(simulations, 1)
joined.std(1)
```

[16]:
```
array([[2.95980851, 4.9043695 , 6.49587772, 7.50716786, 8.27211937,
        8.71428305, 8.71321666, 8.67711577, 8.80764753, 9.17554847]])
```

1.8 Mean Models

All ARCH models start by specifying a mean model.

ZeroMean([y, hold_back, volatility, ...])	Model with zero conditional mean estimation and simulation
ConstantMean([y, hold_back, volatility, ...])	Constant mean model estimation and simulation.
ARX([y, x, lags, constant, hold_back, ...])	Autoregressive model with optional exogenous regressors estimation and simulation
HARX([y, x, lags, constant, use_rotated, ...])	Heterogeneous Autoregression (HAR), with optional exogenous regressors, model estimation and simulation
LS([y, x, constant, hold_back, volatility, ...])	Least squares model estimation and simulation

1.8.1 arch.univariate.ZeroMean

class arch.univariate.**ZeroMean**(*y: ndarray | DataFrame | Series | None = **None**, hold_back: int | None = **None**, volatility: VolatilityProcess | None = **None**, distribution: Distribution | None = **None**, rescale: bool | None = **None***)

Model with zero conditional mean estimation and simulation

Parameters

y: ndarray | DataFrame | Series | None = None
nobs element vector containing the dependent variable

hold_back: int | None = None
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.

volatility: *VolatilityProcess* | None = None
Volatility process to use in the model

distribution: *Distribution* | None = None
Error distribution to use in the model

rescale: bool | None = None
Flag indicating whether to automatically rescale data if the scale of the data is likely to produce convergence issues when estimating model parameters. If False, the model is estimated on the data without transformation. If True, than y is rescaled and the new scale is reported in the estimation results.

Examples

```
>>> import numpy as np
>>> from arch.univariate import ZeroMean
>>> y = np.random.randn(100)
>>> zm = ZeroMean(y)
>>> res = zm.fit()
```

Notes

The zero mean model is described by

$$y_t = \epsilon_t$$

Methods

bounds()	Construct bounds for parameters to use in non-linear optimization
compute_param_cov(params[, backcast, robust])	Computes parameter covariances using numerical derivatives.
constraints()	Construct linear constraint arrays for use in non-linear optimization
fit([update_freq, disp, starting_values, ...])	Estimate model parameters
fix(params[, first_obs, last_obs])	Allows an ARCHModelFixedResult to be constructed from fixed parameters.
forecast(params[, horizon, start, align, ...])	Construct forecasts from estimated model
parameter_names()	List of parameters names
resids(params[, y, regressors])	Compute model residuals
simulate(params, nobs[, burn, ...])	Simulated data from a zero mean model
starting_values()	Returns starting values for the mean model, often the same as the values returned from fit

arch.univariate.ZeroMean.bounds

ZeroMean.**bounds**() → list[tuple[float, float]]

Construct bounds for parameters to use in non-linear optimization

Returns
bounds – Bounds for parameters to use in estimation.

Return type
`list (2-tuple` of `float)`

arch.univariate.ZeroMean.compute_param_cov

ZeroMean.**compute_param_cov**(params: *ndarray*, backcast: *None* | *float* | *ndarray* = ***None***, robust: *bool* = ***True***) → ndarray

Computes parameter covariances using numerical derivatives.

Parameters

params: ndarray
Model parameters

backcast: None | float | ndarray = None
Value to use for pre-sample observations

robust: bool = True
Flag indicating whether to use robust standard errors (True) or classic MLE (False)

arch.univariate.ZeroMean.constraints

`ZeroMean.constraints()` → tuple[ndarray, ndarray]

Construct linear constraint arrays for use in non-linear optimization

Returns

- **a** (`numpy.ndarray`) – Number of constraints by number of parameters loading array
- **b** (`numpy.ndarray`) – Number of constraints array of lower bounds

Notes

Parameters satisfy a.dot(parameters) - b >= 0

arch.univariate.ZeroMean.fit

`ZeroMean.fit`(update_freq: *int* = *1*, disp: *bool* | *'off'* | *'final'* = *'final'*, starting_values: *ndarray* | *Series* | *None* = ***None***, cov_type: *'robust'* | *'classic'* = *'robust'*, show_warning: *bool* = ***True***, first_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = ***None***, last_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = ***None***, tol: *float* | *None* = ***None***, options: *dict[str, Any]* | *None* = ***None***, backcast: *None* | *float* | *ndarray* = ***None***) → *ARCHModelResult*

Estimate model parameters

Parameters

update_freq: int = 1
Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.

disp: bool | 'off' | 'final' = 'final'
Either 'final' to print optimization result or 'off' to display nothing. If using a boolean, False is "off" and True is "final"

starting_values: ndarray | Series | None = None
Array of starting values to use. If not provided, starting values are constructed by the model components.

cov_type: 'robust' | 'classic' = 'robust'
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 = True
Flag indicating whether convergence warnings should be shown.

first_obs: int | str | datetime | datetime64 | Timestamp | None = None
First observation to use when estimating model

last_obs: int | str | datetime | datetime64 | Timestamp | None = None
Last observation to use when estimating model

tol: float | None = None
Tolerance for termination.

options: dict[str, Any] | None = None
Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.

backcast: None | float | ndarray = None
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
results – Object containing model results

Return type
`ARCHModelResult`

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

arch.univariate.ZeroMean.fix

`ZeroMean.fix`(params: *Sequence[float]* | *ndarray* | *Series*, first_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = `None`, last_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = `None`) → *ARCHModelFixedResult*

Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

params: Sequence[float] | ndarray | Series
User specified parameters to use when generating the result. Must have the correct number of parameters for a given choice of mean model, volatility model and distribution.

first_obs: int | str | datetime | datetime64 | Timestamp | None = None
First observation to use when fixing model

last_obs: int | str | datetime | datetime64 | Timestamp | None = None
Last observation to use when fixing model

Returns
results – Object containing model results

Return type
`ARCHModelFixedResult`

Notes

Parameters are not checked against model-specific constraints.

arch.univariate.ZeroMean.forecast

`ZeroMean.forecast`(params: *ArrayLike1D*, horizon: *int* = `1`, start: *None* | *int* | *DateLike* = `None`, align: *'origin'* | *'target'* = `'origin'`, method: *ForecastingMethod* = `'analytic'`, simulations: *int* = `1000`, rng: *collections.abc.Callable[[int* | *tuple[int, ...]], Float64Array]* | *None* = `None`, random_state: *np.random.RandomState* | *None* = `None`, *, reindex: *bool* = `False`, x: *None* | *dict[Label, ArrayLike]* | *ArrayLike* = `None`) → *ARCHModelForecast*

Construct forecasts from estimated model

Parameters

params: ArrayLike1D
Parameters required to forecast. Must be identical in shape to the parameters computed by fitting the model.

horizon: int = 1
Number of steps to forecast

start: None | int | DateLike = None
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' = 'origin'
Either 'origin' or 'target'. When set of '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' simplified computing forecast errors since the realization and h-step forecast are aligned.

method: ForecastingMethod = 'analytic'
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 = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None = None
Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

random_state: np.random.RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

reindex: bool = False

Whether to reindex the forecasts to have the same dimension as the series being forecast. Prior to 4.18 this was the default. As of 4.19 this is now optional. If not provided, a warning is raised about the future change in the default which will occur after September 2021.

New in version 4.19.

x: None | dict[Label, ArrayLike] | ArrayLike = None

Values to use for exogenous regressors if any are included in the model. Three formats are accepted:

- 2-d array-like: This format can be used when there is a single exogenous variable. The input must have shape (nforecast, horizon) or (nobs, horizon) where nforecast is the number of forecasting periods and nobs is the original shape of y. For example, if a single series of forecasts are made from the end of the sample with a horizon of 10, then the input can be (1, 10). Alternatively, if the original data had 1000 observations, then the input can be (1000, 10), and only the final row is used to produce forecasts.
- A dictionary of 2-d array-like: This format is identical to the previous except that the dictionary keys must match the names of the exog variables. Requires that the exog variables were passed as a pandas DataFrame.

- A 3-d NumPy array (or equivalent). In this format, each panel (0th axis) is a 2-d array that must have shape (nforecast, horizon) or (nobs,horizon). The array x[j] corresponds to the j-th column of the exogenous variables.

Due to the complexity required to accommodate all scenarios, please see the example notebook that demonstrates the valid formats for x.

New in version 4.19.

Returns
Container for forecasts. Key properties are `mean`, `variance` and `residual_variance`.

Return type
arch.univariate.base.ARCHModelForecast

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],  vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot()
```

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.

arch.univariate.ZeroMean.parameter_names

`ZeroMean.`**`parameter_names`**`()` → list[str]

List of parameters names

Returns
names – List of variable names for the mean model

Return type
`list (str)`

arch.univariate.ZeroMean.resids

`ZeroMean.resids`(params: *ndarray*, y: *ndarray* | *Series* | *None* = **None**, regressors: *ndarray* | *DataFrame* | *None* = **None**) → ndarray | Series

Compute model residuals

Parameters

params: ndarray
Model parameters

y: ndarray | Series | None = None
Alternative values to use when computing model residuals

regressors: ndarray | DataFrame | None = None
Alternative regressor values to use when computing model residuals

Returns
resids – Model residuals

Return type
`numpy.ndarray`

arch.univariate.ZeroMean.simulate

`ZeroMean.simulate`(params: *ArrayLike1D* | *collections.abc.Sequence[float]*, nobs: *int*, burn: *int* = **500**, initial_value: *None* | *float* | *Float64Array* = **None**, x: *ArrayLike* | *None* = **None**, initial_value_vol: *None* | *float* | *Float64Array* = **None**) → pd.DataFrame

Simulated data from a zero mean model

Parameters

params: ArrayLike1D | collections.abc.Sequence[float]
Parameters to use when simulating the model. Parameter order is [volatility distribution]. There are no mean parameters.

nobs: int
Length of series to simulate

burn: int = 500
Number of values to simulate to initialize the model and remove dependence on initial values.

initial_value: None | float | Float64Array = None
This value is not used.

x: ArrayLike | None = None
This value is not used.

initial_value_vol: None | float | Float64Array = None
An array or scalar to use when initializing the volatility process.

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

Return type
`pandas.DataFrame`

Examples

Basic data simulation with no mean and constant volatility

```
>>> from arch.univariate import ZeroMean
>>> import numpy as np
>>> zm = ZeroMean()
>>> params = np.array([1.0])
>>> sim_data = zm.simulate(params, 1000)
```

Simulating data with a non-trivial volatility process

```
>>> from arch.univariate import GARCH
>>> zm.volatility = GARCH(p=1, o=1, q=1)
>>> sim_data = zm.simulate([0.05, 0.1, 0.1, 0.8], 300)
```

arch.univariate.ZeroMean.starting_values

ZeroMean.**starting_values**() → ndarray

Returns starting values for the mean model, often the same as the values returned from fit

Returns
sv – Starting values

Return type
numpy.ndarray

Properties

distribution	Set or gets the error distribution
name	The name of the model.
num_params	Returns the number of parameters
volatility	Set or gets the volatility process
x	Gets the value of the exogenous regressors in the model
y	Returns the dependent variable

arch.univariate.ZeroMean.distribution

property ZeroMean.**distribution** : *Distribution*

Set or gets the error distribution

Distributions must be a subclass of Distribution

arch.univariate.ZeroMean.name

property ZeroMean.**name** : str

The name of the model.

arch.univariate.ZeroMean.num_params

property ZeroMean.**num_params** : int

Returns the number of parameters

arch.univariate.ZeroMean.volatility

property ZeroMean.**volatility** : *VolatilityProcess*

Set or gets the volatility process

Volatility processes must be a subclass of VolatilityProcess

arch.univariate.ZeroMean.x

property ZeroMean.**x** : ndarray | DataFrame | None

Gets the value of the exogenous regressors in the model

arch.univariate.ZeroMean.y

property ZeroMean.**y** : ndarray | DataFrame | Series | None

Returns the dependent variable

1.8.2 arch.univariate.ConstantMean

class arch.univariate.**ConstantMean**(y: *ndarray* | *DataFrame* | *Series* | *None* = ***None***, hold_back: *int* | *None* = ***None***, volatility: VolatilityProcess | *None* = ***None***, distribution: Distribution | *None* = ***None***, rescale: *bool* | *None* = ***None***)

Constant mean model estimation and simulation.

Parameters

y: ndarray | DataFrame | Series | None = None
nobs element vector containing the dependent variable

hold_back: int | None = None
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.

volatility: *VolatilityProcess* | None = None
Volatility process to use in the model

distribution: *Distribution* | None = None
Error distribution to use in the model

rescale: bool | None = None
Flag indicating whether to automatically rescale data if the scale of the data is likely to produce convergence issues when estimating model parameters. If False, the model is estimated on the data without transformation. If True, than y is rescaled and the new scale is reported in the estimation results.

Examples

```
>>> import numpy as np
>>> from arch.univariate import ConstantMean
>>> y = np.random.randn(100)
>>> cm = ConstantMean(y)
>>> res = cm.fit()
```

Notes

The constant mean model is described by

$$y_t = \mu + \epsilon_t$$

Methods

bounds()	Construct bounds for parameters to use in non-linear optimization
compute_param_cov(params[, backcast, robust])	Computes parameter covariances using numerical derivatives.
constraints()	Construct linear constraint arrays for use in non-linear optimization
fit([update_freq, disp, starting_values, ...])	Estimate model parameters
fix(params[, first_obs, last_obs])	Allows an ARCHModelFixedResult to be constructed from fixed parameters.
forecast(params[, horizon, start, align, ...])	Construct forecasts from estimated model
parameter_names()	List of parameters names
resids(params[, y, regressors])	Compute model residuals
simulate(params, nobs[, burn, ...])	Simulated data from a constant mean model
starting_values()	Returns starting values for the mean model, often the same as the values returned from fit

arch.univariate.ConstantMean.bounds

`ConstantMean.bounds()` → list[tuple[float, float]]

Construct bounds for parameters to use in non-linear optimization

Returns
bounds – Bounds for parameters to use in estimation.

Return type
`list (2-tuple of float)`

arch.univariate.ConstantMean.compute_param_cov

ConstantMean.**compute_param_cov**(params: *ndarray*, backcast: *None | float | ndarray* = **None**, robust: *bool* = **True**) → ndarray

Computes parameter covariances using numerical derivatives.

Parameters

params: ndarray
Model parameters

backcast: None | float | ndarray = None
Value to use for pre-sample observations

robust: bool = True
Flag indicating whether to use robust standard errors (True) or classic MLE (False)

arch.univariate.ConstantMean.constraints

ConstantMean.**constraints**() → tuple[ndarray, ndarray]

Construct linear constraint arrays for use in non-linear optimization

Returns

- **a** (`numpy.ndarray`) – Number of constraints by number of parameters loading array
- **b** (`numpy.ndarray`) – Number of constraints array of lower bounds

Notes

Parameters satisfy a.dot(parameters) - b >= 0

arch.univariate.ConstantMean.fit

ConstantMean.**fit**(update_freq: *int* = **1**, disp: *bool | 'off' | 'final'* = `'final'`, starting_values: *ndarray | Series | None* = **None**, cov_type: *'robust' | 'classic'* = `'robust'`, show_warning: *bool* = **True**, first_obs: *int | str | datetime | datetime64 | Timestamp | None* = **None**, last_obs: *int | str | datetime | datetime64 | Timestamp | None* = **None**, tol: *float | None* = **None**, options: *dict[str, Any] | None* = **None**, backcast: *None | float | ndarray* = **None**) → *ARCHModelResult*

Estimate model parameters

Parameters

update_freq: int = 1
Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.

disp: bool | 'off' | 'final' = 'final'
Either 'final' to print optimization result or 'off' to display nothing. If using a boolean, False is "off" and True is "final"

starting_values: ndarray | Series | None = None
Array of starting values to use. If not provided, starting values are constructed by the model components.

cov_type: 'robust' | 'classic' = `'robust'`
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 = `True`
Flag indicating whether convergence warnings should be shown.

first_obs: int | str | datetime | datetime64 | Timestamp | None = `None`
First observation to use when estimating model

last_obs: int | str | datetime | datetime64 | Timestamp | None = `None`
Last observation to use when estimating model

tol: float | None = `None`
Tolerance for termination.

options: dict[str, Any] | None = `None`
Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.

backcast: None | float | ndarray = `None`
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
results – Object containing model results

Return type
`ARCHModelResult`

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

arch.univariate.ConstantMean.fix

`ConstantMean.fix`(params*: Sequence[float] | ndarray | Series*, first_obs*: int | str | datetime | datetime64 | Timestamp | None* = `None`, last_obs*: int | str | datetime | datetime64 | Timestamp | None* = `None`) → *ARCHModelFixedResult*

Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

params: Sequence[float] | ndarray | Series
User specified parameters to use when generating the result. Must have the correct number of parameters for a given choice of mean model, volatility model and distribution.

first_obs: int | str | datetime | datetime64 | Timestamp | None = `None`
First observation to use when fixing model

last_obs: int | str | datetime | datetime64 | Timestamp | None = `None`
Last observation to use when fixing model

Returns
results – Object containing model results

Return type
`ARCHModelFixedResult`

Notes

Parameters are not checked against model-specific constraints.

arch.univariate.ConstantMean.forecast

`ConstantMean.forecast`(params: *ArrayLike1D*, horizon: *int* = 1, start: *None* | *int* | *DateLike* = None, align: *'origin'* | *'target'* = 'origin', method: *ForecastingMethod* = 'analytic', simulations: *int* = 1000, rng: *collections.abc.Callable[[int* | *tuple[int, ...]], Float64Array]* | *None* = None, random_state: *np.random.RandomState* | *None* = None, *, reindex: *bool* = False, x: *None* | *dict[Label, ArrayLike]* | *ArrayLike* = None) → *ARCHModelForecast*

Construct forecasts from estimated model

Parameters

params: ArrayLike1D
Parameters required to forecast. Must be identical in shape to the parameters computed by fitting the model.

horizon: int = 1
Number of steps to forecast

start: None | int | DateLike = None
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' = 'origin'
Either 'origin' or 'target'. When set of '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' simplified computing forecast errors since the realization and h-step forecast are aligned.

method: ForecastingMethod = 'analytic'
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 = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None = None
Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

random_state: np.random.RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

reindex: bool = False

Whether to reindex the forecasts to have the same dimension as the series being forecast. Prior to 4.18 this was the default. As of 4.19 this is now optional. If not provided, a warning is raised about the future change in the default which will occur after September 2021.

New in version 4.19.

x: None | dict[Label, ArrayLike] | ArrayLike = None

Values to use for exogenous regressors if any are included in the model. Three formats are accepted:

- 2-d array-like: This format can be used when there is a single exogenous variable. The input must have shape (nforecast, horizon) or (nobs, horizon) where nforecast is the number of forecasting periods and nobs is the original shape of y. For example, if a single series of forecasts are made from the end of the sample with a horizon of 10, then the input can be (1, 10). Alternatively, if the original data had 1000 observations, then the input can be (1000, 10), and only the final row is used to produce forecasts.
- A dictionary of 2-d array-like: This format is identical to the previous except that the dictionary keys must match the names of the exog variables. Requires that the exog variables were passed as a pandas DataFrame.
- A 3-d NumPy array (or equivalent). In this format, each panel (0th axis) is a 2-d array that must have shape (nforecast, horizon) or (nobs,horizon). The array x[j] corresponds to the j-th column of the exogenous variables.

Due to the complexity required to accommodate all scenarios, please see the example notebook that demonstrates the valid formats for x.

New in version 4.19.

Returns

Container for forecasts. Key properties are `mean`, `variance` and `residual_variance`.

Return type

arch.univariate.base.ARCHModelForecast

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],  vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot()
```

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.

arch.univariate.ConstantMean.parameter_names

`ConstantMean.parameter_names()` → list[str]

List of parameters names

Returns
names – List of variable names for the mean model

Return type
`list (str)`

arch.univariate.ConstantMean.resids

`ConstantMean.resids(params: ndarray, y: ndarray | Series | None = None, regressors: ndarray | DataFrame | None = None)` → ndarray | Series

Compute model residuals

Parameters

params: ndarray
Model parameters

y: ndarray | Series | None = None
Alternative values to use when computing model residuals

regressors: ndarray | DataFrame | None = None
Alternative regressor values to use when computing model residuals

Returns
resids – Model residuals

Return type
`numpy.ndarray`

arch.univariate.ConstantMean.simulate

ConstantMean.**simulate**(params: *ArrayLike1D | collections.abc.Sequence[float]*, nobs: *int*, burn: *int* = 500, initial_value: *None | float | Float64Array* = None, x: *ArrayLike | None* = None, initial_value_vol: *None | float | Float64Array* = None) → pd.DataFrame

Simulated data from a constant mean model

Parameters

params: ArrayLike1D | collections.abc.Sequence[float]
Parameters to use when simulating the model. Parameter order is [mean volatility distribution]. There is one parameter in the mean model, mu.

nobs: int
Length of series to simulate

burn: int = 500
Number of values to simulate to initialize the model and remove dependence on initial values.

initial_value: None | float | Float64Array = None
This value is not used.

x: ArrayLike | None = None
This value is not used.

initial_value_vol: None | float | Float64Array = None
An array or scalar to use when initializing the volatility process.

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

Return type
`pandas.DataFrame`

Examples

Basic data simulation with a constant mean and volatility

```
>>> import numpy as np
>>> from arch.univariate import ConstantMean, GARCH
>>> cm = ConstantMean()
>>> cm.volatility = GARCH()
>>> cm_params = np.array([1])
>>> garch_params = np.array([0.01, 0.07, 0.92])
>>> params = np.concatenate((cm_params, garch_params))
>>> sim_data = cm.simulate(params, 1000)
```

arch.univariate.ConstantMean.starting_values

`ConstantMean.starting_values()` → ndarray

Returns starting values for the mean model, often the same as the values returned from fit

Returns
sv – Starting values

Return type
`numpy.ndarray`

Properties

distribution	Set or gets the error distribution
name	The name of the model.
num_params	Returns the number of parameters
volatility	Set or gets the volatility process
x	Gets the value of the exogenous regressors in the model
y	Returns the dependent variable

arch.univariate.ConstantMean.distribution

property `ConstantMean.distribution` : *Distribution*

Set or gets the error distribution

Distributions must be a subclass of Distribution

arch.univariate.ConstantMean.name

property `ConstantMean.name` : str

The name of the model.

arch.univariate.ConstantMean.num_params

property `ConstantMean.num_params` : int

Returns the number of parameters

arch.univariate.ConstantMean.volatility

property `ConstantMean.volatility` : *VolatilityProcess*

Set or gets the volatility process

Volatility processes must be a subclass of VolatilityProcess

arch.univariate.ConstantMean.x

property `ConstantMean.x` : ndarray | DataFrame | None

Gets the value of the exogenous regressors in the model

arch.univariate.ConstantMean.y

property `ConstantMean.y` : ndarray | DataFrame | Series | None

Returns the dependent variable

1.8.3 arch.univariate.ARX

class `arch.univariate.ARX`(y: *ndarray* | *DataFrame* | *Series* | *None* = **None**, x: *ndarray* | *DataFrame* | *None* = **None**, lags: *None* | *int* | *list[int]* | *ndarray* = **None**, constant: *bool* = **True**, hold_back: *int* | *None* = **None**, volatility: VolatilityProcess | *None* = **None**, distribution: Distribution | *None* = **None**, rescale: *bool* | *None* = **None**)

Autoregressive model with optional exogenous regressors estimation and simulation

Parameters

y: ndarray | DataFrame | Series | None = None
nobs element vector containing the dependent variable

x: ndarray | DataFrame | None = None
nobs by k element array containing exogenous regressors

lags: None | int | list[int] | ndarray = None
Description of lag structure of the HAR. Scalar included all lags between 1 and the value. A 1-d array includes the AR lags lags[0], lags[1], …

constant: bool = True
Flag whether the model should include a constant

hold_back: int | None = None
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.

volatility: *VolatilityProcess* | None = None
Volatility process to use in the model

distribution: *Distribution* | None = None
Error distribution to use in the model

rescale: bool | None = None
Flag indicating whether to automatically rescale data if the scale of the data is likely to produce convergence issues when estimating model parameters. If False, the model is estimated on the data without transformation. If True, than y is rescaled and the new scale is reported in the estimation results.

Examples

```
>>> import numpy as np
>>> from arch.univariate import ARX
>>> y = np.random.randn(100)
>>> arx = ARX(y, lags=[1, 5, 22])
>>> res = arx.fit()
```

Estimating an AR with GARCH(1,1) errors

```
>>> from arch.univariate import GARCH
>>> arx.volatility = GARCH()
>>> res = arx.fit(update_freq=0, disp='off')
```

Notes

The AR-X model is described by

$$y_t = \mu + \sum_{i=1}^{p} \phi_{L_i} y_{t-L_i} + \gamma' x_t + \epsilon_t$$

Methods

bounds()	Construct bounds for parameters to use in non-linear optimization
compute_param_cov(params[, backcast, robust])	Computes parameter covariances using numerical derivatives.
constraints()	Construct linear constraint arrays for use in non-linear optimization
fit([update_freq, disp, starting_values, ...])	Estimate model parameters
fix(params[, first_obs, last_obs])	Allows an ARCHModelFixedResult to be constructed from fixed parameters.
forecast(params[, horizon, start, align, ...])	Construct forecasts from estimated model
parameter_names()	List of parameters names
resids(params[, y, regressors])	Compute model residuals
simulate(params, nobs[, burn, ...])	Simulates data from a linear regression, AR or HAR models
starting_values()	Returns starting values for the mean model, often the same as the values returned from fit

arch.univariate.ARX.bounds

`ARX.`**bounds**`()` → list[tuple[float, float]]

Construct bounds for parameters to use in non-linear optimization

Returns
bounds – Bounds for parameters to use in estimation.

Return type
`list (2-tuple of float)`

arch.univariate.ARX.compute_param_cov

`ARX.compute_param_cov(params: ndarray, backcast: None | float | ndarray = None, robust: bool = True) → ndarray`

Computes parameter covariances using numerical derivatives.

Parameters

params: ndarray
Model parameters

backcast: None | float | ndarray = None
Value to use for pre-sample observations

robust: bool = True
Flag indicating whether to use robust standard errors (True) or classic MLE (False)

arch.univariate.ARX.constraints

`ARX.constraints() → tuple[ndarray, ndarray]`

Construct linear constraint arrays for use in non-linear optimization

Returns

- **a** (numpy.ndarray) – Number of constraints by number of parameters loading array
- **b** (numpy.ndarray) – Number of constraints array of lower bounds

Notes

Parameters satisfy a.dot(parameters) - b >= 0

arch.univariate.ARX.fit

`ARX.fit(update_freq: int = 1, disp: bool | 'off' | 'final' = 'final', starting_values: ndarray | Series | None = None, cov_type: 'robust' | 'classic' = 'robust', show_warning: bool = True, first_obs: int | str | datetime | datetime64 | Timestamp | None = None, last_obs: int | str | datetime | datetime64 | Timestamp | None = None, tol: float | None = None, options: dict[str, Any] | None = None, backcast: None | float | ndarray = None) → ARCHModelResult`

Estimate model parameters

Parameters

update_freq: int = 1
Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.

disp: bool | 'off' | 'final' = 'final'
Either 'final' to print optimization result or 'off' to display nothing. If using a boolean, False is "off" and True is "final"

starting_values: ndarray | Series | None = None
Array of starting values to use. If not provided, starting values are constructed by the model components.

cov_type: 'robust' | 'classic' = `'robust'`
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 = `True`
Flag indicating whether convergence warnings should be shown.

first_obs: int | str | datetime | datetime64 | Timestamp | None = `None`
First observation to use when estimating model

last_obs: int | str | datetime | datetime64 | Timestamp | None = `None`
Last observation to use when estimating model

tol: float | None = `None`
Tolerance for termination.

options: dict[str, Any] | None = `None`
Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.

backcast: None | float | ndarray = `None`
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
results – Object containing model results

Return type
`ARCHModelResult`

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

arch.univariate.ARX.fix

`ARX.fix`(params*: Sequence[float] | ndarray | Series*, first_obs*: int | str | datetime | datetime64 | Timestamp | None =* **None**, last_obs*: int | str | datetime | datetime64 | Timestamp | None =* **None**) → *ARCHModelFixedResult*

Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

params: Sequence[float] | ndarray | Series
User specified parameters to use when generating the result. Must have the correct number of parameters for a given choice of mean model, volatility model and distribution.

first_obs: int | str | datetime | datetime64 | Timestamp | None = `None`
First observation to use when fixing model

last_obs: int | str | datetime | datetime64 | Timestamp | None = `None`
Last observation to use when fixing model

Returns
results – Object containing model results

Return type
ARCHModelFixedResult

Notes

Parameters are not checked against model-specific constraints.

arch.univariate.ARX.forecast

`ARX.forecast(params: ArrayLike1D, horizon: int = 1, start: None | int | DateLike = None, align: 'origin' | 'target' = 'origin', method: ForecastingMethod = 'analytic', simulations: int = 1000, rng: collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None = None, random_state: np.random.RandomState | None = None, *, reindex: bool = False, x: None | dict[Label, ArrayLike] | ArrayLike = None) → ARCHModelForecast`

Construct forecasts from estimated model

Parameters

params: ArrayLike1D
Parameters required to forecast. Must be identical in shape to the parameters computed by fitting the model.

horizon: int = 1
Number of steps to forecast

start: None | int | DateLike = None
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' = 'origin'
Either 'origin' or 'target'. When set of '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' simplified computing forecast errors since the realization and h-step forecast are aligned.

method: ForecastingMethod = 'analytic'
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 = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None = None
Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

random_state: np.random.RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

reindex: bool = False

Whether to reindex the forecasts to have the same dimension as the series being forecast. Prior to 4.18 this was the default. As of 4.19 this is now optional. If not provided, a warning is raised about the future change in the default which will occur after September 2021.

New in version 4.19.

x: None | dict[Label, ArrayLike] | ArrayLike = None

Values to use for exogenous regressors if any are included in the model. Three formats are accepted:

- 2-d array-like: This format can be used when there is a single exogenous variable. The input must have shape (nforecast, horizon) or (nobs, horizon) where nforecast is the number of forecasting periods and nobs is the original shape of y. For example, if a single series of forecasts are made from the end of the sample with a horizon of 10, then the input can be (1, 10). Alternatively, if the original data had 1000 observations, then the input can be (1000, 10), and only the final row is used to produce forecasts.
- A dictionary of 2-d array-like: This format is identical to the previous except that the dictionary keys must match the names of the exog variables. Requires that the exog variables were passed as a pandas DataFrame.
- A 3-d NumPy array (or equivalent). In this format, each panel (0th axis) is a 2-d array that must have shape (nforecast, horizon) or (nobs,horizon). The array x[j] corresponds to the j-th column of the exogenous variables.

Due to the complexity required to accommodate all scenarios, please see the example notebook that demonstrates the valid formats for x.

New in version 4.19.

Returns

Container for forecasts. Key properties are `mean`, `variance` and `residual_variance`.

Return type

arch.univariate.base.ARCHModelForecast

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],  vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot()
```

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.

arch.univariate.ARX.parameter_names

`ARX.parameter_names()` → list[str]

List of parameters names

Returns
names – List of variable names for the mean model

Return type
`list (str)`

arch.univariate.ARX.resids

`ARX.resids(params: ndarray, y: ndarray | Series | None = None, regressors: ndarray | DataFrame | None = None)` → ndarray | Series

Compute model residuals

Parameters

params: ndarray
Model parameters

y: ndarray | Series | None = None
Alternative values to use when computing model residuals

regressors: ndarray | DataFrame | None = None
Alternative regressor values to use when computing model residuals

Returns
resids – Model residuals

Return type
`numpy.ndarray`

arch.univariate.ARX.simulate

ARX.**simulate**(params: *ArrayLike1D | collections.abc.Sequence[float]*, nobs: *int*, burn: *int = 500*, initial_value: *None | float | Float64Array = None*, x: *ArrayLike | None = None*, initial_value_vol: *None | float | Float64Array = None*) → pd.DataFrame

Simulates data from a linear regression, AR or HAR models

Parameters

params: ArrayLike1D | collections.abc.Sequence[float]
Parameters to use when simulating the model. Parameter order is [mean volatility distribution] where the parameters of the mean model are ordered [constant lag[0] lag[1] … lag[p] ex[0] … ex[k-1]] where lag[j] indicates the coefficient on the jth lag in the model and ex[j] is the coefficient on the jth exogenous variable.

nobs: int
Length of series to simulate

burn: int = 500
Number of values to simulate to initialize the model and remove dependence on initial values.

initial_value: None | float | Float64Array = None
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: ArrayLike | None = None
nobs + burn by k array of exogenous variables to include in the simulation.

initial_value_vol: None | float | Float64Array = None
An array or scalar to use when initializing the volatility process.

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

Return type
`pandas.DataFrame`

Examples

```
>>> import numpy as np
>>> from arch.univariate import HARX, GARCH
>>> harx = HARX(lags=[1, 5, 22])
>>> harx.volatility = GARCH()
>>> harx_params = np.array([1, 0.2, 0.3, 0.4])
>>> garch_params = np.array([0.01, 0.07, 0.92])
>>> params = np.concatenate((harx_params, garch_params))
>>> sim_data = harx.simulate(params, 1000)
```

Simulating models with exogenous regressors requires the regressors to have nobs plus burn data points

```
>>> nobs = 100
>>> burn = 200
>>> x = np.random.randn(nobs + burn, 2)
>>> x_params = np.array([1.0, 2.0])
```

(continues on next page)

(continued from previous page)

```
>>> params = np.concatenate((harx_params, x_params, garch_params))
>>> sim_data = harx.simulate(params, nobs=nobs, burn=burn, x=x)
```

arch.univariate.ARX.starting_values

ARX.**starting_values**() → ndarray

Returns starting values for the mean model, often the same as the values returned from fit

Returns
sv – Starting values

Return type
`numpy.ndarray`

Properties

distribution	Set or gets the error distribution
name	The name of the model.
num_params	Returns the number of parameters
volatility	Set or gets the volatility process
x	Gets the value of the exogenous regressors in the model
y	Returns the dependent variable

arch.univariate.ARX.distribution

property ARX.**distribution** : *Distribution*

Set or gets the error distribution

Distributions must be a subclass of Distribution

arch.univariate.ARX.name

property ARX.**name** : str

The name of the model.

arch.univariate.ARX.num_params

property ARX.**num_params** : int

Returns the number of parameters

arch.univariate.ARX.volatility

property ARX.**volatility** : *VolatilityProcess*

Set or gets the volatility process

Volatility processes must be a subclass of VolatilityProcess

arch.univariate.ARX.x

property ARX.**x** : ndarray | DataFrame | None

Gets the value of the exogenous regressors in the model

arch.univariate.ARX.y

property ARX.**y** : ndarray | DataFrame | Series | None

Returns the dependent variable

1.8.4 arch.univariate.HARX

class arch.univariate.**HARX**(y: *ndarray | DataFrame | Series | None* = **None**, x: *ndarray | DataFrame | None* = **None**, lags: *None | int | Sequence[int] | Sequence[Sequence[int]] | ndarray* = **None**, constant: *bool* = **True**, use_rotated: *bool* = **False**, hold_back: *int | None* = **None**, volatility: VolatilityProcess | *None* = **None**, distribution: Distribution | *None* = **None**, rescale: *bool | None* = **None**)

Heterogeneous Autoregression (HAR), with optional exogenous regressors, model estimation and simulation

Parameters

y: ndarray | DataFrame | Series | None = None
nobs element vector containing the dependent variable

x: ndarray | DataFrame | None = None
nobs by k element array containing exogenous regressors

lags: None | int | Sequence[int] | Sequence[Sequence[int]] | ndarray = None

Description of lag structure of the HAR.

- Scalar included all lags between 1 and the value.
- A 1-d n-element array includes the HAR lags 1:lags[0]+1, 1:lags[1]+1, … 1:lags[n]+1.
- A 2-d (2,n)-element array that includes the HAR lags of the form lags[0,j]:lags[1,j]+1 for all columns of lags.

constant: bool = True
Flag whether the model should include a constant

use_rotated: bool = False
Flag indicating to use the alternative rotated form of the HAR where HAR lags do not overlap

hold_back: int | None = None
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.

volatility: *VolatilityProcess* | None = None
Volatility process to use in the model

distribution: *Distribution* | None = None
Error distribution to use in the model

rescale: bool | None = None
Flag indicating whether to automatically rescale data if the scale of the data is likely to produce convergence issues when estimating model parameters. If False, the model is estimated on the data without transformation. If True, than y is rescaled and the new scale is reported in the estimation results.

Examples

Standard HAR with average lags 1, 5 and 22

```
>>> import numpy as np
>>> from arch.univariate import HARX
>>> y = np.random.RandomState(1234).randn(100)
>>> harx = HARX(y, lags=[1, 5, 22])
>>> res = harx.fit()
```

A standard HAR with average lags 1 and 6 but holding back 10 observations

```
>>> from pandas import Series, date_range
>>> index = date_range('2000-01-01', freq='M', periods=y.shape[0])
>>> y = Series(y, name='y', index=index)
>>> har = HARX(y, lags=[1, 6], hold_back=10)
```

Models with equivalent parametrizations of lags. The first uses overlapping lags.

```
>>> harx_1 = HARX(y, lags=[1,5,22])
```

The next uses rotated lags so that they do not overlap.

```
>>> harx_2 = HARX(y, lags=[1,5,22], use_rotated=True)
```

The third manually specified overlapping lags.

```
>>> harx_3 = HARX(y, lags=[[1, 1, 1], [1, 5, 22]])
```

The final manually specified non-overlapping lags

```
>>> harx_4 = HARX(y, lags=[[1, 2, 6], [1, 5, 22]])
```

It is simple to verify that these are the equivalent by inspecting the R2.

```
>>> models = [harx_1, harx_2, harx_3, harx_4]
>>> print([mod.fit().rsquared for mod in models])
0.085, 0.085, 0.085, 0.085
```

Notes

The HAR-X model is described by

$$y_t = \mu + \sum_{i=1}^{p} \phi_{L_i} \bar{y}_{t-L_{i,0}:L_{i,1}} + \gamma' x_t + \epsilon_t$$

where $\bar{y}_{t-L_{i,0}:L_{i,1}}$ is the average value of y_t between $t - L_{i,0}$ and $t - L_{i,1}$.

Methods

bounds()	Construct bounds for parameters to use in non-linear optimization
compute_param_cov(params[, backcast, robust])	Computes parameter covariances using numerical derivatives.
constraints()	Construct linear constraint arrays for use in non-linear optimization
fit([update_freq, disp, starting_values, ...])	Estimate model parameters
fix(params[, first_obs, last_obs])	Allows an ARCHModelFixedResult to be constructed from fixed parameters.
forecast(params[, horizon, start, align, ...])	Construct forecasts from estimated model
parameter_names()	List of parameters names
resids(params[, y, regressors])	Compute model residuals
simulate(params, nobs[, burn, ...])	Simulates data from a linear regression, AR or HAR models
starting_values()	Returns starting values for the mean model, often the same as the values returned from fit

arch.univariate.HARX.bounds

HARX.**bounds**() → list[tuple[float, float]]

Construct bounds for parameters to use in non-linear optimization

Returns

bounds – Bounds for parameters to use in estimation.

Return type

`list (2-tuple of float)`

arch.univariate.HARX.compute_param_cov

HARX.**compute_param_cov**(params: *ndarray*, backcast: *None | float | ndarray* = **None**, robust: *bool* = **True**) → ndarray

Computes parameter covariances using numerical derivatives.

Parameters

params: ndarray
Model parameters

backcast: None | float | ndarray = None
Value to use for pre-sample observations

robust: bool = True
Flag indicating whether to use robust standard errors (True) or classic MLE (False)

arch.univariate.HARX.constraints

`HARX.constraints()` → tuple[ndarray, ndarray]

Construct linear constraint arrays for use in non-linear optimization

Returns

- **a** (`numpy.ndarray`) – Number of constraints by number of parameters loading array
- **b** (`numpy.ndarray`) – Number of constraints array of lower bounds

Notes

Parameters satisfy a.dot(parameters) - b >= 0

arch.univariate.HARX.fit

`HARX.fit`(update_freq: *int* = **1**, disp: *bool* | *'off'* | *'final'* = `'final'`, starting_values: *ndarray* | *Series* | *None* = **None**, cov_type: *'robust'* | *'classic'* = `'robust'`, show_warning: *bool* = **True**, first_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = **None**, last_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = **None**, tol: *float* | *None* = **None**, options: *dict[str, Any]* | *None* = **None**, backcast: *None* | *float* | *ndarray* = **None**) → *ARCHModelResult*

Estimate model parameters

Parameters

update_freq: int = 1
Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.

disp: bool | 'off' | 'final' = 'final'
Either 'final' to print optimization result or 'off' to display nothing. If using a boolean, False is "off" and True is "final"

starting_values: ndarray | Series | None = None
Array of starting values to use. If not provided, starting values are constructed by the model components.

cov_type: 'robust' | 'classic' = 'robust'
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 = True
Flag indicating whether convergence warnings should be shown.

first_obs: int | str | datetime | datetime64 | Timestamp | None = None
First observation to use when estimating model

last_obs: int | str | datetime | datetime64 | Timestamp | None = None
Last observation to use when estimating model

tol: float | None = None
Tolerance for termination.

options: dict[str, Any] | None = None
Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.

backcast: None | float | ndarray = None
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
results – Object containing model results

Return type
`ARCHModelResult`

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

arch.univariate.HARX.fix

`HARX.fix`(params*: Sequence[float]* | *ndarray* | *Series*, first_obs*: int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = **None**, last_obs*: int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = **None**) → *ARCHModelFixedResult*

Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

params: Sequence[float] | ndarray | Series
User specified parameters to use when generating the result. Must have the correct number of parameters for a given choice of mean model, volatility model and distribution.

first_obs: int | str | datetime | datetime64 | Timestamp | None = None
First observation to use when fixing model

last_obs: int | str | datetime | datetime64 | Timestamp | None = None
Last observation to use when fixing model

Returns
results – Object containing model results

Return type
`ARCHModelFixedResult`

Notes

Parameters are not checked against model-specific constraints.

arch.univariate.HARX.forecast

HARX.**forecast**(params: *ArrayLike1D*, horizon: *int* = *1*, start: *None* | *int* | *DateLike* = **None**, align: *'origin'* | *'target'* = 'origin', method: *ForecastingMethod* = 'analytic', simulations: *int* = *1000*, rng: *collections.abc.Callable[[int | tuple[int, ...]], Float64Array]* | *None* = **None**, random_state: *np.random.RandomState* | *None* = **None**, *, reindex: *bool* = **False**, x: *None* | *dict[Label, ArrayLike]* | *ArrayLike* = **None**) → *ARCHModelForecast*

Construct forecasts from estimated model

Parameters

params: ArrayLike1D
Parameters required to forecast. Must be identical in shape to the parameters computed by fitting the model.

horizon: int = 1
Number of steps to forecast

start: None | int | DateLike = None
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' = 'origin'
Either 'origin' or 'target'. When set of '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' simplified computing forecast errors since the realization and h-step forecast are aligned.

method: ForecastingMethod = 'analytic'
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 = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None = None
Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

random_state: np.random.RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

reindex: bool = False

Whether to reindex the forecasts to have the same dimension as the series being forecast. Prior to 4.18 this was the default. As of 4.19 this is now optional. If not provided, a warning is raised about the future change in the default which will occur after September 2021.

New in version 4.19.

x: None | dict[Label, ArrayLike] | ArrayLike = None

Values to use for exogenous regressors if any are included in the model. Three formats are accepted:

- 2-d array-like: This format can be used when there is a single exogenous variable. The input must have shape (nforecast, horizon) or (nobs, horizon) where nforecast is the number of forecasting periods and nobs is the original shape of y. For example, if a single series of forecasts are made from the end of the sample with a horizon of 10, then the input can be (1, 10). Alternatively, if the original data had 1000 observations, then the input can be (1000, 10), and only the final row is used to produce forecasts.
- A dictionary of 2-d array-like: This format is identical to the previous except that the dictionary keys must match the names of the exog variables. Requires that the exog variables were passed as a pandas DataFrame.
- A 3-d NumPy array (or equivalent). In this format, each panel (0th axis) is a 2-d array that must have shape (nforecast, horizon) or (nobs,horizon). The array x[j] corresponds to the j-th column of the exogenous variables.

Due to the complexity required to accommodate all scenarios, please see the example notebook that demonstrates the valid formats for x.

New in version 4.19.

Returns

Container for forecasts. Key properties are `mean`, `variance` and `residual_variance`.

Return type

arch.univariate.base.ARCHModelForecast

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],  vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot()
```

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.

arch.univariate.HARX.parameter_names

HARX.**parameter_names**() → list[str]

List of parameters names

Returns
names – List of variable names for the mean model

Return type
`list (str)`

arch.univariate.HARX.resids

HARX.**resids**(params: *ndarray*, y: *ndarray | Series | None = None*, regressors: *ndarray | DataFrame | None = None*) → ndarray | Series

Compute model residuals

Parameters

params: ndarray
Model parameters

y: ndarray | Series | None = None
Alternative values to use when computing model residuals

regressors: ndarray | DataFrame | None = None
Alternative regressor values to use when computing model residuals

Returns
resids – Model residuals

Return type
`numpy.ndarray`

arch.univariate.HARX.simulate

HARX.**simulate**(params: *ArrayLike1D | collections.abc.Sequence[float]*, nobs: *int*, burn: *int = 500*, initial_value: *None | float | Float64Array = None*, x: *ArrayLike | None = None*, initial_value_vol: *None | float | Float64Array = None*) → pd.DataFrame

Simulates data from a linear regression, AR or HAR models

Parameters

params: ArrayLike1D | collections.abc.Sequence[float]
Parameters to use when simulating the model. Parameter order is [mean volatility distribution] where the parameters of the mean model are ordered [constant lag[0] lag[1] … lag[p] ex[0] … ex[k-1]] where lag[j] indicates the coefficient on the jth lag in the model and ex[j] is the coefficient on the jth exogenous variable.

nobs: int
Length of series to simulate

burn: int = 500
Number of values to simulate to initialize the model and remove dependence on initial values.

initial_value: None | float | Float64Array = None
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: ArrayLike | None = None
nobs + burn by k array of exogenous variables to include in the simulation.

initial_value_vol: None | float | Float64Array = None
An array or scalar to use when initializing the volatility process.

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

Return type
pandas.DataFrame

Examples

```
>>> import numpy as np
>>> from arch.univariate import HARX, GARCH
>>> harx = HARX(lags=[1, 5, 22])
>>> harx.volatility = GARCH()
>>> harx_params = np.array([1, 0.2, 0.3, 0.4])
>>> garch_params = np.array([0.01, 0.07, 0.92])
>>> params = np.concatenate((harx_params, garch_params))
>>> sim_data = harx.simulate(params, 1000)
```

Simulating models with exogenous regressors requires the regressors to have nobs plus burn data points

```
>>> nobs = 100
>>> burn = 200
>>> x = np.random.randn(nobs + burn, 2)
>>> x_params = np.array([1.0, 2.0])
>>> params = np.concatenate((harx_params, x_params, garch_params))
>>> sim_data = harx.simulate(params, nobs=nobs, burn=burn, x=x)
```

arch.univariate.HARX.starting_values

HARX.**starting_values**() → ndarray

Returns starting values for the mean model, often the same as the values returned from fit

Returns
sv – Starting values

Return type
numpy.ndarray

Properties

distribution	Set or gets the error distribution
name	The name of the model.
num_params	Returns the number of parameters
volatility	Set or gets the volatility process
x	Gets the value of the exogenous regressors in the model
y	Returns the dependent variable

arch.univariate.HARX.distribution

property HARX.**distribution** : *Distribution*

Set or gets the error distribution

Distributions must be a subclass of Distribution

arch.univariate.HARX.name

property HARX.**name** : str

The name of the model.

arch.univariate.HARX.num_params

property HARX.**num_params** : int

Returns the number of parameters

arch.univariate.HARX.volatility

property HARX.**volatility** : *VolatilityProcess*

Set or gets the volatility process

Volatility processes must be a subclass of VolatilityProcess

arch.univariate.HARX.x

property HARX.**x** : ndarray | DataFrame | None

Gets the value of the exogenous regressors in the model

arch.univariate.HARX.y

property HARX.y : ndarray | DataFrame | Series | None

Returns the dependent variable

1.8.5 arch.univariate.LS

class arch.univariate.LS(*y: ndarray | DataFrame | Series | None = None*, *x: ndarray | DataFrame | None = None*, *constant: bool = True*, *hold_back: int | None = None*, *volatility: VolatilityProcess | None = None*, *distribution: Distribution | None = None*, *rescale: bool | None = None*)

Least squares model estimation and simulation

Parameters

y: ndarray | DataFrame | Series | None = None
nobs element vector containing the dependent variable

y: ndarray | DataFrame | Series | None = None
nobs by k element array containing exogenous regressors

constant: bool = True
Flag whether the model should include a constant

hold_back: int | None = None
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.

volatility: *VolatilityProcess* | None = None
Volatility process to use in the model

distribution: *Distribution* | None = None
Error distribution to use in the model

rescale: bool | None = None
Flag indicating whether to automatically rescale data if the scale of the data is likely to produce convergence issues when estimating model parameters. If False, the model is estimated on the data without transformation. If True, than y is rescaled and the new scale is reported in the estimation results.

Examples

```
>>> import numpy as np
>>> from arch.univariate import LS
>>> y = np.random.randn(100)
>>> x = np.random.randn(100,2)
>>> ls = LS(y, x)
>>> res = ls.fit()
```

Notes

The LS model is described by

$$y_t = \mu + \gamma' x_t + \epsilon_t$$

Methods

bounds()	Construct bounds for parameters to use in non-linear optimization
compute_param_cov(params[, backcast, robust])	Computes parameter covariances using numerical derivatives.
constraints()	Construct linear constraint arrays for use in non-linear optimization
fit([update_freq, disp, starting_values, ...])	Estimate model parameters
fix(params[, first_obs, last_obs])	Allows an ARCHModelFixedResult to be constructed from fixed parameters.
forecast(params[, horizon, start, align, ...])	Construct forecasts from estimated model
parameter_names()	List of parameters names
resids(params[, y, regressors])	Compute model residuals
simulate(params, nobs[, burn, ...])	Simulates data from a linear regression, AR or HAR models
starting_values()	Returns starting values for the mean model, often the same as the values returned from fit

arch.univariate.LS.bounds

LS.**bounds**() → list[tuple[float, float]]

Construct bounds for parameters to use in non-linear optimization

Returns

bounds – Bounds for parameters to use in estimation.

Return type

`list (2-tuple of float)`

arch.univariate.LS.compute_param_cov

LS.**compute_param_cov**(params: *ndarray*, backcast: *None* | *float* | *ndarray* = **None**, robust: *bool* = **True**) → ndarray

Computes parameter covariances using numerical derivatives.

Parameters

params: ndarray
Model parameters

backcast: None | float | ndarray = None
Value to use for pre-sample observations

robust: bool = True
Flag indicating whether to use robust standard errors (True) or classic MLE (False)

arch.univariate.LS.constraints

LS.**constraints**() → tuple[ndarray, ndarray]

Construct linear constraint arrays for use in non-linear optimization

Returns

- **a** (numpy.ndarray) – Number of constraints by number of parameters loading array
- **b** (numpy.ndarray) – Number of constraints array of lower bounds

Notes

Parameters satisfy a.dot(parameters) - b >= 0

arch.univariate.LS.fit

LS.**fit**(update_freq: *int* = *1*, disp: *bool* | *'off'* | *'final'* = 'final', starting_values: *ndarray* | *Series* | *None* = *None*, cov_type: *'robust'* | *'classic'* = 'robust', show_warning: *bool* = *True*, first_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = *None*, last_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = *None*, tol: *float* | *None* = *None*, options: *dict[str, Any]* | *None* = *None*, backcast: *None* | *float* | *ndarray* = *None*) → *ARCHModelResult*

Estimate model parameters

Parameters

update_freq: int = 1
Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.

disp: bool | 'off' | 'final' = 'final'
Either 'final' to print optimization result or 'off' to display nothing. If using a boolean, False is "off" and True is "final"

starting_values: ndarray | Series | None = None
Array of starting values to use. If not provided, starting values are constructed by the model components.

cov_type: 'robust' | 'classic' = 'robust'
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 = True
Flag indicating whether convergence warnings should be shown.

first_obs: int | str | datetime | datetime64 | Timestamp | None = None
First observation to use when estimating model

last_obs: int | str | datetime | datetime64 | Timestamp | None = None
Last observation to use when estimating model

tol: float | None = None
Tolerance for termination.

options: dict[str, Any] | None = None
Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.

backcast: None | float | ndarray = None
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
results – Object containing model results

Return type
`ARCHModelResult`

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

arch.univariate.LS.fix

`LS.fix`(params: *Sequence[float]* | *ndarray* | *Series*, first_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = `None`, last_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = `None`) → *ARCHModelFixedResult*

Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

params: Sequence[float] | ndarray | Series
User specified parameters to use when generating the result. Must have the correct number of parameters for a given choice of mean model, volatility model and distribution.

first_obs: int | str | datetime | datetime64 | Timestamp | None = None
First observation to use when fixing model

last_obs: int | str | datetime | datetime64 | Timestamp | None = None
Last observation to use when fixing model

Returns
results – Object containing model results

Return type
`ARCHModelFixedResult`

Notes

Parameters are not checked against model-specific constraints.

arch.univariate.LS.forecast

`LS.forecast`(params: *ArrayLike1D*, horizon: *int* = `1`, start: *None* | *int* | *DateLike* = `None`, align: *'origin'* | *'target'* = `'origin'`, method: *ForecastingMethod* = `'analytic'`, simulations: *int* = `1000`, rng: *collections.abc.Callable[[int* | *tuple[int, ...]], Float64Array]* | *None* = `None`, random_state: *np.random.RandomState* | *None* = `None`, *, reindex: *bool* = `False`, x: *None* | *dict[Label, ArrayLike]* | *ArrayLike* = `None`) → *ARCHModelForecast*

Construct forecasts from estimated model

Parameters

params: ArrayLike1D
Parameters required to forecast. Must be identical in shape to the parameters computed by fitting the model.

horizon: int = 1
Number of steps to forecast

start: None | int | DateLike = None
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' = 'origin'
Either 'origin' or 'target'. When set of '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' simplified computing forecast errors since the realization and h-step forecast are aligned.

method: ForecastingMethod = 'analytic'
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 = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None = None
Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

random_state: np.random.RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

reindex: bool = False

Whether to reindex the forecasts to have the same dimension as the series being forecast. Prior to 4.18 this was the default. As of 4.19 this is now optional. If not provided, a warning is raised about the future change in the default which will occur after September 2021.

New in version 4.19.

x: None | dict[Label, ArrayLike] | ArrayLike = None

Values to use for exogenous regressors if any are included in the model. Three formats are accepted:

- 2-d array-like: This format can be used when there is a single exogenous variable. The input must have shape (nforecast, horizon) or (nobs, horizon) where nforecast is the number of forecasting periods and nobs is the original shape of y. For example, if a single series of forecasts are made from the end of the sample with a horizon of 10, then the input can be (1, 10). Alternatively, if the original data had 1000 observations, then the input can be (1000, 10), and only the final row is used to produce forecasts.
- A dictionary of 2-d array-like: This format is identical to the previous except that the dictionary keys must match the names of the exog variables. Requires that the exog variables were passed as a pandas DataFrame.

- A 3-d NumPy array (or equivalent). In this format, each panel (0th axis) is a 2-d array that must have shape (nforecast, horizon) or (nobs,horizon). The array x[j] corresponds to the j-th column of the exogenous variables.

Due to the complexity required to accommodate all scenarios, please see the example notebook that demonstrates the valid formats for x.

New in version 4.19.

Returns
Container for forecasts. Key properties are `mean`, `variance` and `residual_variance`.

Return type
arch.univariate.base.ARCHModelForecast

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],  vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot()
```

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.

arch.univariate.LS.parameter_names

LS.**parameter_names**() → list[str]

List of parameters names

Returns
names – List of variable names for the mean model

Return type
`list (str)`

arch.univariate.LS.resids

LS.**resids**(params*: ndarray*, y*: ndarray | Series | None = None*, regressors*: ndarray | DataFrame | None = None*) → ndarray | Series

Compute model residuals

Parameters

params: ndarray
Model parameters

y: ndarray | Series | None = None
Alternative values to use when computing model residuals

regressors: ndarray | DataFrame | None = None
Alternative regressor values to use when computing model residuals

Returns
resids – Model residuals

Return type
numpy.ndarray

arch.univariate.LS.simulate

LS.**simulate**(params*: ArrayLike1D | collections.abc.Sequence[float]*, nobs*: int*, burn*: int = 500*, initial_value*: None | float | Float64Array = None*, x*: ArrayLike | None = None*, initial_value_vol*: None | float | Float64Array = None*) → pd.DataFrame

Simulates data from a linear regression, AR or HAR models

Parameters

params: ArrayLike1D | collections.abc.Sequence[float]
Parameters to use when simulating the model. Parameter order is [mean volatility distribution] where the parameters of the mean model are ordered [constant lag[0] lag[1] … lag[p] ex[0] … ex[k-1]] where lag[j] indicates the coefficient on the jth lag in the model and ex[j] is the coefficient on the jth exogenous variable.

nobs: int
Length of series to simulate

burn: int = 500
Number of values to simulate to initialize the model and remove dependence on initial values.

initial_value: None | float | Float64Array = None
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: ArrayLike | None = None
nobs + burn by k array of exogenous variables to include in the simulation.

initial_value_vol: None | float | Float64Array = None
An array or scalar to use when initializing the volatility process.

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

Return type
pandas.DataFrame

Examples

```
>>> import numpy as np
>>> from arch.univariate import HARX, GARCH
>>> harx = HARX(lags=[1, 5, 22])
>>> harx.volatility = GARCH()
>>> harx_params = np.array([1, 0.2, 0.3, 0.4])
>>> garch_params = np.array([0.01, 0.07, 0.92])
>>> params = np.concatenate((harx_params, garch_params))
>>> sim_data = harx.simulate(params, 1000)
```

Simulating models with exogenous regressors requires the regressors to have nobs plus burn data points

```
>>> nobs = 100
>>> burn = 200
>>> x = np.random.randn(nobs + burn, 2)
>>> x_params = np.array([1.0, 2.0])
>>> params = np.concatenate((harx_params, x_params, garch_params))
>>> sim_data = harx.simulate(params, nobs=nobs, burn=burn, x=x)
```

arch.univariate.LS.starting_values

LS.**starting_values**() → ndarray

Returns starting values for the mean model, often the same as the values returned from fit

Returns
sv – Starting values

Return type
numpy.ndarray

Properties

distribution	Set or gets the error distribution
name	The name of the model.
num_params	Returns the number of parameters
volatility	Set or gets the volatility process
x	Gets the value of the exogenous regressors in the model
y	Returns the dependent variable

arch.univariate.LS.distribution

property LS.**distribution** : *Distribution*

Set or gets the error distribution

Distributions must be a subclass of Distribution

arch.univariate.LS.name

property LS.**name** : str

The name of the model.

arch.univariate.LS.num_params

property LS.**num_params** : int

Returns the number of parameters

arch.univariate.LS.volatility

property LS.**volatility** : *VolatilityProcess*

Set or gets the volatility process

Volatility processes must be a subclass of VolatilityProcess

arch.univariate.LS.x

property LS.**x** : ndarray | DataFrame | None

Gets the value of the exogenous regressors in the model

arch.univariate.LS.y

property LS.**y** : ndarray | DataFrame | Series | None

Returns the dependent variable

1.8.6 (G)ARCH-in-mean Models

(G)ARCH-in-mean models allow the conditional variance (or a transformation of it) to enter the conditional mean.

ARCHInMean([y, x, lags, constant, ...])	(G)ARCH-in-mean model and simulation

arch.univariate.ARCHInMean

class arch.univariate.**ARCHInMean**(*y: ndarray | DataFrame | Series | None = None, x: ndarray | DataFrame | None = None, lags: None | int | list[int] | ndarray = None, constant: bool = True, hold_back: int | None = None, volatility: VolatilityProcess | None = None, distribution: Distribution | None = None, rescale: bool | None = None, form: int | float | 'log' | 'vol' | 'var' = 'vol'*)

(G)ARCH-in-mean model and simulation

Parameters

y: ndarray | DataFrame | Series | None = None
nobs element vector containing the dependent variable

x: ndarray | DataFrame | None = None
nobs by k element array containing exogenous regressors

lags: None | int | list[int] | ndarray = None
Description of lag structure of the HAR. Scalar included all lags between 1 and the value. A 1-d array includes the AR lags lags[0], lags[1], …

constant: bool = True
Flag whether the model should include a constant

hold_back: int | None = None
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.

volatility: *VolatilityProcess* | None = None
Volatility process to use in the model. `volatility.updateable` must return `True`.

distribution: *Distribution* | None = None
Error distribution to use in the model

rescale: bool | None = None
Flag indicating whether to automatically rescale data if the scale of the data is likely to produce convergence issues when estimating model parameters. If False, the model is estimated on the data without transformation. If True, than y is rescaled and the new scale is reported in the estimation results.

form: int | float | 'log' | 'vol' | 'var' = 'vol'
The form of the conditional variance that appears in the mean equation. The string names use the log of the conditional variance ("log"), the square-root of the conditional variance ("vol") or the conditional variance. When specified using a float, interpreted as σ_t^{form} so that 1 is equivalent to "vol" and 2 is equivalent to "var". When using a number, must be different from 0.

Examples

```
>>> import numpy as np
>>> from arch.univariate import ARCHInMean, GARCH
>>> from arch.data.sp500 import load
>>> sp500 = load()
>>> rets = 100 * sp500["Adj Close"].pct_change().dropna()
>>> gim = ARCHInMean(rets, lags=[1, 2], volatility=GARCH())
>>> res = gim.fit()
```

Notes

The (G)arch-in-mean model with exogenous regressors (-X) is described by

$$y_t = \mu + \kappa f(\sigma_t^2) + \sum_{i=1}^{p} \phi_{L_i} y_{t-L_i} + \gamma' x_t + \epsilon_t$$

where $f(\cdot)$ is the function specified by `form`.

Methods

`bounds`()	Construct bounds for parameters to use in non-linear optimization
`compute_param_cov`(params[, backcast, robust])	Computes parameter covariances using numerical derivatives.
`constraints`()	Construct linear constraint arrays for use in non-linear optimization
`fit`([update_freq, disp, starting_values, ...])	Estimate model parameters
`fix`(params[, first_obs, last_obs])	Allows an ARCHModelFixedResult to be constructed from fixed parameters.
`forecast`(params[, horizon, start, align, ...])	Construct forecasts from estimated model
`parameter_names`()	List of parameters names
`resids`(params[, y, regressors])	Compute model residuals
`simulate`(params, nobs[, burn, ...])	Simulates data from a linear regression, AR or HAR models
`starting_values`()	Returns starting values for the mean model, often the same as the values returned from fit

arch.univariate.ARCHInMean.bounds

`ARCHInMean.`**`bounds`**`()` → list[tuple[float, float]]

Construct bounds for parameters to use in non-linear optimization

Returns
bounds – Bounds for parameters to use in estimation.

Return type
`list (2-tuple` of `float)`

arch.univariate.ARCHInMean.compute_param_cov

`ARCHInMean.`**`compute_param_cov`**`(`params*: ndarray*, backcast*: None | float | ndarray* = ***None***, robust*: bool* = ***True***`)` → ndarray

Computes parameter covariances using numerical derivatives.

Parameters

params: ndarray
Model parameters

backcast: None | float | ndarray = None
Value to use for pre-sample observations

robust: bool = True
Flag indicating whether to use robust standard errors (True) or classic MLE (False)

arch.univariate.ARCHInMean.constraints

`ARCHInMean.constraints()` → tuple[ndarray, ndarray]

Construct linear constraint arrays for use in non-linear optimization

Returns

- **a** (`numpy.ndarray`) – Number of constraints by number of parameters loading array
- **b** (`numpy.ndarray`) – Number of constraints array of lower bounds

Notes

Parameters satisfy a.dot(parameters) - b >= 0

arch.univariate.ARCHInMean.fit

`ARCHInMean.fit(update_freq: int = 1, disp: bool | 'off' | 'final' = 'final', starting_values: ndarray | Series | None = None, cov_type: 'robust' | 'classic' = 'robust', show_warning: bool = True, first_obs: int | str | datetime | datetime64 | Timestamp | None = None, last_obs: int | str | datetime | datetime64 | Timestamp | None = None, tol: float | None = None, options: dict[str, Any] | None = None, backcast: None | float | ndarray = None)` → *ARCHModelResult*

Estimate model parameters

Parameters

update_freq: int = 1
Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.

disp: bool | 'off' | 'final' = 'final'
Either 'final' to print optimization result or 'off' to display nothing. If using a boolean, False is "off" and True is "final"

starting_values: ndarray | Series | None = None
Array of starting values to use. If not provided, starting values are constructed by the model components.

cov_type: 'robust' | 'classic' = 'robust'
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 = True
Flag indicating whether convergence warnings should be shown.

first_obs: int | str | datetime | datetime64 | Timestamp | None = None
First observation to use when estimating model

last_obs: int | str | datetime | datetime64 | Timestamp | None = None
Last observation to use when estimating model

tol: float | None = None
Tolerance for termination.

options: dict[str, Any] | None = None
Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.

backcast: None | float | ndarray = None
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
results – Object containing model results

Return type
`ARCHModelResult`

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

arch.univariate.ARCHInMean.fix

`ARCHInMean.fix`(params: *Sequence[float]* | *ndarray* | *Series*, first_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = ***None***, last_obs: *int* | *str* | *datetime* | *datetime64* | *Timestamp* | *None* = ***None***) → *ARCHModelFixedResult*

Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

params: Sequence[float] | ndarray | Series
User specified parameters to use when generating the result. Must have the correct number of parameters for a given choice of mean model, volatility model and distribution.

first_obs: int | str | datetime | datetime64 | Timestamp | None = None
First observation to use when fixing model

last_obs: int | str | datetime | datetime64 | Timestamp | None = None
Last observation to use when fixing model

Returns
results – Object containing model results

Return type
`ARCHModelFixedResult`

Notes

Parameters are not checked against model-specific constraints.

arch.univariate.ARCHInMean.forecast

ARCHInMean.**forecast**(params: *ArrayLike1D*, horizon: *int* = *1*, start: *None* | *int* | *DateLike* = *None*, align: *'origin'* | *'target'* = `'origin'`, method: *ForecastingMethod* = `'analytic'`, simulations: *int* = *1000*, rng: *collections.abc.Callable[[int* | *tuple[int, ...]], Float64Array]* | *None* = *None*, random_state: *np.random.RandomState* | *None* = *None*, *, reindex: *bool* | *None* = *None*, x: *None* | *dict[Label, ArrayLike]* | *ArrayLike* = *None*) → *ARCHModelForecast*

Construct forecasts from estimated model

Parameters

params: ArrayLike1D
Parameters required to forecast. Must be identical in shape to the parameters computed by fitting the model.

horizon: int = 1
Number of steps to forecast

start: None | int | DateLike = None
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' = 'origin'
Either 'origin' or 'target'. When set of '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' simplified computing forecast errors since the realization and h-step forecast are aligned.

method: ForecastingMethod = 'analytic'
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 = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None = None
Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

random_state: np.random.RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

reindex: bool | None = None

Whether to reindex the forecasts to have the same dimension as the series being forecast. Prior to 4.18 this was the default. As of 4.19 this is now optional. If not provided, a warning is raised about the future change in the default which will occur after September 2021.

New in version 4.19.

x: None | dict[Label, ArrayLike] | ArrayLike = None

Values to use for exogenous regressors if any are included in the model. Three formats are accepted:

- 2-d array-like: This format can be used when there is a single exogenous variable. The input must have shape (nforecast, horizon) or (nobs, horizon) where nforecast is the number of forecasting periods and nobs is the original shape of y. For example, if a single series of forecasts are made from the end of the sample with a horizon of 10, then the input can be (1, 10). Alternatively, if the original data had 1000 observations, then the input can be (1000, 10), and only the final row is used to produce forecasts.
- A dictionary of 2-d array-like: This format is identical to the previous except that the dictionary keys must match the names of the exog variables. Requires that the exog variables were passed as a pandas DataFrame.
- A 3-d NumPy array (or equivalent). In this format, each panel (0th axis) is a 2-d array that must have shape (nforecast, horizon) or (nobs,horizon). The array x[j] corresponds to the j-th column of the exogenous variables.

Due to the complexity required to accommodate all scenarios, please see the example notebook that demonstrates the valid formats for x.

New in version 4.19.

Returns

Container for forecasts. Key properties are `mean`, `variance` and `residual_variance`.

Return type

arch.univariate.base.ARCHModelForecast

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],  vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot()
```

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.

arch.univariate.ARCHInMean.parameter_names

ARCHInMean.**parameter_names**() → list[str]

List of parameters names

Returns
names – List of variable names for the mean model

Return type
`list (str)`

arch.univariate.ARCHInMean.resids

ARCHInMean.**resids**(params: *ndarray*, y: *ndarray* | *Series* | *None* = ***None***, regressors: *ndarray* | *DataFrame* | *None* = ***None***) → ndarray | Series

Compute model residuals

Parameters

params: ndarray
Model parameters

y: ndarray | Series | None = None
Alternative values to use when computing model residuals

regressors: ndarray | DataFrame | None = None
Alternative regressor values to use when computing model residuals

Returns
resids – Model residuals

Return type
`numpy.ndarray`

arch.univariate.ARCHInMean.simulate

ARCHInMean.**simulate**(params: *ArrayLike1D* | *collections.abc.Sequence[float]*, nobs: *int*, burn: *int* = ***500***, initial_value: *None* | *float* | *Float64Array* = ***None***, x: *ArrayLike* | *None* = ***None***, initial_value_vol: *None* | *float* | *Float64Array* = ***None***) → pd.DataFrame

Simulates data from a linear regression, AR or HAR models

Parameters

params: ArrayLike1D | collections.abc.Sequence[float]
Parameters to use when simulating the model. Parameter order is [mean volatility distribution] where the parameters of the mean model are ordered [constant lag[0] lag[1] … lag[p] ex[0] … ex[k-1]] where lag[j] indicates the coefficient on the jth lag in the model and ex[j] is the coefficient on the jth exogenous variable.

nobs: int
Length of series to simulate

burn: int = 500
Number of values to simulate to initialize the model and remove dependence on initial values.

initial_value: None | float | Float64Array = None
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: ArrayLike | None = None
nobs + burn by k array of exogenous variables to include in the simulation.

initial_value_vol: None | float | Float64Array = None
An array or scalar to use when initializing the volatility process.

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

Return type
`pandas.DataFrame`

Examples

```
>>> import numpy as np
>>> from arch.univariate import HARX, GARCH
>>> harx = HARX(lags=[1, 5, 22])
>>> harx.volatility = GARCH()
>>> harx_params = np.array([1, 0.2, 0.3, 0.4])
>>> garch_params = np.array([0.01, 0.07, 0.92])
>>> params = np.concatenate((harx_params, garch_params))
>>> sim_data = harx.simulate(params, 1000)
```

Simulating models with exogenous regressors requires the regressors to have nobs plus burn data points

```
>>> nobs = 100
>>> burn = 200
>>> x = np.random.randn(nobs + burn, 2)
>>> x_params = np.array([1.0, 2.0])
>>> params = np.concatenate((harx_params, x_params, garch_params))
>>> sim_data = harx.simulate(params, nobs=nobs, burn=burn, x=x)
```

arch.univariate.ARCHInMean.starting_values

`ARCHInMean.`**`starting_values`**`()` → ndarray

Returns starting values for the mean model, often the same as the values returned from fit

Returns
sv – Starting values

Return type
`numpy.ndarray`

Properties

distribution	Set or gets the error distribution
form	The form of the conditional variance in the mean
name	The name of the model.
num_params	Returns the number of parameters
volatility	Set or gets the volatility process
x	Gets the value of the exogenous regressors in the model
y	Returns the dependent variable

arch.univariate.ARCHInMean.distribution

property ARCHInMean.**distribution** : *Distribution*

Set or gets the error distribution

Distributions must be a subclass of Distribution

arch.univariate.ARCHInMean.form

property ARCHInMean.**form** : int | float | 'log' | 'vol' | 'var'

The form of the conditional variance in the mean

arch.univariate.ARCHInMean.name

property ARCHInMean.**name** : str

The name of the model.

arch.univariate.ARCHInMean.num_params

property ARCHInMean.**num_params** : int

Returns the number of parameters

arch.univariate.ARCHInMean.volatility

property ARCHInMean.**volatility** : *VolatilityProcess*

Set or gets the volatility process

Volatility processes must be a subclass of VolatilityProcess

arch.univariate.ARCHInMean.x

property ARCHInMean.**x** : ndarray | DataFrame | None

Gets the value of the exogenous regressors in the model

arch.univariate.ARCHInMean.y

property ARCHInMean.**y** : ndarray | DataFrame | Series | None

Returns the dependent variable

Special Requirements

Not all volatility processes support application to AIM modeling. Specifically, the property `updateable` must be `True`.

```
In [1]: from arch.univariate import GARCH, EGARCH

In [2]: GARCH().updateable
Out[2]: True

In [3]: EGARCH().updateable
Out[3]: True
```

1.8.7 Writing New Mean Models

All mean models must inherit from :class:ARCHModel and provide all public methods. There are two optional private methods that should be provided if applicable.

ARCHModel([y, volatility, distribution, ...])	Abstract base class for mean models in ARCH processes.

arch.univariate.base.ARCHModel

class arch.univariate.base.**ARCHModel**(*y: ndarray | DataFrame | Series | None =* **None**, *volatility:* VolatilityProcess | *None =* **None**, *distribution:* Distribution | *None =* **None**, *hold_back: int | None =* **None**, *rescale: bool | None =* **None**)

Abstract base class for mean models in ARCH processes. Specifies the conditional mean process.

All public methods that raise NotImplementedError should be overridden by any subclass. Private methods that raise NotImplementedError are optional to override but recommended where applicable.

Methods

bounds()	Construct bounds for parameters to use in non-linear optimization
compute_param_cov(params[, backcast, robust])	Computes parameter covariances using numerical derivatives.
constraints()	Construct linear constraint arrays for use in non-linear optimization
fit([update_freq, disp, starting_values, ...])	Estimate model parameters
fix(params[, first_obs, last_obs])	Allows an ARCHModelFixedResult to be constructed from fixed parameters.
forecast(params[, horizon, start, align, ...])	Construct forecasts from estimated model
parameter_names()	List of parameters names
resids(params[, y, regressors])	Compute model residuals
simulate(params, nobs[, burn, ...])	
starting_values()	Returns starting values for the mean model, often the same as the values returned from fit

arch.univariate.base.ARCHModel.bounds

ARCHModel.**bounds**() → list[tuple[float, float]]

Construct bounds for parameters to use in non-linear optimization

Returns

bounds – Bounds for parameters to use in estimation.

Return type

`list (2-tuple of float)`

arch.univariate.base.ARCHModel.compute_param_cov

ARCHModel.**compute_param_cov**(params: *ndarray*, backcast: *None* | *float* | *ndarray* = ***None***, robust: *bool* = ***True***) → ndarray

Computes parameter covariances using numerical derivatives.

Parameters

params: ndarray
Model parameters

backcast: None | float | ndarray = None
Value to use for pre-sample observations

robust: bool = True
Flag indicating whether to use robust standard errors (True) or classic MLE (False)

arch.univariate.base.ARCHModel.constraints

`ARCHModel.constraints()` → tuple[ndarray, ndarray]

Construct linear constraint arrays for use in non-linear optimization

Returns

- **a** (`numpy.ndarray`) – Number of constraints by number of parameters loading array
- **b** (`numpy.ndarray`) – Number of constraints array of lower bounds

Notes

Parameters satisfy a.dot(parameters) - b >= 0

arch.univariate.base.ARCHModel.fit

`ARCHModel.fit(update_freq: int = 1, disp: bool | 'off' | 'final' = 'final', starting_values: ndarray | Series | None = None, cov_type: 'robust' | 'classic' = 'robust', show_warning: bool = True, first_obs: int | str | datetime | datetime64 | Timestamp | None = None, last_obs: int | str | datetime | datetime64 | Timestamp | None = None, tol: float | None = None, options: dict[str, Any] | None = None, backcast: None | float | ndarray = None)` → *ARCHModelResult*

Estimate model parameters

Parameters

update_freq: int = 1
Frequency of iteration updates. Output is generated every *update_freq* iterations. Set to 0 to disable iterative output.

disp: bool | 'off' | 'final' = 'final'
Either 'final' to print optimization result or 'off' to display nothing. If using a boolean, False is "off" and True is "final"

starting_values: ndarray | Series | None = None
Array of starting values to use. If not provided, starting values are constructed by the model components.

cov_type: 'robust' | 'classic' = 'robust'
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 = True
Flag indicating whether convergence warnings should be shown.

first_obs: int | str | datetime | datetime64 | Timestamp | None = None
First observation to use when estimating model

last_obs: int | str | datetime | datetime64 | Timestamp | None = None
Last observation to use when estimating model

tol: float | None = None
Tolerance for termination.

options: dict[str, Any] | None = None
Options to pass to *scipy.optimize.minimize*. Valid entries include 'ftol', 'eps', 'disp', and 'maxiter'.

backcast: None | float | ndarray = None
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
results – Object containing model results

Return type
ARCHModelResult

Notes

A ConvergenceWarning is raised if SciPy's optimizer indicates difficulty finding the optimum.

Parameters are optimized using SLSQP.

arch.univariate.base.ARCHModel.fix

ARCHModel.**fix**(params: *Sequence[float] | ndarray | Series*, first_obs: *int | str | datetime | datetime64 | Timestamp | None* = **None**, last_obs: *int | str | datetime | datetime64 | Timestamp | None* = **None**) → *ARCHModelFixedResult*

Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

params: Sequence[float] | ndarray | Series
User specified parameters to use when generating the result. Must have the correct number of parameters for a given choice of mean model, volatility model and distribution.

first_obs: int | str | datetime | datetime64 | Timestamp | None = None
First observation to use when fixing model

last_obs: int | str | datetime | datetime64 | Timestamp | None = None
Last observation to use when fixing model

Returns
results – Object containing model results

Return type
ARCHModelFixedResult

Notes

Parameters are not checked against model-specific constraints.

arch.univariate.base.ARCHModel.forecast

abstract ARCHModel.**forecast**(params: *ArrayLike1D*, horizon: *int* = **1**, start: *int | DateLike | None* = **None**, align: *'origin' | 'target'* = 'origin', method: *ForecastingMethod* = 'analytic', simulations: *int* = **1000**, rng: *collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None* = **None**, random_state: *np.random.RandomState | None* = **None**, *, reindex: *bool* = **False**, x: *None | dict[Label, ArrayLike] | ArrayLike* = **None**) → *ARCHModelForecast*

Construct forecasts from estimated model

Parameters

params: ArrayLike1D
Parameters required to forecast. Must be identical in shape to the parameters computed by fitting the model.

horizon: int = 1
Number of steps to forecast

start: int | DateLike | None = None
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' = 'origin'
Either 'origin' or 'target'. When set of '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' simplified computing forecast errors since the realization and h-step forecast are aligned.

method: ForecastingMethod = 'analytic'
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 = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None = None
Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

random_state: np.random.RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

reindex: bool = False

Whether to reindex the forecasts to have the same dimension as the series being forecast. Prior to 4.18 this was the default. As of 4.19 this is now optional. If not provided, a warning is raised about the future change in the default which will occur after September 2021.

New in version 4.19.

x: None | dict[Label, ArrayLike] | ArrayLike = None

Values to use for exogenous regressors if any are included in the model. Three formats are accepted:

- 2-d array-like: This format can be used when there is a single exogenous variable. The input must have shape (nforecast, horizon) or (nobs, horizon) where nforecast is the number of forecasting periods and nobs is the original shape of y. For example, if a single series of forecasts are made from the end of the sample with a horizon of 10, then the input can be (1, 10). Alternatively, if the original data had 1000 observations, then the input can be (1000, 10), and only the final row is used to produce forecasts.

- A dictionary of 2-d array-like: This format is identical to the previous except that the dictionary keys must match the names of the exog variables. Requires that the exog variables were passed as a pandas DataFrame.
- A 3-d NumPy array (or equivalent). In this format, each panel (0th axis) is a 2-d array that must have shape (nforecast, horizon) or (nobs,horizon). The array x[j] corresponds to the j-th column of the exogenous variables.

Due to the complexity required to accommodate all scenarios, please see the example notebook that demonstrates the valid formats for x.

New in version 4.19.

Returns
Container for forecasts. Key properties are `mean`, `variance` and `residual_variance`.

Return type
arch.univariate.base.ARCHModelForecast

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],  vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot()
```

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.

arch.univariate.base.ARCHModel.parameter_names

abstract `ARCHModel.parameter_names()` → list[str]

List of parameters names

Returns
names – List of variable names for the mean model

Return type
`list (str)`

arch.univariate.base.ARCHModel.resids

abstract ARCHModel.**resids**(params: *ndarray*, y: *ndarray | Series | None* = **None**, regressors: *ndarray | DataFrame | None* = **None**) → ndarray | Series

Compute model residuals

Parameters

params: ndarray
Model parameters

y: ndarray | Series | None = None
Alternative values to use when computing model residuals

regressors: ndarray | DataFrame | None = None
Alternative regressor values to use when computing model residuals

Returns
resids – Model residuals

Return type
`numpy.ndarray`

arch.univariate.base.ARCHModel.simulate

abstract ARCHModel.**simulate**(*params: ArrayLike1D | collections.abc.Sequence[float], nobs: int, burn: int* = **500**, *initial_value: float | None* = **None**, *x: ArrayLike | None* = **None**, *initial_value_vol: float | None* = **None**) → pd.DataFrame

arch.univariate.base.ARCHModel.starting_values

ARCHModel.**starting_values**() → ndarray

Returns starting values for the mean model, often the same as the values returned from fit

Returns
sv – Starting values

Return type
`numpy.ndarray`

Properties

distribution	Set or gets the error distribution
name	The name of the model.
num_params	Number of parameters in the model
volatility	Set or gets the volatility process
y	Returns the dependent variable

arch.univariate.base.ARCHModel.distribution

property `ARCHModel.distribution` : *Distribution*

Set or gets the error distribution

Distributions must be a subclass of Distribution

arch.univariate.base.ARCHModel.name

property `ARCHModel.name` : str

The name of the model.

arch.univariate.base.ARCHModel.num_params

property `ARCHModel.num_params` : int

Number of parameters in the model

arch.univariate.base.ARCHModel.volatility

property `ARCHModel.volatility` : *VolatilityProcess*

Set or gets the volatility process

Volatility processes must be a subclass of VolatilityProcess

arch.univariate.base.ARCHModel.y

property `ARCHModel.y` : ndarray | DataFrame | Series | None

Returns the dependent variable

1.9 Volatility Processes

A volatility process is added to a mean model to capture time-varying volatility.

ConstantVariance()	Constant volatility process
GARCH([p, o, q, power])	GARCH and related model estimation
FIGARCH([p, q, power, truncation])	FIGARCH model
EGARCH([p, o, q])	EGARCH model estimation
HARCH([lags])	Heterogeneous ARCH process
MIDASHyperbolic([m, asym])	MIDAS Hyperbolic ARCH process
ARCH([p])	ARCH process
APARCH([p, o, q, delta, common_asym])	Asymmetric Power ARCH (APARCH) volatility process

1.9.1 arch.univariate.ConstantVariance

`class arch.univariate.ConstantVariance`

Constant volatility process

Notes

Model has the same variance in all periods

Methods

backcast(resids)	Construct values for backcasting to start the recursion
backcast_transform(backcast)	Transformation to apply to user-provided backcast values
bounds(resids)	Returns bounds for parameters
compute_variance(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
constraints()	Construct parameter constraints arrays for parameter estimation
forecast(parameters, resids, backcast, ...)	Forecast volatility from the model
parameter_names()	Names of model parameters
simulate(parameters, nobs, rng[, burn, ...])	Simulate data from the model
starting_values(resids)	Returns starting values for the ARCH model
update(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
variance_bounds(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.ConstantVariance.backcast

`ConstantVariance.backcast(resids: ndarray) → float | ndarray`

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
backcast – Value to use in backcasting in the volatility recursion

Return type
`float`

arch.univariate.ConstantVariance.backcast_transform

`ConstantVariance.backcast_transform(backcast: float | ndarray) → float | ndarray`

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns

backcast – Backcast transformed to the model-appropriate scale

Return type

`{float, ndarray}`

arch.univariate.ConstantVariance.bounds

`ConstantVariance.bounds(resids: ndarray) → list[tuple[float, float]]`

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns

bounds – List of bounds where each element is (lower, upper).

Return type

`list[tuple[float,float]]`

arch.univariate.ConstantVariance.compute_variance

`ConstantVariance.compute_variance(parameters: ndarray, resids: ndarray, sigma2: ndarray, backcast: float | ndarray, var_bounds: ndarray) → ndarray`

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.ConstantVariance.constraints

`ConstantVariance.constraints() → tuple[ndarray, ndarray]`

Construct parameter constraints arrays for parameter estimation

Returns

- **A** (`numpy.ndarray`) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (`numpy.ndarray`) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

arch.univariate.ConstantVariance.forecast

`ConstantVariance.forecast`(parameters: *ArrayLike1D*, resids: *Float64Array*, backcast: *Float64Array | float*, var_bounds: *Float64Array*, start: *int | None = None*, horizon: *int = 1*, method: *ForecastingMethod = 'analytic'*, simulations: *int = 1000*, rng: *RNGType | None = None*, random_state: *RandomState | None = None*) → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = None
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = 1
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = 'analytic'
Method to use when producing the forecast. The default is analytic.

simulations: int = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is 'simulation'. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

Returns
forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type
`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported

- ValueError –
  - If the method is not known

Notes

The analytic method is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.ConstantVariance.parameter_names

ConstantVariance.**parameter_names**() → list[str]

Names of model parameters

Returns
names – Variables names

Return type
list (str)

arch.univariate.ConstantVariance.simulate

ConstantVariance.**simulate**(parameters: *Sequence[float | int] | ndarray | Series*, nobs: *int*, rng: *Callable[[int | tuple[int, ...]], ndarray]*, burn: *int* = **500**, initial_value: *None | float | ndarray* = **None**) → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (numpy.ndarray) – The simulated residuals
- **variance** (numpy.ndarray) – The simulated variance

arch.univariate.ConstantVariance.starting_values

ConstantVariance.**starting_values**(resids: *ndarray*) → ndarray

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
`numpy.ndarray`

arch.univariate.ConstantVariance.update

ConstantVariance.**update**(index: *int*, parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float | ndarray*, var_bounds: *ndarray*) → float

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
`float`

arch.univariate.ConstantVariance.variance_bounds

ConstantVariance.**variance_bounds**(resids: *ndarray*, power: *float* = 2.0) → ndarray

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.ConstantVariance.name

property ConstantVariance.**name** : str

The name of the volatility process

arch.univariate.ConstantVariance.num_params

property ConstantVariance.**num_params** : int

The number of parameters in the model

arch.univariate.ConstantVariance.start

`property ConstantVariance.start : int`

Index to use to start variance subarray selection

arch.univariate.ConstantVariance.stop

`property ConstantVariance.stop : int`

Index to use to stop variance subarray selection

arch.univariate.ConstantVariance.updateable

`property ConstantVariance.updateable : bool`

Flag indicating that the volatility process supports update

arch.univariate.ConstantVariance.volatility_updater

`property ConstantVariance.volatility_updater : VolatilityUpdater`

Get the volatility updater associated with the volatility process

Returns
The updater class

Return type
`VolatilityUpdater`

Raises
`NotImplementedError` – If the process is not updateable

1.9.2 arch.univariate.GARCH

`class arch.univariate.GARCH(p: int = 1, o: int = 0, q: int = 1, power: float = 2.0)`

GARCH and related model estimation

The following models can be specified using GARCH:

- 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

Parameters

p: int = 1
Order of the symmetric innovation

o: int = 0
Order of the asymmetric innovation

q: int = 1
Order of the lagged (transformed) conditional variance

power: float = 2.0
Power to use with the innovations, abs(e) ** power. 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.

Examples

```
>>> from arch.univariate import GARCH
```

Standard GARCH(1,1)

```
>>> garch = GARCH(p=1, q=1)
```

Asymmetric GJR-GARCH process

```
>>> gjr = GARCH(p=1, o=1, q=1)
```

Asymmetric TARCH process

```
>>> tarch = GARCH(p=1, o=1, q=1, power=1.0)
```

Notes

In this class of processes, the variance dynamics are

$$\sigma_t^{\lambda} = \omega + \sum_{i=1}^{p} \alpha_i \left|\epsilon_{t-i}\right|^{\lambda} + \sum_{j=1}^{o} \gamma_j \left|\epsilon_{t-j}\right|^{\lambda} I\left[\epsilon_{t-j} < 0\right] + \sum_{k=1}^{q} \beta_k \sigma_{t-k}^{\lambda}$$

where λ is the `power`.

Methods

`backcast`(resids)	Construct values for backcasting to start the recursion
`backcast_transform`(backcast)	Transformation to apply to user-provided backcast values
`bounds`(resids)	Returns bounds for parameters
`compute_variance`(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
`constraints`()	Construct parameter constraints arrays for parameter estimation
`forecast`(parameters, resids, backcast, ...)	Forecast volatility from the model
`parameter_names`()	Names of model parameters
`simulate`(parameters, nobs, rng[, burn, ...])	Simulate data from the model
`starting_values`(resids)	Returns starting values for the ARCH model
`update`(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
`variance_bounds`(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.GARCH.backcast

GARCH.**backcast**(resids: *ndarray*) → float | ndarray

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
backcast – Value to use in backcasting in the volatility recursion

Return type
`float`

arch.univariate.GARCH.backcast_transform

GARCH.**backcast_transform**(backcast: *float* | *ndarray*) → float | ndarray

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns
backcast – Backcast transformed to the model-appropriate scale

Return type
`{float, ndarray}`

arch.univariate.GARCH.bounds

GARCH.**bounds**(resids: *ndarray*) → list[tuple[float, float]]

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
bounds – List of bounds where each element is (lower, upper).

Return type
`list[tuple[float,float]]`

arch.univariate.GARCH.compute_variance

GARCH.**compute_variance**(parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float* | *ndarray*, var_bounds: *ndarray*) → ndarray

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.GARCH.constraints

GARCH.**constraints**() → tuple[ndarray, ndarray]

Construct parameter constraints arrays for parameter estimation

Returns

- **A** (`numpy.ndarray`) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (`numpy.ndarray`) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

arch.univariate.GARCH.forecast

GARCH.**forecast**(parameters: *ArrayLike1D*, resids: *Float64Array*, backcast: *Float64Array* | *float*, var_bounds: *Float64Array*, start: *int* | *None* = **None**, horizon: *int* = **1**, method: *ForecastingMethod* = 'analytic', simulations: *int* = **1000**, rng: *RNGType* | *None* = **None**, random_state: *RandomState* | *None* = **None**) → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = None
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = 1
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = 'analytic'
Method to use when producing the forecast. The default is analytic.

simulations: int = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is 'simulation'. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

Returns
forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type
`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported
- `ValueError` –
  - If the method is not known

Notes

The analytic `method` is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.GARCH.parameter_names

`GARCH.parameter_names()` → list[str]

Names of model parameters

Returns
names – Variables names

Return type
`list (str)`

arch.univariate.GARCH.simulate

GARCH.**simulate**(parameters: *Sequence[float | int] | ndarray | Series*, nobs: *int*, rng: *Callable[[int | tuple[int, ...]], ndarray]*, burn: *int* = **500**, initial_value: *None | float | ndarray* = ***None***) → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (`numpy.ndarray`) – The simulated residuals
- **variance** (`numpy.ndarray`) – The simulated variance

arch.univariate.GARCH.starting_values

GARCH.**starting_values**(resids: *ndarray*) → ndarray

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
`numpy.ndarray`

arch.univariate.GARCH.update

GARCH.**update**(index: *int*, parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float | ndarray*, var_bounds: *ndarray*) → float

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
`float`

arch.univariate.GARCH.variance_bounds

`GARCH.variance_bounds(resids: ndarray, power: float = 2.0) → ndarray`

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.GARCH.name

property GARCH.**name** : str

The name of the volatility process

arch.univariate.GARCH.num_params

property GARCH.**num_params** : int

The number of parameters in the model

arch.univariate.GARCH.start

property GARCH.**start** : int

Index to use to start variance subarray selection

arch.univariate.GARCH.stop

property GARCH.**stop** : int

Index to use to stop variance subarray selection

arch.univariate.GARCH.updateable

property GARCH.**updateable** : bool

Flag indicating that the volatility process supports update

arch.univariate.GARCH.volatility_updater

property GARCH.**volatility_updater** : VolatilityUpdater

Get the volatility updater associated with the volatility process

Returns
The updater class

Return type
`VolatilityUpdater`

Raises
`NotImplementedError` – If the process is not updateable

1.9.3 arch.univariate.FIGARCH

class arch.univariate.**FIGARCH**(p: *int* = *1*, q: *int* = *1*, power: *float* = *2.0*, truncation: *int* = *1000*)

FIGARCH model

Parameters

p: int = 1
Order of the symmetric innovation

q: int = 1
Order of the lagged (transformed) conditional variance

power: float = 2.0
Power to use with the innovations, abs(e) ** power. Default is 2.0, which produces FIGARCH and related models. Using 1.0 produces FIAVARCH and related models. Other powers can be specified, although these should be strictly positive, and usually larger than 0.25.

truncation: int = 1000
Truncation point to use in ARCH(∞) representation. Default is 1000.

Examples

```
>>> from arch.univariate import FIGARCH
```

Standard FIGARCH

```
>>> figarch = FIGARCH()
```

FIARCH

```
>>> fiarch = FIGARCH(p=0)
```

FIAVGARCH process

```
>>> fiavarch = FIGARCH(power=1.0)
```

Notes

In this class of processes, the variance dynamics are

$$h_t = \omega + [1 - \beta L - \phi L(1 - L)^d]\epsilon_t^2 + \beta h_{t-1}$$

where `L` is the lag operator and `d` is the fractional differencing parameter. The model is estimated using the ARCH(∞) representation,

$$h_t = (1 - \beta)^{-1}\omega + \sum_{i=1}^{\infty} \lambda_i \epsilon_{t-i}^2$$

The weights are constructed using

$$\begin{aligned}\delta_1 &= d \\ \lambda_1 &= d - \beta + \phi\end{aligned}$$

and the recursive equations

$$\begin{aligned}\delta_j &= \frac{j - 1 - d}{j}\delta_{j-1} \\ \lambda_j &= \beta\lambda_{j-1} + \delta_j - \phi\delta_{j-1}.\end{aligned}$$

When *power* is not 2, the ARCH(∞) representation is still used where ϵ_t^2 is replaced by $|\epsilon_t|^p$ and `p` is the power.

Methods

`backcast`(resids)	Construct values for backcasting to start the recursion
`backcast_transform`(backcast)	Transformation to apply to user-provided backcast values
`bounds`(resids)	Returns bounds for parameters
`compute_variance`(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
`constraints`()	Construct parameter constraints arrays for parameter estimation
`forecast`(parameters, resids, backcast, ...)	Forecast volatility from the model
`parameter_names`()	Names of model parameters
`simulate`(parameters, nobs, rng[, burn, ...])	Simulate data from the model
`starting_values`(resids)	Returns starting values for the ARCH model
`update`(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
`variance_bounds`(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.FIGARCH.backcast

FIGARCH.**backcast**(resids: *ndarray*) → float | ndarray

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
backcast – Value to use in backcasting in the volatility recursion

Return type
`float`

arch.univariate.FIGARCH.backcast_transform

FIGARCH.**backcast_transform**(backcast: *float* | *ndarray*) → float | ndarray

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns
backcast – Backcast transformed to the model-appropriate scale

Return type
`{float, ndarray}`

arch.univariate.FIGARCH.bounds

FIGARCH.**bounds**(resids: *ndarray*) → list[tuple[float, float]]

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
bounds – List of bounds where each element is (lower, upper).

Return type
`list[tuple[float,float]]`

arch.univariate.FIGARCH.compute_variance

FIGARCH.**compute_variance**(parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float* | *ndarray*, var_bounds: *ndarray*) → ndarray

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.FIGARCH.constraints

FIGARCH.**constraints**() → tuple[ndarray, ndarray]

Construct parameter constraints arrays for parameter estimation

Returns

- **A** (`numpy.ndarray`) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (`numpy.ndarray`) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

arch.univariate.FIGARCH.forecast

FIGARCH.**forecast**(parameters*: ArrayLike1D*, resids*: Float64Array*, backcast*: Float64Array | float*, var_bounds*: Float64Array*, start*: int | None = None*, horizon*: int = 1*, method*: ForecastingMethod = 'analytic'*, simulations*: int = 1000*, rng*: RNGType | None = None*, random_state*: RandomState | None = None*) → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = None
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = 1
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = 'analytic'
Method to use when producing the forecast. The default is analytic.

simulations: int = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is 'simulation'. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

Returns

forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type

`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported

- ValueError –
  - If the method is not known

Notes

The analytic method is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.FIGARCH.parameter_names

FIGARCH.**parameter_names**() → list[str]

Names of model parameters

Returns
names – Variables names

Return type
list (str)

arch.univariate.FIGARCH.simulate

FIGARCH.**simulate**(parameters: *Sequence[float | int] | ndarray | Series*, nobs: *int*, rng: *Callable[[int | tuple[int, ...]], ndarray]*, burn: *int* = 500, initial_value: *None | float | ndarray* = None) → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (numpy.ndarray) – The simulated residuals
- **variance** (numpy.ndarray) – The simulated variance

arch.univariate.FIGARCH.starting_values

FIGARCH.**starting_values**(resids: *ndarray*) → ndarray

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
`numpy.ndarray`

arch.univariate.FIGARCH.update

FIGARCH.**update**(index: *int*, parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float* | *ndarray*, var_bounds: *ndarray*) → float

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
`float`

arch.univariate.FIGARCH.variance_bounds

FIGARCH.**variance_bounds**(resids: *ndarray*, power: *float* = 2.0) → ndarray

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
truncation	Truncation lag for the ARCH-infinity approximation
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.FIGARCH.name

property FIGARCH.**name** : str

The name of the volatility process

arch.univariate.FIGARCH.num_params

property FIGARCH.**num_params** : int

The number of parameters in the model

arch.univariate.FIGARCH.start

property `FIGARCH.start` : int

Index to use to start variance subarray selection

arch.univariate.FIGARCH.stop

property `FIGARCH.stop` : int

Index to use to stop variance subarray selection

arch.univariate.FIGARCH.truncation

property `FIGARCH.truncation` : int

Truncation lag for the ARCH-infinity approximation

arch.univariate.FIGARCH.updateable

property `FIGARCH.updateable` : bool

Flag indicating that the volatility process supports update

arch.univariate.FIGARCH.volatility_updater

property `FIGARCH.volatility_updater` : VolatilityUpdater

Get the volatility updater associated with the volatility process

Returns
The updater class

Return type
`VolatilityUpdater`

Raises
`NotImplementedError` – If the process is not updateable

1.9.4 arch.univariate.EGARCH

class `arch.univariate.EGARCH`(p: *int* = *1*, o: *int* = *0*, q: *int* = *1*)

EGARCH model estimation

Parameters

p: int = 1
Order of the symmetric innovation

o: int = 0
Order of the asymmetric innovation

q: int = 1
Order of the lagged (transformed) conditional variance

Examples

```
>>> from arch.univariate import EGARCH
```

Symmetric EGARCH(1,1)

```
>>> egarch = EGARCH(p=1, q=1)
```

Standard EGARCH process

```
>>> egarch = EGARCH(p=1, o=1, q=1)
```

Exponential ARCH process

```
>>> earch = EGARCH(p=5)
```

Notes

In this class of processes, the variance dynamics are

$$\ln\sigma_t^2 = \omega + \sum_{i=1}^{p}\alpha_i\left(|e_{t-i}| - \sqrt{2/\pi}\right) + \sum_{j=1}^{o}\gamma_j e_{t-j} + \sum_{k=1}^{q}\beta_k\ln\sigma_{t-k}^2$$

where $e_t = \epsilon_t/\sigma_t$.

Methods

backcast(resids)	Construct values for backcasting to start the recursion
backcast_transform(backcast)	Transformation to apply to user-provided backcast values
bounds(resids)	Returns bounds for parameters
compute_variance(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
constraints()	Construct parameter constraints arrays for parameter estimation
forecast(parameters, resids, backcast, ...)	Forecast volatility from the model
parameter_names()	Names of model parameters
simulate(parameters, nobs, rng[, burn, ...])	Simulate data from the model
starting_values(resids)	Returns starting values for the ARCH model
update(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
variance_bounds(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.EGARCH.backcast

EGARCH.**backcast**(resids: *ndarray*) → float | ndarray

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
: **backcast** – Value to use in backcasting in the volatility recursion

Return type
: `float`

arch.univariate.EGARCH.backcast_transform

`EGARCH.backcast_transform(backcast: float | ndarray) → float | ndarray`

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns
: **backcast** – Backcast transformed to the model-appropriate scale

Return type
: `{float, ndarray}`

arch.univariate.EGARCH.bounds

`EGARCH.bounds(resids: ndarray) → list[tuple[float, float]]`

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
: **bounds** – List of bounds where each element is (lower, upper).

Return type
: `list[tuple[float,float]]`

arch.univariate.EGARCH.compute_variance

`EGARCH.compute_variance(parameters: ndarray, resids: ndarray, sigma2: ndarray, backcast: float | ndarray, var_bounds: ndarray) → ndarray`

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.EGARCH.constraints

`EGARCH.constraints()` → tuple[ndarray, ndarray]

Construct parameter constraints arrays for parameter estimation

Returns

- **A** (`numpy.ndarray`) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (`numpy.ndarray`) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

arch.univariate.EGARCH.forecast

`EGARCH.forecast`(parameters: *ArrayLike1D*, resids: *Float64Array*, backcast: *Float64Array* | *float*, var_bounds: *Float64Array*, start: *int* | *None* = **None**, horizon: *int* = **1**, method: *ForecastingMethod* = `'analytic'`, simulations: *int* = **1000**, rng: *RNGType* | *None* = **None**, random_state: *RandomState* | *None* = **None**) → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = None
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = 1
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = 'analytic'
Method to use when producing the forecast. The default is analytic.

simulations: int = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is 'simulation'. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

Returns
forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type
`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported
- `ValueError` –
  - If the method is not known

Notes

The analytic `method` is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.EGARCH.parameter_names

`EGARCH.parameter_names()` → list[str]

Names of model parameters

Returns
names – Variables names

Return type
`list (str)`

arch.univariate.EGARCH.simulate

`EGARCH.simulate(parameters: Sequence[float | int] | ndarray | Series, nobs: int, rng: Callable[[int | tuple[int, ...]], ndarray], burn: int = 500, initial_value: None | float | ndarray = None)` → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (`numpy.ndarray`) – The simulated residuals
- **variance** (`numpy.ndarray`) – The simulated variance

arch.univariate.EGARCH.starting_values

`EGARCH.starting_values(resids: ndarray) → ndarray`

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
`numpy.ndarray`

arch.univariate.EGARCH.update

`EGARCH.update(index: int, parameters: ndarray, resids: ndarray, sigma2: ndarray, backcast: float | ndarray, var_bounds: ndarray) → float`

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
`float`

arch.univariate.EGARCH.variance_bounds

`EGARCH.variance_bounds(resids: ndarray, power: float = 2.0) → ndarray`

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.EGARCH.name

`property EGARCH.name : str`

The name of the volatility process

arch.univariate.EGARCH.num_params

`property EGARCH.num_params : int`

The number of parameters in the model

arch.univariate.EGARCH.start

property EGARCH.**start** : int

Index to use to start variance subarray selection

arch.univariate.EGARCH.stop

property EGARCH.**stop** : int

Index to use to stop variance subarray selection

arch.univariate.EGARCH.updateable

property EGARCH.**updateable** : bool

Flag indicating that the volatility process supports update

arch.univariate.EGARCH.volatility_updater

property EGARCH.**volatility_updater** : VolatilityUpdater

Get the volatility updater associated with the volatility process

Returns
The updater class

Return type
`VolatilityUpdater`

Raises
`NotImplementedError` – If the process is not updateable

1.9.5 arch.univariate.HARCH

class arch.univariate.**HARCH**(*lags: int | Sequence[int] = 1*)

Heterogeneous ARCH process

Parameters

lags: int | Sequence[int] = 1
List of lags to include in the model, or if scalar, includes all lags up the value

Examples

```
>>> from arch.univariate import HARCH
```

Lag-1 HARCH, which is identical to an ARCH(1)

```
>>> harch = HARCH()
```

More useful and realistic lag lengths

```
>>> harch = HARCH(lags=[1, 5, 22])
```

Notes

In a Heterogeneous ARCH process, variance dynamics are

$$\sigma_t^2 = \omega + \sum_{i=1}^{m} \alpha_{l_i} \left(l_i^{-1} \sum_{j=1}^{l_i} \epsilon_{t-j}^2 \right)$$

In the common case where `lags=[1,5,22]`, the model is

$$\sigma_t^2 = \omega + \alpha_1 \epsilon_{t-1}^2 + \alpha_5 \left(\frac{1}{5} \sum_{j=1}^{5} \epsilon_{t-j}^2 \right) + \alpha_{22} \left(\frac{1}{22} \sum_{j=1}^{22} \epsilon_{t-j}^2 \right)$$

A HARCH process is a special case of an ARCH process where parameters in the more general ARCH process have been restricted.

Methods

backcast(resids)	Construct values for backcasting to start the recursion
backcast_transform(backcast)	Transformation to apply to user-provided backcast values
bounds(resids)	Returns bounds for parameters
compute_variance(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
constraints()	Construct parameter constraints arrays for parameter estimation
forecast(parameters, resids, backcast, ...)	Forecast volatility from the model
parameter_names()	Names of model parameters
simulate(parameters, nobs, rng[, burn, ...])	Simulate data from the model
starting_values(resids)	Returns starting values for the ARCH model
update(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
variance_bounds(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.HARCH.backcast

HARCH.**backcast**(resids: *ndarray*) → float | ndarray

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
backcast – Value to use in backcasting in the volatility recursion

Return type
`float`

arch.univariate.HARCH.backcast_transform

HARCH.**backcast_transform**(backcast: *float* | *ndarray*) → float | ndarray

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns
backcast – Backcast transformed to the model-appropriate scale

Return type
`{float, ndarray}`

arch.univariate.HARCH.bounds

HARCH.**bounds**(resids: *ndarray*) → list[tuple[float, float]]

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
bounds – List of bounds where each element is (lower, upper).

Return type
`list[tuple[float,float]]`

arch.univariate.HARCH.compute_variance

HARCH.**compute_variance**(parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float* | *ndarray*, var_bounds: *ndarray*) → ndarray

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.HARCH.constraints

`HARCH.constraints()` → tuple[ndarray, ndarray]

Construct parameter constraints arrays for parameter estimation

Returns

- **A** (`numpy.ndarray`) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (`numpy.ndarray`) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

arch.univariate.HARCH.forecast

`HARCH.forecast(parameters: ArrayLike1D, resids: Float64Array, backcast: Float64Array | float, var_bounds: Float64Array, start: int | None = None, horizon: int = 1, method: ForecastingMethod = 'analytic', simulations: int = 1000, rng: RNGType | None = None, random_state: RandomState | None = None)` → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = None
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = 1
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = 'analytic'
Method to use when producing the forecast. The default is analytic.

simulations: int = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is ‘simulation’. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is ‘bootstrap’

Returns
forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type
`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported
- `ValueError` –
  - If the method is not known

Notes

The analytic `method` is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.HARCH.parameter_names

`HARCH.parameter_names()` → list[str]

Names of model parameters

Returns
names – Variables names

Return type
`list (str)`

arch.univariate.HARCH.simulate

`HARCH.simulate(parameters: Sequence[float | int] | ndarray | Series, nobs: int, rng: Callable[[int | tuple[int, ...]], ndarray], burn: int = 500, initial_value: None | float | ndarray = None)` → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (numpy.ndarray) – The simulated residuals
- **variance** (numpy.ndarray) – The simulated variance

arch.univariate.HARCH.starting_values

HARCH.**starting_values**(resids: *ndarray*) → ndarray

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
numpy.ndarray

arch.univariate.HARCH.update

HARCH.**update**(index: *int*, parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float* | *ndarray*, var_bounds: *ndarray*) → float

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
float

arch.univariate.HARCH.variance_bounds

HARCH.**variance_bounds**(resids: *ndarray*, power: *float* = 2.0) → ndarray

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.HARCH.name

property HARCH.**name** : str

The name of the volatility process

arch.univariate.HARCH.num_params

property HARCH.**num_params** : int

The number of parameters in the model

arch.univariate.HARCH.start

property HARCH.**start** : int

Index to use to start variance subarray selection

arch.univariate.HARCH.stop

property HARCH.**stop** : int

Index to use to stop variance subarray selection

arch.univariate.HARCH.updateable

property HARCH.**updateable** : bool

Flag indicating that the volatility process supports update

arch.univariate.HARCH.volatility_updater

property HARCH.**volatility_updater** : VolatilityUpdater

Get the volatility updater associated with the volatility process

Returns
The updater class

Return type
`VolatilityUpdater`

Raises
`NotImplementedError` – If the process is not updateable

1.9.6 arch.univariate.MIDASHyperbolic

class arch.univariate.**MIDASHyperbolic**(m: *int* = *22*, asym: *bool* = *False*)

MIDAS Hyperbolic ARCH process

Parameters

m: int = 22
Length of maximum lag to include in the model

asym: bool = False
Flag indicating whether to include an asymmetric term

Examples

```
>>> from arch.univariate import MIDASHyperbolic
```

22-lag MIDAS Hyperbolic process

```
>>> harch = MIDASHyperbolic()
```

Longer 66-period lag

```
>>> harch = MIDASHyperbolic(m=66)
```

Asymmetric MIDAS Hyperbolic process

```
>>> harch = MIDASHyperbolic(asym=True)
```

Notes

In a MIDAS Hyperbolic process, the variance evolves according to

$$\sigma_t^2 = \omega + \sum_{i=1}^{m} \left(\alpha + \gamma I\left[\epsilon_{t-j} < 0\right]\right) \phi_i(\theta)\epsilon_{t-i}^2$$

where

$$\phi_i(\theta) \propto \Gamma(i+\theta)/(\Gamma(i+1)\Gamma(\theta))$$

where Γ is the gamma function. $\{\phi_i(\theta)\}$ is normalized so that $\sum \phi_i(\theta) = 1$

References

Methods

backcast(resids)	Construct values for backcasting to start the recursion
backcast_transform(backcast)	Transformation to apply to user-provided backcast values
bounds(resids)	Returns bounds for parameters
compute_variance(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
constraints()	Constraints
forecast(parameters, resids, backcast, ...)	Forecast volatility from the model
parameter_names()	Names of model parameters
simulate(parameters, nobs, rng[, burn, ...])	Simulate data from the model
starting_values(resids)	Returns starting values for the ARCH model
update(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
variance_bounds(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.MIDASHyperbolic.backcast

MIDASHyperbolic.**backcast**(resids: *ndarray*) → float | ndarray

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns

backcast – Value to use in backcasting in the volatility recursion

Return type

float

arch.univariate.MIDASHyperbolic.backcast_transform

`MIDASHyperbolic.backcast_transform(backcast: float | ndarray) → float | ndarray`

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns
backcast – Backcast transformed to the model-appropriate scale

Return type
`{float, ndarray}`

arch.univariate.MIDASHyperbolic.bounds

`MIDASHyperbolic.bounds(resids: ndarray) → list[tuple[float, float]]`

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
bounds – List of bounds where each element is (lower, upper).

Return type
`list[tuple[float,float]]`

arch.univariate.MIDASHyperbolic.compute_variance

`MIDASHyperbolic.compute_variance(parameters: ndarray, resids: ndarray, sigma2: ndarray, backcast: float | ndarray, var_bounds: ndarray) → ndarray`

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.MIDASHyperbolic.constraints

MIDASHyperbolic.**constraints**() → tuple[ndarray, ndarray]

Constraints

Notes

Parameters are (omega, alpha, gamma, theta)

A.dot(parameters) - b >= 0

1. omega >0
2. alpha>0 or alpha + gamma > 0
3. alpha<1 or alpha+0.5*gamma<1
4. theta > 0
5. theta < 1

arch.univariate.MIDASHyperbolic.forecast

MIDASHyperbolic.**forecast**(parameters: *ArrayLike1D*, resids: *Float64Array*, backcast: *Float64Array* | *float*, var_bounds: *Float64Array*, start: *int* | *None* = **None**, horizon: *int* = **1**, method: *ForecastingMethod* = 'analytic', simulations: *int* = **1000**, rng: *RNGType* | *None* = **None**, random_state: *RandomState* | *None* = **None**) → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = None
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = 1
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = 'analytic'
Method to use when producing the forecast. The default is analytic.

simulations: int = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is 'simulation'. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

Returns
forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type
`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported
- `ValueError` –
  - If the method is not known

Notes

The analytic `method` is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.MIDASHyperbolic.parameter_names

`MIDASHyperbolic.parameter_names()` → list[str]

Names of model parameters

Returns
names – Variables names

Return type
`list (str)`

arch.univariate.MIDASHyperbolic.simulate

`MIDASHyperbolic.simulate(parameters: Sequence[float | int] | ndarray | Series, nobs: int, rng: Callable[[int | tuple[int, ...]], ndarray], burn: int = 500, initial_value: None | float | ndarray = None)` → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (`numpy.ndarray`) – The simulated residuals
- **variance** (`numpy.ndarray`) – The simulated variance

arch.univariate.MIDASHyperbolic.starting_values

`MIDASHyperbolic.starting_values(resids: ndarray) → ndarray`

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
`numpy.ndarray`

arch.univariate.MIDASHyperbolic.update

`MIDASHyperbolic.update(index: int, parameters: ndarray, resids: ndarray, sigma2: ndarray, backcast: float | ndarray, var_bounds: ndarray) → float`

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
`float`

arch.univariate.MIDASHyperbolic.variance_bounds

`MIDASHyperbolic.variance_bounds(resids: ndarray, power: float = 2.0) → ndarray`

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.MIDASHyperbolic.name

`property MIDASHyperbolic.name : str`

The name of the volatility process

arch.univariate.MIDASHyperbolic.num_params

`property MIDASHyperbolic.num_params : int`

The number of parameters in the model

arch.univariate.MIDASHyperbolic.start

property `MIDASHyperbolic.start` : int

Index to use to start variance subarray selection

arch.univariate.MIDASHyperbolic.stop

property `MIDASHyperbolic.stop` : int

Index to use to stop variance subarray selection

arch.univariate.MIDASHyperbolic.updateable

property `MIDASHyperbolic.updateable` : bool

Flag indicating that the volatility process supports update

arch.univariate.MIDASHyperbolic.volatility_updater

property `MIDASHyperbolic.volatility_updater` : VolatilityUpdater

Get the volatility updater associated with the volatility process

Returns

The updater class

Return type

`VolatilityUpdater`

Raises

`NotImplementedError` – If the process is not updateable

1.9.7 arch.univariate.ARCH

class `arch.univariate.ARCH(p: int = 1)`

ARCH process

Parameters

p: int = 1

Order of the symmetric innovation

Examples

ARCH(1) process

```
>>> from arch.univariate import ARCH
```

ARCH(5) process

```
>>> arch = ARCH(p=5)
```

Notes

The variance dynamics of the model estimated

$$\sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i \epsilon_{t-i}^2$$

Methods

`backcast`(resids)	Construct values for backcasting to start the recursion
`backcast_transform`(backcast)	Transformation to apply to user-provided backcast values
`bounds`(resids)	Returns bounds for parameters
`compute_variance`(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
`constraints`()	Construct parameter constraints arrays for parameter estimation
`forecast`(parameters, resids, backcast, ...)	Forecast volatility from the model
`parameter_names`()	Names of model parameters
`simulate`(parameters, nobs, rng[, burn, ...])	Simulate data from the model
`starting_values`(resids)	Returns starting values for the ARCH model
`update`(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
`variance_bounds`(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.ARCH.backcast

`ARCH.backcast(resids: ndarray) → float | ndarray`

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
backcast – Value to use in backcasting in the volatility recursion

Return type
`float`

arch.univariate.ARCH.backcast_transform

`ARCH.backcast_transform(backcast: float | ndarray) → float | ndarray`

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns
backcast – Backcast transformed to the model-appropriate scale

Return type
`{float, ndarray}`

arch.univariate.ARCH.bounds

ARCH.**bounds**(resids: *ndarray*) → list[tuple[float, float]]

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
bounds – List of bounds where each element is (lower, upper).

Return type
`list[tuple[float,float]]`

arch.univariate.ARCH.compute_variance

ARCH.**compute_variance**(parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float* | *ndarray*, var_bounds: *ndarray*) → ndarray

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.ARCH.constraints

ARCH.**constraints**() → tuple[ndarray, ndarray]

Construct parameter constraints arrays for parameter estimation

Returns

- **A** (`numpy.ndarray`) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (`numpy.ndarray`) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

arch.univariate.ARCH.forecast

ARCH.**forecast**(parameters: *ArrayLike1D*, resids: *Float64Array*, backcast: *Float64Array* | *float*, var_bounds: *Float64Array*, start: *int* | *None* = **None**, horizon: *int* = **1**, method: *ForecastingMethod* = `'analytic'`, simulations: *int* = **1000**, rng: *RNGType* | *None* = **None**, random_state: *RandomState* | *None* = **None**) → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = None
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = 1
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = 'analytic'
Method to use when producing the forecast. The default is analytic.

simulations: int = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is 'simulation'. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

Returns
forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type
`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported

- ValueError –
  - If the method is not known

Notes

The analytic method is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.ARCH.parameter_names

ARCH.**parameter_names**() → list[str]

Names of model parameters

Returns
names – Variables names

Return type
list (str)

arch.univariate.ARCH.simulate

ARCH.**simulate**(parameters: *Sequence[float | int] | ndarray | Series*, nobs: *int*, rng: *Callable[[int | tuple[int, ...]], ndarray]*, burn: *int* = 500, initial_value: *None | float | ndarray* = None) → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (numpy.ndarray) – The simulated residuals
- **variance** (numpy.ndarray) – The simulated variance

arch.univariate.ARCH.starting_values

ARCH.**starting_values**(resids: *ndarray*) → ndarray

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
`numpy.ndarray`

arch.univariate.ARCH.update

ARCH.**update**(index: *int*, parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float* | *ndarray*, var_bounds: *ndarray*) → float

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
`float`

arch.univariate.ARCH.variance_bounds

ARCH.**variance_bounds**(resids: *ndarray*, power: *float* = *2.0*) → ndarray

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.ARCH.name

property ARCH.**name** : str

The name of the volatility process

arch.univariate.ARCH.num_params

property ARCH.**num_params** : int

The number of parameters in the model

arch.univariate.ARCH.start

property `ARCH.start` : int

Index to use to start variance subarray selection

arch.univariate.ARCH.stop

property `ARCH.stop` : int

Index to use to stop variance subarray selection

arch.univariate.ARCH.updateable

property `ARCH.updateable` : bool

Flag indicating that the volatility process supports update

arch.univariate.ARCH.volatility_updater

property `ARCH.volatility_updater` : VolatilityUpdater

Get the volatility updater associated with the volatility process

Returns

The updater class

Return type

`VolatilityUpdater`

Raises

`NotImplementedError` – If the process is not updateable

1.9.8 arch.univariate.APARCH

class `arch.univariate.APARCH`(p: *int* = *1*, o: *int* = *1*, q: *int* = *1*, delta: *float* | *None* = *None*, common_asym: *bool* = *False*)

Asymmetric Power ARCH (APARCH) volatility process

Parameters

p: int = 1

Order of the symmetric innovation. Must satisfy p>=o.

o: int = 1

Order of the asymmetric innovation. Must satisfy o<=p.

q: int = 1

Order of the lagged (transformed) conditional variance

delta: float | None = None

Value to use for a fixed delta in the APARCH model. If not provided, the value of delta is jointly estimated with other model parameters. User provided delta is restricted to lie in (0.05, 4.0).

common_asym: bool = False

Restrict all asymmetry terms to share the same asymmetry parameter. If False (default), then there are no restrictions on the `o` asymmetry parameters.

Examples

```
>>> from arch.univariate import APARCH
```

Symmetric Power ARCH(1,1)

```
>>> aparch = APARCH(p=1, q=1)
```

Standard APARCH process

```
>>> aparch = APARCH(p=1, o=1, q=1)
```

Fixed power parameters

```
>>> aparch = APARCH(p=1, o=1, q=1, delta=1.3)
```

Notes

In this class of processes, the variance dynamics are

$$\sigma_t^\delta = \omega + \sum_{i=1}^{p} \alpha_i \left(|\epsilon_{t-i}| - \gamma_i I_{[o\geq i]} \epsilon_{t-i}\right)^\delta + \sum_{k=1}^{q} \beta_k \sigma_{t-k}^\delta$$

If `common_asym` is `True`, then all of γ_i are restricted to have a common value.

Methods

backcast(resids)	Construct values for backcasting to start the recursion
backcast_transform(backcast)	Transformation to apply to user-provided backcast values
bounds(resids)	Returns bounds for parameters
compute_variance(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
constraints()	Construct parameter constraints arrays for parameter estimation
forecast(parameters, resids, backcast, ...)	Forecast volatility from the model
parameter_names()	Names of model parameters
simulate(parameters, nobs, rng[, burn, ...])	Simulate data from the model
starting_values(resids)	Returns starting values for the ARCH model
update(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
variance_bounds(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.APARCH.backcast

APARCH.**backcast**(resids: *ndarray*) → float | ndarray

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns

backcast – Value to use in backcasting in the volatility recursion

Return type

`float`

arch.univariate.APARCH.backcast_transform

`APARCH.backcast_transform(backcast: float | ndarray) → float | ndarray`

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns

backcast – Backcast transformed to the model-appropriate scale

Return type

`{float, ndarray}`

arch.univariate.APARCH.bounds

`APARCH.bounds(resids: ndarray) → list[tuple[float, float]]`

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns

bounds – List of bounds where each element is (lower, upper).

Return type

`list[tuple[float,float]]`

arch.univariate.APARCH.compute_variance

`APARCH.compute_variance(parameters: ndarray, resids: ndarray, sigma2: ndarray, backcast: float | ndarray, var_bounds: ndarray) → ndarray`

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.APARCH.constraints

`APARCH.constraints()` → tuple[ndarray, ndarray]

Construct parameter constraints arrays for parameter estimation

Returns

- **A** (`numpy.ndarray`) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (`numpy.ndarray`) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

arch.univariate.APARCH.forecast

`APARCH.forecast`(parameters: *ArrayLike1D*, resids: *Float64Array*, backcast: *Float64Array* | *float*, var_bounds: *Float64Array*, start: *int* | *None* = `None`, horizon: *int* = `1`, method: *ForecastingMethod* = `'analytic'`, simulations: *int* = `1000`, rng: *RNGType* | *None* = `None`, random_state: *RandomState* | *None* = `None`) → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = `None`
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = `1`
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = `'analytic'`
Method to use when producing the forecast. The default is analytic.

simulations: int = `1000`
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is 'simulation'. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

Returns
forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type
`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported
- `ValueError` –
  - If the method is not known

Notes

The analytic `method` is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.APARCH.parameter_names

`APARCH.parameter_names()` → list[str]

Names of model parameters

Returns
names – Variables names

Return type
`list (str)`

arch.univariate.APARCH.simulate

`APARCH.simulate(parameters: Sequence[float | int] | ndarray | Series, nobs: int, rng: Callable[[int | tuple[int, ...]], ndarray], burn: int = 500, initial_value: None | float | ndarray = None)` → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (`numpy.ndarray`) – The simulated residuals
- **variance** (`numpy.ndarray`) – The simulated variance

arch.univariate.APARCH.starting_values

`APARCH.starting_values(resids: ndarray) → ndarray`

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
`numpy.ndarray`

arch.univariate.APARCH.update

`APARCH.update(index: int, parameters: ndarray, resids: ndarray, sigma2: ndarray, backcast: float | ndarray, var_bounds: ndarray) → float`

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
`float`

arch.univariate.APARCH.variance_bounds

`APARCH.variance_bounds(resids: ndarray, power: float = 2.0) → ndarray`

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

common_asym	The value of delta in the model.
delta	The value of delta in the model.
name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.APARCH.common_asym

`property APARCH.common_asym : bool`

The value of delta in the model. NaN is delta is estimated.

arch.univariate.APARCH.delta

`property APARCH.delta : float`

The value of delta in the model. NaN is delta is estimated.

arch.univariate.APARCH.name

property APARCH.**name** : str

The name of the volatility process

arch.univariate.APARCH.num_params

property APARCH.**num_params** : int

The number of parameters in the model

arch.univariate.APARCH.start

property APARCH.**start** : int

Index to use to start variance subarray selection

arch.univariate.APARCH.stop

property APARCH.**stop** : int

Index to use to stop variance subarray selection

arch.univariate.APARCH.updateable

property APARCH.**updateable** : bool

Flag indicating that the volatility process supports update

arch.univariate.APARCH.volatility_updater

property APARCH.**volatility_updater** : VolatilityUpdater

Get the volatility updater associated with the volatility process

Returns
The updater class

Return type
`VolatilityUpdater`

Raises
`NotImplementedError` – If the process is not updateable

1.9.9 Parameterless Variance Processes

Some volatility processes use fixed parameters and so have no parameters that are estimable.

EWMAVariance([lam])	Exponentially Weighted Moving-Average (RiskMetrics) Variance process
RiskMetrics2006([tau0, tau1, kmax, rho])	RiskMetrics 2006 Variance process

arch.univariate.EWMAVariance

class arch.univariate.**EWMAVariance**(lam: *float* | *None* = 0.94)

Exponentially Weighted Moving-Average (RiskMetrics) Variance process

Parameters

lam: float | None = 0.94
Smoothing parameter. Default is 0.94. Set to None to estimate lam jointly with other model parameters

Examples

Daily RiskMetrics EWMA process

```
>>> from arch.univariate import EWMAVariance
>>> rm = EWMAVariance(0.94)
```

Notes

The variance dynamics of the model

$$\sigma_t^2 = \lambda\sigma_{t-1}^2 + (1-\lambda)\epsilon_{t-1}^2$$

When lam is provided, this model has no parameters since the smoothing parameter is treated as fixed. Set lam to `None` to jointly estimate this parameter when fitting the model.

Methods

backcast(resids)	Construct values for backcasting to start the recursion
backcast_transform(backcast)	Transformation to apply to user-provided backcast values
bounds(resids)	Returns bounds for parameters
compute_variance(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
constraints()	Construct parameter constraints arrays for parameter estimation
forecast(parameters, resids, backcast, ...)	Forecast volatility from the model
parameter_names()	Names of model parameters
simulate(parameters, nobs, rng[, burn, ...])	Simulate data from the model
starting_values(resids)	Returns starting values for the ARCH model
update(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
variance_bounds(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.EWMAVariance.backcast

`EWMAVariance.backcast(resids: ndarray) → float | ndarray`

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
backcast – Value to use in backcasting in the volatility recursion

Return type
`float`

arch.univariate.EWMAVariance.backcast_transform

`EWMAVariance.backcast_transform(backcast: float | ndarray) → float | ndarray`

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns
backcast – Backcast transformed to the model-appropriate scale

Return type
`{float, ndarray}`

arch.univariate.EWMAVariance.bounds

`EWMAVariance.bounds(resids: ndarray) → list[tuple[float, float]]`

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
bounds – List of bounds where each element is (lower, upper).

Return type
`list[tuple[float,float]]`

arch.univariate.EWMAVariance.compute_variance

EWMAVariance.**compute_variance**(parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float* | *ndarray*, var_bounds: *ndarray*) → ndarray

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.EWMAVariance.constraints

EWMAVariance.**constraints**() → tuple[ndarray, ndarray]

Construct parameter constraints arrays for parameter estimation

Returns

- **A** (`numpy.ndarray`) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (`numpy.ndarray`) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

arch.univariate.EWMAVariance.forecast

EWMAVariance.**forecast**(parameters: *ArrayLike1D*, resids: *Float64Array*, backcast: *Float64Array* | *float*, var_bounds: *Float64Array*, start: *int* | *None* = **None**, horizon: *int* = **1**, method: *ForecastingMethod* = 'analytic', simulations: *int* = **1000**, rng: *RNGType* | *None* = **None**, random_state: *RandomState* | *None* = **None**) → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = None
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = 1
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = 'analytic'
Method to use when producing the forecast. The default is analytic.

simulations: int = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is 'simulation'. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

Returns
forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type
`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported
- `ValueError` –
  - If the method is not known

Notes

The analytic `method` is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.EWMAVariance.parameter_names

`EWMAVariance.parameter_names()` → list[str]

Names of model parameters

Returns
names – Variables names

Return type
`list (str)`

arch.univariate.EWMAVariance.simulate

EWMAVariance.**simulate**(parameters: *Sequence[float | int] | ndarray | Series*, nobs: *int*, rng: *Callable[[int | tuple[int, ...]], ndarray]*, burn: *int* = 500, initial_value: *None | float | ndarray* = *None*) → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (`numpy.ndarray`) – The simulated residuals
- **variance** (`numpy.ndarray`) – The simulated variance

arch.univariate.EWMAVariance.starting_values

EWMAVariance.**starting_values**(resids: *ndarray*) → ndarray

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
`numpy.ndarray`

arch.univariate.EWMAVariance.update

EWMAVariance.**update**(index: *int*, parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float | ndarray*, var_bounds: *ndarray*) → float

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
`float`

arch.univariate.EWMAVariance.variance_bounds

`EWMAVariance.variance_bounds(resids: ndarray, power: float = 2.0) → ndarray`

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.EWMAVariance.name

property `EWMAVariance.`**`name`** : str

The name of the volatility process

arch.univariate.EWMAVariance.num_params

property `EWMAVariance.`**`num_params`** : int

The number of parameters in the model

arch.univariate.EWMAVariance.start

property `EWMAVariance.`**`start`** : int

Index to use to start variance subarray selection

arch.univariate.EWMAVariance.stop

property `EWMAVariance.`**`stop`** : int

Index to use to stop variance subarray selection

arch.univariate.EWMAVariance.updateable

property `EWMAVariance.`**`updateable`** : bool

Flag indicating that the volatility process supports update

arch.univariate.EWMAVariance.volatility_updater

property `EWMAVariance.`**`volatility_updater`** : VolatilityUpdater

Get the volatility updater associated with the volatility process

Returns
The updater class

Return type
`VolatilityUpdater`

Raises
`NotImplementedError` – If the process is not updateable

arch.univariate.RiskMetrics2006

class arch.univariate.**RiskMetrics2006**(tau0: *float* = ***1560***, tau1: *float* = ***4***, kmax: *int* = ***14***, rho: *float* = ***1.4142135623730951***)

RiskMetrics 2006 Variance process

Parameters

tau0: float = 1560
Length of long cycle. Default is 1560.

tau1: float = 4
Length of short cycle. Default is 4.

kmax: int = 14
Number of components. Default is 14.

rho: float = 1.4142135623730951
Relative scale of adjacent cycles. Default is sqrt(2)

Examples

Daily RiskMetrics 2006 process

```
>>> from arch.univariate import RiskMetrics2006
>>> rm = RiskMetrics2006()
```

Notes

The variance dynamics of the model are given as a weighted average of kmax EWMA variance processes where the smoothing parameters and weights are determined by tau0, tau1 and rho.

This model has no parameters since the smoothing parameter is fixed.

Methods

backcast(resids)	Construct values for backcasting to start the recursion
backcast_transform(backcast)	Transformation to apply to user-provided backcast values
bounds(resids)	Returns bounds for parameters
compute_variance(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
constraints()	Construct parameter constraints arrays for parameter estimation
forecast(parameters, resids, backcast, ...)	Forecast volatility from the model
parameter_names()	Names of model parameters
simulate(parameters, nobs, rng[, burn, ...])	Simulate data from the model
starting_values(resids)	Returns starting values for the ARCH model
update(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
variance_bounds(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.RiskMetrics2006.backcast

RiskMetrics2006.**backcast**(resids: *ndarray*) → float | ndarray

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
backcast – Backcast values for each EWMA component

Return type
`numpy.ndarray`

arch.univariate.RiskMetrics2006.backcast_transform

RiskMetrics2006.**backcast_transform**(backcast: *float* | *ndarray*) → float | ndarray

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns
backcast – Backcast transformed to the model-appropriate scale

Return type
`{float, ndarray}`

arch.univariate.RiskMetrics2006.bounds

RiskMetrics2006.**bounds**(resids: *ndarray*) → list[tuple[float, float]]

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
bounds – List of bounds where each element is (lower, upper).

Return type
`list[tuple[float,float]]`

arch.univariate.RiskMetrics2006.compute_variance

RiskMetrics2006.**compute_variance**(parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float* | *ndarray*, var_bounds: *ndarray*) → ndarray

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.RiskMetrics2006.constraints

RiskMetrics2006.**constraints**() → tuple[ndarray, ndarray]

Construct parameter constraints arrays for parameter estimation

Returns

- **A** (`numpy.ndarray`) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (`numpy.ndarray`) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

arch.univariate.RiskMetrics2006.forecast

RiskMetrics2006.**forecast**(parameters: *ArrayLike1D*, resids: *Float64Array*, backcast: *Float64Array* | *float*, var_bounds: *Float64Array*, start: *int* | *None* = **None**, horizon: *int* = **1**, method: *ForecastingMethod* = `'analytic'`, simulations: *int* = **1000**, rng: *RNGType* | *None* = **None**, random_state: *RandomState* | *None* = **None**) → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = None
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = 1
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = `'analytic'`
Method to use when producing the forecast. The default is analytic.

simulations: int = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is 'simulation'. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

Returns
forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type
`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported
- `ValueError` –
  - If the method is not known

Notes

The analytic `method` is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.RiskMetrics2006.parameter_names

`RiskMetrics2006.parameter_names()` → list[str]

Names of model parameters

Returns
names – Variables names

Return type
`list (str)`

arch.univariate.RiskMetrics2006.simulate

`RiskMetrics2006.simulate(parameters: Sequence[float | int] | ndarray | Series, nobs: int, rng: Callable[[int | tuple[int, ...]], ndarray], burn: int = 500, initial_value: None | float | ndarray = None)` → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (`numpy.ndarray`) – The simulated residuals
- **variance** (`numpy.ndarray`) – The simulated variance

arch.univariate.RiskMetrics2006.starting_values

`RiskMetrics2006.starting_values(resids: ndarray)` → ndarray

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
`numpy.ndarray`

arch.univariate.RiskMetrics2006.update

RiskMetrics2006.**update**(index: *int*, parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float* | *ndarray*, var_bounds: *ndarray*) → float

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
`float`

arch.univariate.RiskMetrics2006.variance_bounds

RiskMetrics2006.**variance_bounds**(resids: *ndarray*, power: *float* = 2.0) → ndarray

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.RiskMetrics2006.name

property `RiskMetrics2006.`**`name`** : str

The name of the volatility process

arch.univariate.RiskMetrics2006.num_params

property `RiskMetrics2006.`**`num_params`** : int

The number of parameters in the model

arch.univariate.RiskMetrics2006.start

property `RiskMetrics2006.`**`start`** : int

Index to use to start variance subarray selection

arch.univariate.RiskMetrics2006.stop

property `RiskMetrics2006.`**`stop`** : int

Index to use to stop variance subarray selection

arch.univariate.RiskMetrics2006.updateable

property `RiskMetrics2006.`**`updateable`** : bool

Flag indicating that the volatility process supports update

arch.univariate.RiskMetrics2006.volatility_updater

property `RiskMetrics2006.`**`volatility_updater`** : VolatilityUpdater

Get the volatility updater associated with the volatility process

Returns

The updater class

Return type

`VolatilityUpdater`

Raises

`NotImplementedError` – If the process is not updateable

1.9.10 FixedVariance

The `FixedVariance` class is a special-purpose volatility process that allows the so-called zig-zag algorithm to be used. See the example for usage.

FixedVariance(variance[, unit_scale])	Fixed volatility process

arch.univariate.FixedVariance

class arch.univariate.**FixedVariance**(variance: *ndarray*, unit_scale: *bool* = ***False***)

Fixed volatility process

Parameters

variance: ndarray
Array containing the variances to use. Should have the same shape as the data used in the model.

unit_scale: bool = False
Flag whether to enforce a unit scale. If False, a scale parameter will be estimated so that the model variance will be proportional to `variance`. If True, the model variance is set of `variance`

Notes

Allows a fixed set of variances to be used when estimating a mean model, allowing GLS estimation.

Methods

backcast(resids)	Construct values for backcasting to start the recursion
backcast_transform(backcast)	Transformation to apply to user-provided backcast values
bounds(resids)	Returns bounds for parameters
compute_variance(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
constraints()	Construct parameter constraints arrays for parameter estimation
forecast(parameters, resids, backcast, ...)	Forecast volatility from the model
parameter_names()	Names of model parameters
simulate(parameters, nobs, rng[, burn, ...])	Simulate data from the model
starting_values(resids)	Returns starting values for the ARCH model
update(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
variance_bounds(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.FixedVariance.backcast

`FixedVariance.backcast(resids: ndarray) → float | ndarray`

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
backcast – Value to use in backcasting in the volatility recursion

Return type
`float`

arch.univariate.FixedVariance.backcast_transform

`FixedVariance.backcast_transform(backcast: float | ndarray) → float | ndarray`

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns
backcast – Backcast transformed to the model-appropriate scale

Return type
`{float, ndarray}`

arch.univariate.FixedVariance.bounds

`FixedVariance.bounds(resids: ndarray) → list[tuple[float, float]]`

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
bounds – List of bounds where each element is (lower, upper).

Return type
`list[tuple[float,float]]`

arch.univariate.FixedVariance.compute_variance

FixedVariance.**compute_variance**(parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float* | *ndarray*, var_bounds: *ndarray*) → ndarray

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.FixedVariance.constraints

FixedVariance.**constraints**() → tuple[ndarray, ndarray]

Construct parameter constraints arrays for parameter estimation

Returns

- **A** (`numpy.ndarray`) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (`numpy.ndarray`) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

arch.univariate.FixedVariance.forecast

FixedVariance.**forecast**(parameters: *ArrayLike1D*, resids: *Float64Array*, backcast: *Float64Array* | *float*, var_bounds: *Float64Array*, start: *int* | *None* = **None**, horizon: *int* = **1**, method: *ForecastingMethod* = 'analytic', simulations: *int* = **1000**, rng: *RNGType* | *None* = **None**, random_state: *RandomState* | *None* = **None**) → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = None
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = 1
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = 'analytic'
Method to use when producing the forecast. The default is analytic.

simulations: int = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is 'simulation'. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

Returns
forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type
`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported
- `ValueError` –
  - If the method is not known

Notes

The analytic `method` is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.FixedVariance.parameter_names

`FixedVariance.parameter_names()` → list[str]

Names of model parameters

Returns
names – Variables names

Return type
`list (str)`

arch.univariate.FixedVariance.simulate

FixedVariance.**simulate**(parameters: *Sequence[float | int] | ndarray | Series*, nobs: *int*, rng: *Callable[[int | tuple[int, ...]], ndarray]*, burn: *int* = **500**, initial_value: *None | float | ndarray* = ***None***) → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (`numpy.ndarray`) – The simulated residuals
- **variance** (`numpy.ndarray`) – The simulated variance

arch.univariate.FixedVariance.starting_values

FixedVariance.**starting_values**(resids: *ndarray*) → ndarray

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
`numpy.ndarray`

arch.univariate.FixedVariance.update

FixedVariance.**update**(index: *int*, parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, backcast: *float | ndarray*, var_bounds: *ndarray*) → float

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
`float`

arch.univariate.FixedVariance.variance_bounds

`FixedVariance.variance_bounds(resids: ndarray, power: float = 2.0) → ndarray`

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.FixedVariance.name

property FixedVariance.**name** : str

The name of the volatility process

arch.univariate.FixedVariance.num_params

property FixedVariance.**num_params** : int

The number of parameters in the model

arch.univariate.FixedVariance.start

property FixedVariance.**start** : int

Index to use to start variance subarray selection

arch.univariate.FixedVariance.stop

property FixedVariance.**stop** : int

Index to use to stop variance subarray selection

arch.univariate.FixedVariance.updateable

property FixedVariance.**updateable** : bool

Flag indicating that the volatility process supports update

arch.univariate.FixedVariance.volatility_updater

property FixedVariance.**volatility_updater** : VolatilityUpdater

Get the volatility updater associated with the volatility process

Returns
The updater class

Return type
`VolatilityUpdater`

Raises
`NotImplementedError` – If the process is not updateable

1.9.11 Writing New Volatility Processes

All volatility processes must inherit from *VolatilityProcess* and provide all public methods.

VolatilityProcess()	Abstract base class for ARCH models.

arch.univariate.volatility.VolatilityProcess

class arch.univariate.volatility.**VolatilityProcess**

Abstract base class for ARCH models. Allows the conditional mean model to be specified separately from the conditional variance, even though parameters are estimated jointly.

Methods

backcast(resids)	Construct values for backcasting to start the recursion
backcast_transform(backcast)	Transformation to apply to user-provided backcast values
bounds(resids)	Returns bounds for parameters
compute_variance(parameters, resids, sigma2, ...)	Compute the variance for the ARCH model
constraints()	Construct parameter constraints arrays for parameter estimation
forecast(parameters, resids, backcast, ...)	Forecast volatility from the model
parameter_names()	Names of model parameters
simulate(parameters, nobs, rng[, burn, ...])	Simulate data from the model
starting_values(resids)	Returns starting values for the ARCH model
update(index, parameters, resids, sigma2, ...)	Compute the variance for a single observation
variance_bounds(resids[, power])	Construct loose bounds for conditional variances.

arch.univariate.volatility.VolatilityProcess.backcast

VolatilityProcess.**backcast**(resids: *ndarray*) → float | ndarray

Construct values for backcasting to start the recursion

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
backcast – Value to use in backcasting in the volatility recursion

Return type
float

arch.univariate.volatility.VolatilityProcess.backcast_transform

`VolatilityProcess.backcast_transform(backcast: float | ndarray) → float | ndarray`

Transformation to apply to user-provided backcast values

Parameters

backcast: float | ndarray
User-provided `backcast` that approximates sigma2[0].

Returns
backcast – Backcast transformed to the model-appropriate scale

Return type
`{float, ndarray}`

arch.univariate.volatility.VolatilityProcess.bounds

`abstract VolatilityProcess.bounds(resids: ndarray) → list[tuple[float, float]]`

Returns bounds for parameters

Parameters

resids: ndarray
Vector of (approximate) residuals

Returns
bounds – List of bounds where each element is (lower, upper).

Return type
`list[tuple[float,float]]`

arch.univariate.volatility.VolatilityProcess.compute_variance

`abstract VolatilityProcess.compute_variance(parameters: ndarray, resids: ndarray, sigma2: ndarray, backcast: float | ndarray, var_bounds: ndarray) → ndarray`

Compute the variance for the ARCH model

Parameters

parameters: ndarray
Model parameters

resids: ndarray
Vector of mean zero residuals

sigma2: ndarray
Array with same size as resids to store the conditional variance

backcast: float | ndarray
Value to use when initializing ARCH recursion. Can be an ndarray when the model contains multiple components.

var_bounds: ndarray
Array containing columns of lower and upper bounds

arch.univariate.volatility.VolatilityProcess.constraints

abstract VolatilityProcess.**constraints**() → tuple[ndarray, ndarray]

Construct parameter constraints arrays for parameter estimation

Returns

- **A** (numpy.ndarray) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (numpy.ndarray) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form A.dot(parameters) - b >= 0

arch.univariate.volatility.VolatilityProcess.forecast

VolatilityProcess.**forecast**(parameters: *ArrayLike1D*, resids: *Float64Array*, backcast: *Float64Array | float*, var_bounds: *Float64Array*, start: *int | None* = **None**, horizon: *int* = **1**, method: *ForecastingMethod* = 'analytic', simulations: *int* = **1000**, rng: *RNGType | None* = **None**, random_state: *RandomState | None* = **None**) → VarianceForecast

Forecast volatility from the model

Parameters

parameters: ArrayLike1D
Parameters required to forecast the volatility model

resids: Float64Array
Residuals to use in the recursion

backcast: Float64Array | float
Value to use when initializing the recursion

var_bounds: Float64Array
Array containing columns of lower and upper bounds

start: int | None = None
Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon: int = 1
Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method: ForecastingMethod = 'analytic'
Method to use when producing the forecast. The default is analytic.

simulations: int = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: RNGType | None = None
Callable random number generator required if method is 'simulation'. Must take a single shape input and return random samples numbers with that shape.

random_state: RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

Returns
forecasts – Class containing the variance forecasts, and, if using simulation or bootstrap, the simulated paths.

Return type
`arch.univariate.volatility.VarianceForecast`

Raises

- `NotImplementedError` –
  - If method is not supported
- `ValueError` –
  - If the method is not known

Notes

The analytic `method` is not supported for all models. Attempting to use this method when not available will raise a ValueError.

arch.univariate.volatility.VolatilityProcess.parameter_names

`abstract VolatilityProcess.parameter_names()` → list[str]

Names of model parameters

Returns
names – Variables names

Return type
`list (str)`

arch.univariate.volatility.VolatilityProcess.simulate

`abstract VolatilityProcess.simulate(`parameters: *Sequence[float | int] | ndarray | Series*, nobs: *int*, rng: *Callable[[int | tuple[int, ...]], ndarray]*, burn: *int* = 500, initial_value: *None | float | ndarray* = None) → tuple[ndarray, ndarray]

Simulate data from the model

Parameters

parameters: Sequence[float | int] | ndarray | Series
Parameters required to simulate the volatility model

nobs: int
Number of data points to simulate

rng: Callable[[int | tuple[int, ...]], ndarray]
Callable function that takes a single integer input and returns a vector of random numbers

burn: int = 500
Number of additional observations to generate when initializing the simulation

initial_value: None | float | ndarray = None
Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (`numpy.ndarray`) – The simulated residuals
- **variance** (`numpy.ndarray`) – The simulated variance

arch.univariate.volatility.VolatilityProcess.starting_values

abstract `VolatilityProcess.starting_values(resids: ndarray) → ndarray`

Returns starting values for the ARCH model

Parameters

resids: ndarray
Array of (approximate) residuals to use when computing starting values

Returns
sv – Array of starting values

Return type
`numpy.ndarray`

arch.univariate.volatility.VolatilityProcess.update

`VolatilityProcess.update(index: int, parameters: ndarray, resids: ndarray, sigma2: ndarray, backcast: float | ndarray, var_bounds: ndarray) → float`

Compute the variance for a single observation

Parameters

index: int
The numerical index of the variance to compute

parameters: ndarray
The variance model parameters

resids: ndarray
The residual array. Only uses `resids[:index]` when computing `sigma2[index]`

sigma2: ndarray
The array containing the variances. Only uses `sigma2[:index]` when computing `sigma2[index]`. The computed value is stored in `sigma2[index]`.

backcast: float | ndarray
Value to use when initializing the recursion

var_bounds: ndarray
Array containing columns of lower and upper bounds

Returns
The variance computed for location `index`

Return type
`float`

arch.univariate.volatility.VolatilityProcess.variance_bounds

`VolatilityProcess.variance_bounds(resids: ndarray, power: float = 2.0) → ndarray`

Construct loose bounds for conditional variances.

These bounds are used in parameter estimation to ensure that the log-likelihood does not produce NaN values.

Parameters

resids: ndarray
Approximate residuals to use to compute the lower and upper bounds on the conditional variance

power: float = 2.0
Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

Returns
var_bounds – Array containing columns of lower and upper bounds with the same number of elements as resids

Return type
`numpy.ndarray`

Properties

name	The name of the volatility process
num_params	The number of parameters in the model
start	Index to use to start variance subarray selection
stop	Index to use to stop variance subarray selection
updateable	Flag indicating that the volatility process supports update
volatility_updater	Get the volatility updater associated with the volatility process

arch.univariate.volatility.VolatilityProcess.name

`property VolatilityProcess.name : str`

The name of the volatility process

arch.univariate.volatility.VolatilityProcess.num_params

`property VolatilityProcess.num_params : int`

The number of parameters in the model

arch.univariate.volatility.VolatilityProcess.start

property VolatilityProcess.**start** : int

Index to use to start variance subarray selection

arch.univariate.volatility.VolatilityProcess.stop

property VolatilityProcess.**stop** : int

Index to use to stop variance subarray selection

arch.univariate.volatility.VolatilityProcess.updateable

property VolatilityProcess.**updateable** : bool

Flag indicating that the volatility process supports update

arch.univariate.volatility.VolatilityProcess.volatility_updater

property VolatilityProcess.**volatility_updater** : VolatilityUpdater

Get the volatility updater associated with the volatility process

Returns
The updater class

Return type
`VolatilityUpdater`

Raises
`NotImplementedError` – If the process is not updateable

They may optionally expose a *VolatilityUpdater* class that can be used in *ARCHInMean* estimation.

VolatilityUpdater()	Base class that all volatility updaters must inherit from.

arch.univariate.recursions_python.VolatilityUpdater

class arch.univariate.recursions_python.**VolatilityUpdater**

Base class that all volatility updaters must inherit from.

Notes

See the implementation available for information on modifying `__init__` to capture model-specific parameters and how `initialize_update` is used to precompute values that change in each likelihood but not each iteration of the recursion.

When writing a volatility updater, it is recommended to follow the examples in recursions.pyx which use Cython to produce a C-callable update function that can then be used to improve performance. The subclasses of this abstract metaclass are all pure Python and model estimation performance is poor since loops are written in Python.

Methods

initialize_update(parameters, backcast, nobs)	Initialize the recursion prior to calling update
update(t, parameters, resids, sigma2, var_bounds)	Update the current variance at location t

arch.univariate.recursions_python.VolatilityUpdater.initialize_update

abstract VolatilityUpdater.**initialize_update**(parameters: *ndarray*, backcast: *float* | *ndarray*, nobs: *int*) → None

Initialize the recursion prior to calling update

Parameters

parameters: ndarray
The model parameters.

backcast: float | ndarray
The backcast value(s).

nobs: int
The number of observations in the sample.

Notes

This function is called once per likelihood evaluation and can be used to pre-compute expensive parameter transformations that do not change with each call to `update`.

arch.univariate.recursions_python.VolatilityUpdater.update

abstract VolatilityUpdater.**update**(t: *int*, parameters: *ndarray*, resids: *ndarray*, sigma2: *ndarray*, var_bounds: *ndarray*) → None

Update the current variance at location t

Parameters

t: int
The index of the value of sigma2 to update. Assumes but does not check that update has been called recursively for 0,1,…,t-1.

parameters: ndarray
Model parameters

resids: ndarray
Residuals to use in the recursion

sigma2: ndarray
Conditional variances with same shape as resids

var_bounds: ndarray
nobs by 2-element array of upper and lower bounds for conditional variances for each time period

Notes

The update to sigma2 occurs inplace.

1.10 Using the Fixed Variance process

The FixedVariance volatility process can be used to implement zig-zag model estimation where two steps are repeated until convergence. This can be used to estimate models which may not be easy to estimate as a single process due to numerical issues or a high-dimensional parameter space.

This setup code is required to run in an IPython notebook

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn

seaborn.set_style("darkgrid")
plt.rc("figure", figsize=(16, 6))
plt.rc("savefig", dpi=90)
plt.rc("font", family="sans-serif")
plt.rc("font", size=14)
```

1.10.1 Setup

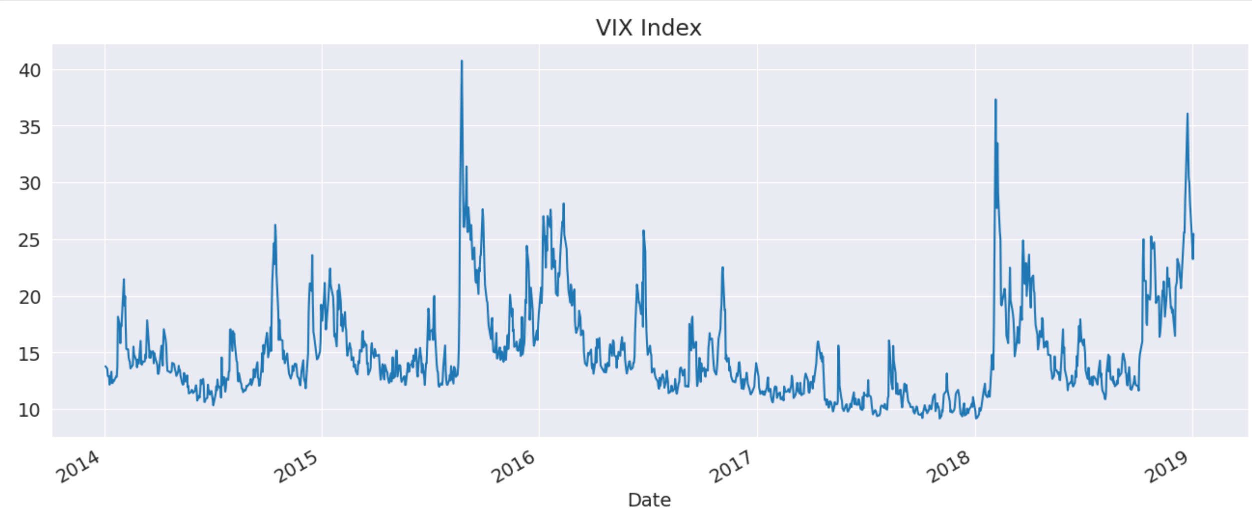
Imports used in this example.

[2]:
```
import datetime as dt

import numpy as np
```

Data

The VIX index will be used to illustrate the use of the FixedVariance process. The data is from FRED and is provided by the arch package.

[3]:
```
import arch.data.vix

vix_data = arch.data.vix.load()
vix = vix_data.vix.dropna()
vix.name = "VIX Index"
ax = vix.plot(title="VIX Index")
```



Initial Mean Model Estimation

The first step is to estimate the mean to filter the residuals using a constant variance.

[4]:
```python
from arch.univariate.mean import HARX, ZeroMean
from arch.univariate.volatility import GARCH, FixedVariance

mod = HARX(vix, lags=[1, 5, 22])
res = mod.fit()
print(res.summary())
```

```
                    HAR - Constant Variance Model Results
==============================================================================
Dep. Variable:              VIX Index   R-squared:                       0.876
Mean Model:                       HAR   Adj. R-squared:                  0.876
Vol Model:          Constant Variance   Log-Likelihood:               -2267.95
Distribution:                  Normal   AIC:                           4545.90
Method:            Maximum Likelihood   BIC:                           4571.50
                                        No. Observations:                 1237
Date:                Fri, Jan 05 2024   Df Residuals:                     1233
Time:                        16:02:15   Df Model:                            4
                               Mean Model
================================================================================
                     coef    std err          t      P>|t|      95.0% Conf. Int.
--------------------------------------------------------------------------------
Const              0.6335      0.189      3.359  7.831e-04     [  0.264,  1.003]
VIX Index[0:1]     0.9287  6.589e-02     14.095  4.056e-45     [  0.800,  1.058]
VIX Index[0:5]    -0.0318  6.449e-02     -0.492      0.622  [ -0.158,9.463e-02]
VIX Index[0:22]    0.0612  3.180e-02      1.926  5.409e-02 [-1.076e-03,  0.124]
                            Volatility Model
========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
sigma2         2.2910      0.396      5.782  7.361e-09 [  1.514,  3.068]
========================================================================
```

(continues on next page)

(continued from previous page)

```
Covariance estimator: White's Heteroskedasticity Consistent Estimator
```

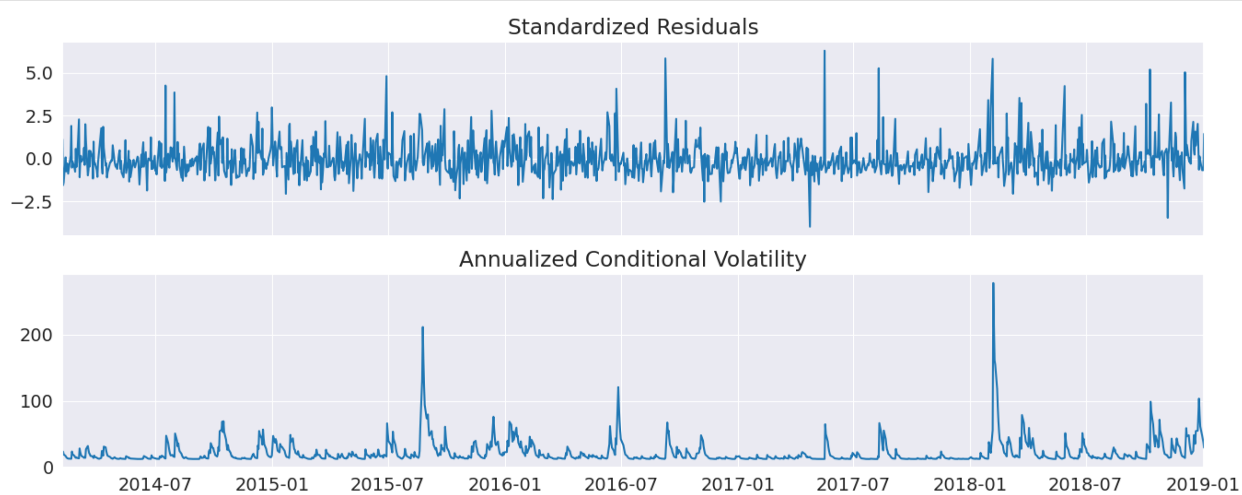
Initial Volatility Model Estimation

Using the previously estimated residuals, a volatility model can be estimated using a ZeroMean. In this example, a GJR-GARCH process is used for the variance.

[5]:
```
vol_mod = ZeroMean(res.resid.dropna(), volatility=GARCH(p=1, o=1, q=1))
vol_res = vol_mod.fit(disp="off")
print(vol_res.summary())
```

```
                     Zero Mean - GJR-GARCH Model Results
==============================================================================
Dep. Variable:                  resid   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.001
Vol Model:                  GJR-GARCH   Log-Likelihood:               -1936.93
Distribution:                  Normal   AIC:                           3881.86
Method:            Maximum Likelihood   BIC:                           3902.35
                                        No. Observations:                 1237
Date:                Fri, Jan 05 2024   Df Residuals:                     1237
Time:                        16:02:15   Df Model:                            0
                              Volatility Model
===========================================================================
                 coef    std err          t      P>|t|     95.0% Conf. Int.
---------------------------------------------------------------------------
omega          0.2355  9.134e-02      2.578  9.932e-03  [5.647e-02,  0.415]
alpha[1]       0.7217      0.374      1.931  5.353e-02 [-1.098e-02,  1.454]
gamma[1]      -0.7217      0.252     -2.859  4.255e-03   [ -1.217, -0.227]
beta[1]        0.5789      0.184      3.140  1.692e-03   [  0.218,  0.940]
===========================================================================

Covariance estimator: robust
```

[6]:
```
ax = vol_res.plot("D")
```



Re-estimating the mean with a FixedVariance

The FixedVariance requires that the variance is provided when initializing the object. The variance provided should have the same shape as the original data. Since the variance estimated from the GJR-GARCH model is missing the first 22 observations due to the HAR lags, we simply fill these with 1. These values will not be used to estimate the model, and so the value is not important.

The summary shows that there is a single parameter, scale, which is close to 1. The mean parameters have changed which reflects the GLS-like weighting that this re-estimation imposes.

[7]:
```
variance = np.empty_like(vix)
variance.fill(1.0)
variance[22:] = vol_res.conditional_volatility**2.0
fv = FixedVariance(variance)
mod = HARX(vix, lags=[1, 5, 22], volatility=fv)
res = mod.fit()
print(res.summary())
```

```
Iteration:      1,   Func. Count:      7,   Neg. LLF: 255810120110.59235
Iteration:      2,   Func. Count:     19,   Neg. LLF: 930358.9478537004
Iteration:      3,   Func. Count:     28,   Neg. LLF: 3486.6993418452503
Iteration:      4,   Func. Count:     36,   Neg. LLF: 2885.687080924194
Iteration:      5,   Func. Count:     44,   Neg. LLF: 65536358.857983164
Iteration:      6,   Func. Count:     53,   Neg. LLF: 1935.952754561917
Iteration:      7,   Func. Count:     59,   Neg. LLF: 1935.9470521047324
Iteration:      8,   Func. Count:     65,   Neg. LLF: 1935.9470514942734
Optimization terminated successfully    (Exit mode 0)
            Current function value: 1935.9470514942734
            Iterations: 8
            Function evaluations: 65
            Gradient evaluations: 8
                      HAR - Fixed Variance Model Results
==============================================================================
Dep. Variable:              VIX Index   R-squared:                       0.876
Mean Model:                       HAR   Adj. R-squared:                  0.876
Vol Model:             Fixed Variance   Log-Likelihood:               -1935.95
Distribution:                  Normal   AIC:                           3881.89
Method:            Maximum Likelihood   BIC:                           3907.50
                                        No. Observations:                 1237
Date:                Fri, Jan 05 2024   Df Residuals:                     1233
Time:                        16:02:16   Df Model:                            4
                                  Mean Model
==================================================================================
                      coef    std err          t      P>|t|       95.0% Conf. Int.
----------------------------------------------------------------------------------
Const               0.5584      0.153      3.661  2.507e-04      [  0.260,  0.857]
VIX Index[0:1]      0.9376  3.625e-02     25.866 1.607e-147      [  0.867,  1.009]
VIX Index[0:5]     -0.0249  3.782e-02     -0.657      0.511 [-9.899e-02,4.926e-02]
VIX Index[0:22]     0.0493  2.102e-02      2.344  1.909e-02  [8.064e-03,9.044e-02]
                            Volatility Model
========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
scale          0.9986  8.081e-02     12.358  4.420e-35 [  0.840,  1.157]
```

(continues on next page)

(continued from previous page)

```
========================================================================

Covariance estimator: robust
```

Zig-Zag estimation

A small repetitions of the previous two steps can be used to implement a so-called zig-zag estimation strategy.

[8]:
```python
for i in range(5):
    print(i)
    vol_mod = ZeroMean(res.resid.dropna(), volatility=GARCH(p=1, o=1, q=1))
    vol_res = vol_mod.fit(disp="off")
    variance[22:] = vol_res.conditional_volatility**2.0
    fv = FixedVariance(variance, unit_scale=True)
    mod = HARX(vix, lags=[1, 5, 22], volatility=fv)
    res = mod.fit(disp="off")
print(res.summary())
```

```
0
1
2
3
```

```
4
```

```
                    HAR - Fixed Variance (Unit Scale) Model Results
=======================================================================================
Dep. Variable:                     VIX Index   R-squared:                       0.876
Mean Model:                              HAR   Adj. R-squared:                  0.876
Vol Model:        Fixed Variance (Unit Scale)   Log-Likelihood:               -1935.74
Distribution:                         Normal   AIC:                           3879.48
Method:                   Maximum Likelihood   BIC:                           3899.96
                                               No. Observations:                 1237
Date:                       Fri, Jan 05 2024   Df Residuals:                     1233
Time:                               16:02:16   Df Model:                            4
                                  Mean Model
=================================================================================
                     coef    std err          t      P>|t|      95.0% Conf. Int.
---------------------------------------------------------------------------------
Const              0.5602      0.152      3.681  2.323e-04     [  0.262,  0.858]
VIX Index[0:1]     0.9381  3.616e-02     25.939 2.390e-148     [  0.867,  1.009]
VIX Index[0:5]    -0.0262  3.774e-02     -0.693      0.488   [ -0.100,4.781e-02]
VIX Index[0:22]    0.0499  2.099e-02      2.380  1.733e-02 [8.810e-03,9.109e-02]
=================================================================================

Covariance estimator: robust
```

Direct Estimation

This model can be directly estimated. The results are provided for comparison to the previous `FixedVariance` estimates of the mean parameters.

[9]:
```
mod = HARX(vix, lags=[1, 5, 22], volatility=GARCH(1, 1, 1))
res = mod.fit(disp="off")
print(res.summary())
```

```
                         HAR - GJR-GARCH Model Results
==============================================================================
Dep. Variable:                VIX Index   R-squared:                       0.876
Mean Model:                         HAR   Adj. R-squared:                  0.875
Vol Model:                    GJR-GARCH   Log-Likelihood:               -1932.61
Distribution:                    Normal   AIC:                           3881.23
Method:              Maximum Likelihood   BIC:                           3922.19
                                          No. Observations:                 1237
Date:                  Fri, Jan 05 2024   Df Residuals:                     1233
Time:                          16:02:17   Df Model:                            4
                                 Mean Model
================================================================================
                     coef    std err          t      P>|t|     95.0% Conf. Int.
--------------------------------------------------------------------------------
Const              0.7796      1.190      0.655      0.513    [ -1.554,  3.113]
VIX Index[0:1]     0.9180      0.291      3.156  1.597e-03    [  0.348,  1.488]
VIX Index[0:5]    -0.0393      0.296     -0.133      0.894    [ -0.620,  0.541]
VIX Index[0:22]    0.0632  6.353e-02      0.994      0.320 [-6.136e-02,  0.188]
                            Volatility Model
==========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
--------------------------------------------------------------------------
omega          0.2357      0.250      0.944      0.345 [ -0.254,  0.725]
alpha[1]       0.7091      1.069      0.664      0.507 [ -1.386,  2.804]
gamma[1]      -0.7091      0.519     -1.367      0.172 [ -1.726,  0.308]
beta[1]        0.5579      0.855      0.653      0.514 [ -1.117,  2.233]
==========================================================================

Covariance estimator: robust
```

1.11 Distributions

A distribution is the final component of an ARCH Model.

Normal([random_state, seed])	Standard normal distribution for use with ARCH models
StudentsT([random_state, seed])	Standardized Student's distribution for use with ARCH models
SkewStudent([random_state, seed])	Standardized Skewed Student's distribution for use with ARCH models
GeneralizedError([random_state, seed])	Generalized Error distribution for use with ARCH models

1.11.1 arch.univariate.Normal

class arch.univariate.**Normal**(random_state: *RandomState | None* = ***None***, *, seed: *None | int | RandomState | Generator* = ***None***)

Standard normal distribution for use with ARCH models

Parameters

random_state: RandomState | None = None

Deprecated since version 5.0: random_state is deprecated. Use seed instead.

seed: None | int | RandomState | Generator = None

Random number generator instance or int to use. Set to ensure reproducibility. If using an int, the argument is passed to `np.random.default_rng`. If not provided, `default_rng` is used with system-provided entropy.

Methods

bounds(resids)	Parameter bounds for use in optimization.
cdf(resids[, parameters])	Cumulative distribution function
constraints()	Construct arrays to use in constrained optimization.
loglikelihood(parameters, resids, sigma2[, ...])	Computes the log-likelihood of assuming residuals are normally distributed, conditional on the variance
moment(n[, parameters])	Moment of order n
parameter_names()	Names of distribution shape parameters
partial_moment(n[, z, parameters])	Order n lower partial moment from -inf to z
ppf(pits[, parameters])	Inverse cumulative density function (ICDF)
simulate(parameters)	Simulates i.i.d.
starting_values(std_resid)	Construct starting values for use in optimization.

arch.univariate.Normal.bounds

Normal.**bounds**(resids: *ndarray*) → list[tuple[float, float]]

Parameter bounds for use in optimization.

Parameters

resids: ndarray

Residuals to use when computing the bounds

Returns

bounds – List containing a single tuple with (lower, upper) bounds

Return type

list

arch.univariate.Normal.cdf

Normal.**cdf**(resids: *Sequence[float] | ndarray | Series*, parameters: *None | Sequence[float] | ndarray | Series = None*) → ndarray

Cumulative distribution function

Parameters

resids: Sequence[float] | ndarray | Series
Values at which to evaluate the cdf

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use `None` for parameterless distributions.

Returns
f – CDF values

Return type
`numpy.ndarray`

arch.univariate.Normal.constraints

Normal.**constraints**() → tuple[ndarray, ndarray]

Construct arrays to use in constrained optimization.

Returns

- **A** (`numpy.ndarray`) – Constraint loadings
- **b** (`numpy.ndarray`) – Constraint values

Notes

Parameters satisfy the constraints A.dot(parameters)-b >= 0

arch.univariate.Normal.loglikelihood

Normal.**loglikelihood**(parameters: *Sequence[float] | ndarray | Series*, resids: *ndarray | DataFrame | Series*, sigma2: *ndarray | DataFrame | Series*, individual: *bool = False*) → float | ndarray

Computes the log-likelihood of assuming residuals are normally distributed, conditional on the variance

Parameters

parameters: Sequence[float] | ndarray | Series
The normal likelihood has no shape parameters. Empty since the standard normal has no shape parameters.

resids: ndarray | DataFrame | Series
The residuals to use in the log-likelihood calculation

sigma2: ndarray | DataFrame | Series
Conditional variances of resids

individual: bool = False
Flag indicating whether to return the vector of individual log likelihoods (True) or the sum (False)

Returns
ll – The log-likelihood

Return type
`float`

Notes

The log-likelihood of a single data point x is

$$\ln f\left(x\right)=-\frac{1}{2}\left(\ln 2\pi+\ln \sigma^{2}+\frac{x^{2}}{\sigma^{2}}\right)$$

arch.univariate.Normal.moment

`Normal.moment(n: int, parameters: None | Sequence[float] | ndarray | Series = None) → float`

Moment of order n

Parameters

n: int
Order of moment

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use None for parameterless distributions.

Returns
Calculated moment

Return type
`float`

arch.univariate.Normal.parameter_names

`Normal.parameter_names() → list[str]`

Names of distribution shape parameters

Returns
names – Parameter names

Return type
`list (str)`

arch.univariate.Normal.partial_moment

`Normal.partial_moment(n: int, z: float = 0.0, parameters: None | Sequence[float] | ndarray | Series = None) → float`

Order n lower partial moment from -inf to z

Parameters

n: int
Order of partial moment

z: float = 0.0
Upper bound for partial moment integral

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use None for parameterless distributions.

Returns
Partial moment

Return type
`float`

References

Notes

The order n lower partial moment to z is

$$\int_{-\infty}^{z} x^n f(x) dx$$

See[1] for more details.

arch.univariate.Normal.ppf

`Normal.ppf`(pits*: float | Sequence[float] | ndarray | Series*, parameters*: None | Sequence[float] | ndarray | Series = None*) → ndarray

Inverse cumulative density function (ICDF)

Parameters

pits: float | Sequence[float] | ndarray | Series
Probability-integral-transformed values in the interval (0, 1).

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use `None` for parameterless distributions.

Returns
i – Inverse CDF values

Return type
`{float, ndarray}`

arch.univariate.Normal.simulate

`Normal.simulate`(parameters*: int | float | Sequence[float | int] | ndarray | Series*) → Callable[[int | tuple[int, ...]], ndarray]

Simulates i.i.d. draws from the distribution

Parameters

parameters: int | float | Sequence[float | int] | ndarray | Series
Distribution parameters

Returns
simulator – Callable that take a single output size argument and returns i.i.d. draws from the distribution

Return type
`callable`

[1] Winkler et al. (1972) "The Determination of Partial Moments" *Management Science* Vol. 19 No. 3

arch.univariate.Normal.starting_values

Normal.**starting_values**(std_resid: *ndarray*) → ndarray

Construct starting values for use in optimization.

Parameters

std_resid: ndarray
Estimated standardized residuals to use in computing starting values for the shape parameter

Returns
sv – The estimated shape parameters for the distribution

Return type
`numpy.ndarray`

Notes

Size of sv depends on the distribution

Properties

generator	The NumPy Generator or RandomState attached to the distribution
name	The name of the distribution
random_state	The NumPy RandomState attached to the distribution

arch.univariate.Normal.generator

property Normal.**generator** : RandomState | Generator

The NumPy Generator or RandomState attached to the distribution

arch.univariate.Normal.name

property Normal.**name** : str

The name of the distribution

arch.univariate.Normal.random_state

property Normal.**random_state** : RandomState | Generator

The NumPy RandomState attached to the distribution

Deprecated since version 5.0: random_state is deprecated. Use generator instead.

1.11.2 arch.univariate.StudentsT

class arch.univariate.StudentsT(random_state: *RandomState | None* = None, *, seed: *None | int | RandomState | Generator* = None)

Standardized Student's distribution for use with ARCH models

Parameters

random_state: RandomState | None = None

Deprecated since version 5.0: random_state is deprecated. Use seed instead.

seed: None | int | RandomState | Generator = None

Random number generator instance or int to use. Set to ensure reproducibility. If using an int, the argument is passed to `np.random.default_rng`. If not provided, `default_rng` is used with system-provided entropy.

Methods

bounds(resids)	Parameter bounds for use in optimization.
cdf(resids[, parameters])	Cumulative distribution function
constraints()	Construct arrays to use in constrained optimization.
loglikelihood(parameters, resids, sigma2[, ...])	Computes the log-likelihood of assuming residuals are have a standardized (to have unit variance) Student's t distribution, conditional on the variance.
moment(n[, parameters])	Moment of order n
parameter_names()	Names of distribution shape parameters
partial_moment(n[, z, parameters])	Order n lower partial moment from -inf to z
ppf(pits[, parameters])	Inverse cumulative density function (ICDF)
simulate(parameters)	Simulates i.i.d.
starting_values(std_resid)	Construct starting values for use in optimization.

arch.univariate.StudentsT.bounds

StudentsT.bounds(resids: *ndarray*) → list[tuple[float, float]]

Parameter bounds for use in optimization.

Parameters

resids: ndarray

Residuals to use when computing the bounds

Returns

bounds – List containing a single tuple with (lower, upper) bounds

Return type

`list`

arch.univariate.StudentsT.cdf

StudentsT.**cdf**(resids: *Sequence[float] | ndarray | Series*, parameters: *None | Sequence[float] | ndarray | Series = None*) → ndarray

Cumulative distribution function

Parameters

resids: Sequence[float] | ndarray | Series
Values at which to evaluate the cdf

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use `None` for parameterless distributions.

Returns
f – CDF values

Return type
`numpy.ndarray`

arch.univariate.StudentsT.constraints

StudentsT.**constraints**() → tuple[ndarray, ndarray]

Construct arrays to use in constrained optimization.

Returns

- **A** (`numpy.ndarray`) – Constraint loadings
- **b** (`numpy.ndarray`) – Constraint values

Notes

Parameters satisfy the constraints A.dot(parameters)-b >= 0

arch.univariate.StudentsT.loglikelihood

StudentsT.**loglikelihood**(parameters: *Sequence[float] | ndarray | Series*, resids: *ndarray | DataFrame | Series*, sigma2: *ndarray | DataFrame | Series*, individual: *bool = False*) → float | ndarray

Computes the log-likelihood of assuming residuals are have a standardized (to have unit variance) Student's t distribution, conditional on the variance.

Parameters

parameters: Sequence[float] | ndarray | Series
Shape parameter of the t distribution

resids: ndarray | DataFrame | Series
The residuals to use in the log-likelihood calculation

sigma2: ndarray | DataFrame | Series
Conditional variances of resids

individual: bool = False
Flag indicating whether to return the vector of individual log likelihoods (True) or the sum (False)

Returns
: **ll** – The log-likelihood

Return type
: `float`

Notes

The log-likelihood of a single data point x is

$$\ln\Gamma\left(\frac{\nu+1}{2}\right)-\ln\Gamma\left(\frac{\nu}{2}\right)-\frac{1}{2}\ln(\pi\left(\nu-2\right)\sigma^{2})-\frac{\nu+1}{2}\ln(1+x^{2}/(\sigma^{2}(\nu-2)))$$

where Γ is the gamma function.

arch.univariate.StudentsT.moment

`StudentsT.moment(n: int, parameters: None | Sequence[float] | ndarray | Series = None) → float`

Moment of order n

Parameters

n: int
: Order of moment

parameters: None | Sequence[float] | ndarray | Series = None
: Distribution parameters. Use None for parameterless distributions.

Returns
: Calculated moment

Return type
: `float`

arch.univariate.StudentsT.parameter_names

`StudentsT.parameter_names() → list[str]`

Names of distribution shape parameters

Returns
: **names** – Parameter names

Return type
: `list (str)`

arch.univariate.StudentsT.partial_moment

`StudentsT.partial_moment(n: int, z: float = 0.0, parameters: None | Sequence[float] | ndarray | Series = None) → float`

Order n lower partial moment from -inf to z

Parameters

n: int
: Order of partial moment

z: float = 0.0
Upper bound for partial moment integral

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use None for parameterless distributions.

Returns
Partial moment

Return type
`float`

References

Notes

The order n lower partial moment to z is

$$\int_{-\infty}^{z} x^n f(x) dx$$

See[1] for more details.

arch.univariate.StudentsT.ppf

`StudentsT.ppf(pits: float | Sequence[float] | ndarray | Series, parameters: None | Sequence[float] | ndarray | Series = None) → ndarray`

Inverse cumulative density function (ICDF)

Parameters

pits: float | Sequence[float] | ndarray | Series
Probability-integral-transformed values in the interval (0, 1).

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use `None` for parameterless distributions.

Returns
i – Inverse CDF values

Return type
`{float, ndarray}`

arch.univariate.StudentsT.simulate

`StudentsT.simulate(parameters: int | float | Sequence[float | int] | ndarray | Series) → Callable[[int | tuple[int, ...]], ndarray]`

Simulates i.i.d. draws from the distribution

Parameters

parameters: int | float | Sequence[float | int] | ndarray | Series
Distribution parameters

[1] Winkler et al. (1972) "The Determination of Partial Moments" *Management Science* Vol. 19 No. 3

Returns
simulator – Callable that take a single output size argument and returns i.i.d. draws from the distribution

Return type
`callable`

arch.univariate.StudentsT.starting_values

`StudentsT.starting_values(std_resid: ndarray) → ndarray`

Construct starting values for use in optimization.

Parameters

std_resid: ndarray
Estimated standardized residuals to use in computing starting values for the shape parameter

Returns
sv – Array containing starting valuer for shape parameter

Return type
`numpy.ndarray`

Notes

Uses relationship between kurtosis and degree of freedom parameter to produce a moment-based estimator for the starting values.

Properties

generator	The NumPy Generator or RandomState attached to the distribution
name	The name of the distribution
random_state	The NumPy RandomState attached to the distribution

arch.univariate.StudentsT.generator

`property StudentsT.generator : RandomState | Generator`

The NumPy Generator or RandomState attached to the distribution

arch.univariate.StudentsT.name

`property StudentsT.name : str`

The name of the distribution

arch.univariate.StudentsT.random_state

property StudentsT.**random_state** : RandomState | Generator

The NumPy RandomState attached to the distribution

Deprecated since version 5.0: random_state is deprecated. Use generator instead.

1.11.3 arch.univariate.SkewStudent

class arch.univariate.**SkewStudent**(random_state: *RandomState | None = None*, *, seed: *None | int | RandomState | Generator = None*)

Standardized Skewed Student's distribution for use with ARCH models

Parameters

random_state: RandomState | None = None

Deprecated since version 5.0: random_state is deprecated. Use seed instead.

seed: None | int | RandomState | Generator = None

Random number generator instance or int to use. Set to ensure reproducibility. If using an int, the argument is passed to `np.random.default_rng`. If not provided, `default_rng` is used with system-provided entropy.

Notes

The Standardized Skewed Student's distribution ([1]) takes two parameters, η and λ. η controls the tail shape and is similar to the shape parameter in a Standardized Student's t. λ controls the skewness. When $\lambda = 0$ the distribution is identical to a standardized Student's t.

References

Methods

bounds(resids)	Parameter bounds for use in optimization.
cdf(resids[, parameters])	Cumulative distribution function
constraints()	Construct arrays to use in constrained optimization.
loglikelihood(parameters, resids, sigma2[, ...])	Computes the log-likelihood of assuming residuals are have a standardized (to have unit variance) Skew Student's t distribution, conditional on the variance.
moment(n[, parameters])	Moment of order n
parameter_names()	Names of distribution shape parameters
partial_moment(n[, z, parameters])	Order n lower partial moment from -inf to z
ppf(pits[, parameters])	Inverse cumulative density function (ICDF)
simulate(parameters)	Simulates i.i.d.
starting_values(std_resid)	Construct starting values for use in optimization.

[1] Hansen, B. E. (1994). Autoregressive conditional density estimation. *International Economic Review*, 35(3), 705–730. <https://www.ssc.wisc.edu/~bhansen/papers/ier_94.pdf>

arch.univariate.SkewStudent.bounds

`SkewStudent.bounds(resids: ndarray) → list[tuple[float, float]]`

Parameter bounds for use in optimization.

Parameters

resids: ndarray
Residuals to use when computing the bounds

Returns
bounds – List containing a single tuple with (lower, upper) bounds

Return type
`list`

arch.univariate.SkewStudent.cdf

`SkewStudent.cdf(resids: Sequence[float] | ndarray | Series, parameters: None | Sequence[float] | ndarray | Series = None) → ndarray`

Cumulative distribution function

Parameters

resids: Sequence[float] | ndarray | Series
Values at which to evaluate the cdf

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use `None` for parameterless distributions.

Returns
f – CDF values

Return type
`numpy.ndarray`

arch.univariate.SkewStudent.constraints

`SkewStudent.constraints() → tuple[ndarray, ndarray]`

Construct arrays to use in constrained optimization.

Returns

- **A** (`numpy.ndarray`) – Constraint loadings
- **b** (`numpy.ndarray`) – Constraint values

Notes

Parameters satisfy the constraints A.dot(parameters)-b >= 0

arch.univariate.SkewStudent.loglikelihood

SkewStudent.**loglikelihood**(parameters: *Sequence[float]* | *ndarray* | *Series*, resids: *ndarray* | *DataFrame* | *Series*, sigma2: *ndarray* | *DataFrame* | *Series*, individual: *bool* = ***False***) → ndarray

Computes the log-likelihood of assuming residuals are have a standardized (to have unit variance) Skew Student's t distribution, conditional on the variance.

Parameters

parameters: Sequence[float] | ndarray | Series
Shape parameter of the skew-t distribution

resids: ndarray | DataFrame | Series
The residuals to use in the log-likelihood calculation

sigma2: ndarray | DataFrame | Series
Conditional variances of resids

individual: bool = False
Flag indicating whether to return the vector of individual log likelihoods (True) or the sum (False)

Returns
ll – The log-likelihood

Return type
`float`

Notes

The log-likelihood of a single data point x is

$$\ln\left[\frac{bc}{\sigma}\left(1+\frac{1}{\eta-2}\left(\frac{a+bx/\sigma}{1+sgn(x/\sigma+a/b)\lambda}\right)^2\right)^{-(\eta+1)/2}\right],$$

where $2<\eta<\infty$, and $-1<\lambda<1$. The constants a, b, and c are given by

$$a=4\lambda c\frac{\eta-2}{\eta-1},\quad b^2=1+3\lambda^2-a^2,\quad c=\frac{\Gamma\left(\frac{\eta+1}{2}\right)}{\sqrt{\pi\left(\eta-2\right)}\Gamma\left(\frac{\eta}{2}\right)},$$

and Γ is the gamma function.

arch.univariate.SkewStudent.moment

SkewStudent.**moment**(n: *int*, parameters: *None* | *Sequence[float]* | *ndarray* | *Series* = ***None***) → float

Moment of order n

Parameters

n: int
Order of moment

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use None for parameterless distributions.

Returns
Calculated moment

Return type
float

arch.univariate.SkewStudent.parameter_names

SkewStudent.**parameter_names**() → list[str]

Names of distribution shape parameters

Returns
names – Parameter names

Return type
list (str)

arch.univariate.SkewStudent.partial_moment

SkewStudent.**partial_moment**(n: *int*, z: *float* = 0.0, parameters: *None* | *Sequence[float]* | *ndarray* | *Series* = *None*) → float

Order n lower partial moment from -inf to z

Parameters

n: int
Order of partial moment

z: float = 0.0
Upper bound for partial moment integral

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use None for parameterless distributions.

Returns
Partial moment

Return type
float

References

Notes

The order n lower partial moment to z is

$$\int_{-\infty}^{z} x^n f(x) dx$$

See[1] for more details.

[1] Winkler et al. (1972) "The Determination of Partial Moments" *Management Science* Vol. 19 No. 3

arch.univariate.SkewStudent.ppf

SkewStudent.**ppf**(pits: *float* | *Sequence[float]* | *ndarray* | *Series*, parameters: *None* | *Sequence[float]* | *ndarray* | *Series* = ***None***) → float | ndarray

Inverse cumulative density function (ICDF)

Parameters

pits: float | Sequence[float] | ndarray | Series
Probability-integral-transformed values in the interval (0, 1).

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use `None` for parameterless distributions.

Returns
i – Inverse CDF values

Return type
`{float, ndarray}`

arch.univariate.SkewStudent.simulate

SkewStudent.**simulate**(parameters: *int* | *float* | *Sequence[float* | *int]* | *ndarray* | *Series*) → Callable[[int | tuple[int, ...]], ndarray]

Simulates i.i.d. draws from the distribution

Parameters

parameters: int | float | Sequence[float | int] | ndarray | Series
Distribution parameters

Returns
simulator – Callable that take a single output size argument and returns i.i.d. draws from the distribution

Return type
`callable`

arch.univariate.SkewStudent.starting_values

SkewStudent.**starting_values**(std_resid: *ndarray*) → ndarray

Construct starting values for use in optimization.

Parameters

std_resid: ndarray
Estimated standardized residuals to use in computing starting values for the shape parameter

Returns
sv – Array containing starting valuer for shape parameter

Return type
`numpy.ndarray`

Notes

Uses relationship between kurtosis and degree of freedom parameter to produce a moment-based estimator for the starting values.

Properties

generator	The NumPy Generator or RandomState attached to the distribution
name	The name of the distribution
random_state	The NumPy RandomState attached to the distribution

arch.univariate.SkewStudent.generator

property SkewStudent.**generator** : RandomState | Generator

The NumPy Generator or RandomState attached to the distribution

arch.univariate.SkewStudent.name

property SkewStudent.**name** : str

The name of the distribution

arch.univariate.SkewStudent.random_state

property SkewStudent.**random_state** : RandomState | Generator

The NumPy RandomState attached to the distribution

Deprecated since version 5.0: random_state is deprecated. Use generator instead.

1.11.4 arch.univariate.GeneralizedError

class arch.univariate.**GeneralizedError**(random_state: *RandomState | None = **None***, *, seed: *None | int | RandomState | Generator = **None***)

Generalized Error distribution for use with ARCH models

Parameters

random_state: RandomState | None = None

Deprecated since version 5.0: random_state is deprecated. Use seed instead.

seed: None | int | RandomState | Generator = None

Random number generator instance or int to use. Set to ensure reproducibility. If using an int, the argument is passed to `np.random.default_rng`. If not provided, `default_rng` is used with system-provided entropy.

Methods

bounds(resids)	Parameter bounds for use in optimization.
cdf(resids[, parameters])	Cumulative distribution function
constraints()	Construct arrays to use in constrained optimization.
loglikelihood(parameters, resids, sigma2[, ...])	Computes the log-likelihood of assuming residuals are have a Generalized Error Distribution, conditional on the variance.
moment(n[, parameters])	Moment of order n
parameter_names()	Names of distribution shape parameters
partial_moment(n[, z, parameters])	Order n lower partial moment from -inf to z
ppf(pits[, parameters])	Inverse cumulative density function (ICDF)
simulate(parameters)	Simulates i.i.d.
starting_values(std_resid)	Construct starting values for use in optimization.

arch.univariate.GeneralizedError.bounds

GeneralizedError.**bounds**(resids: *ndarray*) → list[tuple[float, float]]

Parameter bounds for use in optimization.

Parameters

resids: ndarray
Residuals to use when computing the bounds

Returns
bounds – List containing a single tuple with (lower, upper) bounds

Return type
`list`

arch.univariate.GeneralizedError.cdf

GeneralizedError.**cdf**(resids: *Sequence[float]* | *ndarray* | *Series*, parameters: *None* | *Sequence[float]* | *ndarray* | *Series* = *None*) → ndarray

Cumulative distribution function

Parameters

resids: Sequence[float] | ndarray | Series
Values at which to evaluate the cdf

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use `None` for parameterless distributions.

Returns
f – CDF values

Return type
`numpy.ndarray`

arch.univariate.GeneralizedError.constraints

GeneralizedError.**constraints**() → tuple[ndarray, ndarray]

Construct arrays to use in constrained optimization.

Returns

- **A** (`numpy.ndarray`) – Constraint loadings
- **b** (`numpy.ndarray`) – Constraint values

Notes

Parameters satisfy the constraints A.dot(parameters)-b >= 0

arch.univariate.GeneralizedError.loglikelihood

GeneralizedError.**loglikelihood**(parameters: *Sequence[float]* | *ndarray* | *Series*, resids: *ndarray* | *DataFrame* | *Series*, sigma2: *ndarray* | *DataFrame* | *Series*, individual: *bool* = ***False***) → ndarray

Computes the log-likelihood of assuming residuals are have a Generalized Error Distribution, conditional on the variance.

Parameters

parameters: Sequence[float] | ndarray | Series
Shape parameter of the GED distribution

resids: ndarray | DataFrame | Series
The residuals to use in the log-likelihood calculation

sigma2: ndarray | DataFrame | Series
Conditional variances of resids

individual: bool = `False`
Flag indicating whether to return the vector of individual log likelihoods (True) or the sum (False)

Returns
ll – The log-likelihood

Return type
`float`

Notes

The log-likelihood of a single data point x is

$$\ln\nu - \ln c - \ln\Gamma(\frac{1}{\nu}) - (1+\frac{1}{\nu})\ln 2 - \frac{1}{2}\ln\sigma^2 - \frac{1}{2}\left|\frac{x}{c\sigma}\right|^{\nu}$$

where Γ is the gamma function and $\ln c$ is

$$\ln c = \frac{1}{2}\left(\frac{-2}{\nu}\ln 2 + \ln\Gamma(\frac{1}{\nu}) - \ln\Gamma(\frac{3}{\nu})\right).$$

arch.univariate.GeneralizedError.moment

GeneralizedError.**moment**(n: *int*, parameters: *None* | *Sequence[float]* | *ndarray* | *Series* = ***None***) → float

Moment of order n

Parameters

n: int
Order of moment

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use None for parameterless distributions.

Returns
Calculated moment

Return type
`float`

arch.univariate.GeneralizedError.parameter_names

GeneralizedError.**parameter_names**() → list[str]

Names of distribution shape parameters

Returns
names – Parameter names

Return type
`list (str)`

arch.univariate.GeneralizedError.partial_moment

GeneralizedError.**partial_moment**(n: *int*, z: *float* = **0.0**, parameters: *None* | *Sequence[float]* | *ndarray* | *Series* = ***None***) → float

Order n lower partial moment from -inf to z

Parameters

n: int
Order of partial moment

z: float = 0.0
Upper bound for partial moment integral

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use None for parameterless distributions.

Returns
Partial moment

Return type
`float`

References

Notes

The order n lower partial moment to z is

$$\int_{-\infty}^{z} x^n f(x)dx$$

See[1] for more details.

arch.univariate.GeneralizedError.ppf

GeneralizedError.**ppf**(pits: *float | Sequence[float] | ndarray | Series*, parameters: *None | Sequence[float] | ndarray | Series = None*) → ndarray

Inverse cumulative density function (ICDF)

Parameters

pits: float | Sequence[float] | ndarray | Series
Probability-integral-transformed values in the interval (0, 1).

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use `None` for parameterless distributions.

Returns
i – Inverse CDF values

Return type
`{float, ndarray}`

arch.univariate.GeneralizedError.simulate

GeneralizedError.**simulate**(parameters: *int | float | Sequence[float | int] | ndarray | Series*) → Callable[[int | tuple[int, ...]], ndarray]

Simulates i.i.d. draws from the distribution

Parameters

parameters: int | float | Sequence[float | int] | ndarray | Series
Distribution parameters

Returns
simulator – Callable that take a single output size argument and returns i.i.d. draws from the distribution

Return type
`callable`

[1] Winkler et al. (1972) "The Determination of Partial Moments" *Management Science* Vol. 19 No. 3

arch.univariate.GeneralizedError.starting_values

GeneralizedError.**starting_values**(std_resid: *ndarray*) → ndarray

Construct starting values for use in optimization.

Parameters

std_resid: ndarray
Estimated standardized residuals to use in computing starting values for the shape parameter

Returns
sv – Array containing starting valuer for shape parameter

Return type
`numpy.ndarray`

Notes

Defaults to 1.5 which is implies heavier tails than a normal

Properties

generator	The NumPy Generator or RandomState attached to the distribution
name	The name of the distribution
random_state	The NumPy RandomState attached to the distribution

arch.univariate.GeneralizedError.generator

property GeneralizedError.**generator** : RandomState | Generator

The NumPy Generator or RandomState attached to the distribution

arch.univariate.GeneralizedError.name

property GeneralizedError.**name** : str

The name of the distribution

arch.univariate.GeneralizedError.random_state

property GeneralizedError.**random_state** : RandomState | Generator

The NumPy RandomState attached to the distribution

Deprecated since version 5.0: random_state is deprecated. Use generator instead.

1.11.5 Writing New Distributions

All distributions must inherit from :class:Distribution and provide all public methods.

Distribution([random_state, seed])	Template for subclassing only

arch.univariate.distribution.Distribution

class arch.univariate.distribution.**Distribution**(*random_state: RandomState* | *None* = **None**, *, *seed: None* | *int* | *RandomState* | *Generator* = **None**)

Template for subclassing only

Methods

bounds(resids)	Parameter bounds for use in optimization.
cdf(resids[, parameters])	Cumulative distribution function
constraints()	Construct arrays to use in constrained optimization.
loglikelihood(parameters, resids, sigma2[, ...])	Loglikelihood evaluation.
moment(n[, parameters])	Moment of order n
parameter_names()	Names of distribution shape parameters
partial_moment(n[, z, parameters])	Order n lower partial moment from -inf to z
ppf(pits[, parameters])	Inverse cumulative density function (ICDF)
simulate(parameters)	Simulates i.i.d.
starting_values(std_resid)	Construct starting values for use in optimization.

arch.univariate.distribution.Distribution.bounds

abstract Distribution.**bounds**(resids: *ndarray*) → list[tuple[float, float]]

Parameter bounds for use in optimization.

Parameters

resids: ndarray
Residuals to use when computing the bounds

Returns
bounds – List containing a single tuple with (lower, upper) bounds

Return type
list

arch.univariate.distribution.Distribution.cdf

abstract `Distribution.cdf`(resids: *Sequence[float] | ndarray | Series*, parameters: *None | Sequence[float] | ndarray | Series* = `None`) → ndarray

Cumulative distribution function

Parameters

resids: Sequence[float] | ndarray | Series
Values at which to evaluate the cdf

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use `None` for parameterless distributions.

Returns
f – CDF values

Return type
`numpy.ndarray`

arch.univariate.distribution.Distribution.constraints

abstract `Distribution.constraints()` → tuple[ndarray, ndarray]

Construct arrays to use in constrained optimization.

Returns

- **A** (`numpy.ndarray`) – Constraint loadings
- **b** (`numpy.ndarray`) – Constraint values

Notes

Parameters satisfy the constraints A.dot(parameters)-b >= 0

arch.univariate.distribution.Distribution.loglikelihood

abstract `Distribution.loglikelihood`(parameters: *Sequence[float] | ndarray | Series*, resids: *ndarray | DataFrame | Series*, sigma2: *ndarray | DataFrame | Series*, individual: *bool* = `False`) → float | ndarray

Loglikelihood evaluation.

Parameters

parameters: Sequence[float] | ndarray | Series
Distribution shape parameters

resids: ndarray | DataFrame | Series
nobs array of model residuals

sigma2: ndarray | DataFrame | Series
nobs array of conditional variances

individual: bool = False
Flag indicating whether to return the vector of individual log likelihoods (True) or the sum (False)

Notes

Returns the loglikelihood where resids are the "data", and parameters and sigma2 are inputs.

arch.univariate.distribution.Distribution.moment

`abstract Distribution.moment(n: int, parameters: None | Sequence[float] | ndarray | Series = None) → float`

Moment of order n

Parameters

n: int
Order of moment

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use None for parameterless distributions.

Returns
Calculated moment

Return type
`float`

arch.univariate.distribution.Distribution.parameter_names

`abstract Distribution.parameter_names() → list[str]`

Names of distribution shape parameters

Returns
names – Parameter names

Return type
`list (str)`

arch.univariate.distribution.Distribution.partial_moment

`abstract Distribution.partial_moment(n: int, z: float = 0.0, parameters: None | Sequence[float] | ndarray | Series = None) → float`

Order n lower partial moment from -inf to z

Parameters

n: int
Order of partial moment

z: float = 0.0
Upper bound for partial moment integral

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use None for parameterless distributions.

Returns
Partial moment

Return type
`float`

References

Notes

The order n lower partial moment to z is

$$\int_{-\infty}^{z} x^n f(x)dx$$

See[1] for more details.

arch.univariate.distribution.Distribution.ppf

abstract Distribution.**ppf**(pits: *float* | *Sequence[float]* | *ndarray* | *Series*, parameters: *None* | *Sequence[float]* | *ndarray* | *Series* = **None**) → float | ndarray

Inverse cumulative density function (ICDF)

Parameters

pits: float | Sequence[float] | ndarray | Series
Probability-integral-transformed values in the interval (0, 1).

parameters: None | Sequence[float] | ndarray | Series = None
Distribution parameters. Use `None` for parameterless distributions.

Returns
i – Inverse CDF values

Return type
`{float, ndarray}`

arch.univariate.distribution.Distribution.simulate

abstract Distribution.**simulate**(parameters: *int* | *float* | *Sequence[float* | *int]* | *ndarray* | *Series*) → Callable[[int | tuple[int, ...]], ndarray]

Simulates i.i.d. draws from the distribution

Parameters

parameters: int | float | Sequence[float | int] | ndarray | Series
Distribution parameters

Returns
simulator – Callable that take a single output size argument and returns i.i.d. draws from the distribution

Return type
`callable`

[1] Winkler et al. (1972) "The Determination of Partial Moments" *Management Science* Vol. 19 No. 3

arch.univariate.distribution.Distribution.starting_values

abstract Distribution.**starting_values**(std_resid: *ndarray*) → ndarray

Construct starting values for use in optimization.

Parameters

std_resid: ndarray
Estimated standardized residuals to use in computing starting values for the shape parameter

Returns
sv – The estimated shape parameters for the distribution

Return type
numpy.ndarray

Notes

Size of sv depends on the distribution

Properties

generator	The NumPy Generator or RandomState attached to the distribution
name	The name of the distribution
random_state	The NumPy RandomState attached to the distribution

arch.univariate.distribution.Distribution.generator

property Distribution.**generator** : RandomState | Generator

The NumPy Generator or RandomState attached to the distribution

arch.univariate.distribution.Distribution.name

property Distribution.**name** : str

The name of the distribution

arch.univariate.distribution.Distribution.random_state

property Distribution.**random_state** : RandomState | Generator

The NumPy RandomState attached to the distribution

Deprecated since version 5.0: random_state is deprecated. Use generator instead.

1.12 Model Results

All model return the same object, a results class (ARCHModelResult). When using the fix method, a (ARCHModelFixedResult) is produced that lacks some properties of a (ARCHModelResult) that are not relevant when parameters are not estimated.

ARCHModelResult(params, param_cov, r2, ...)	Results from estimation of an ARCHModel model
ARCHModelFixedResult(params, resid, ...)	Results for fixed parameters for an ARCHModel model

1.12.1 arch.univariate.base.ARCHModelResult

class arch.univariate.base.**ARCHModelResult**(params: *ndarray*, param_cov: *ndarray* | *None*, r2: *float*, resid: *ndarray*, volatility: *ndarray*, cov_type: *str*, dep_var: *pandas.Series*, names: *list[str]*, loglikelihood: *float*, is_pandas: *bool*, optim_output: *OptimizeResult*, fit_start: *int*, fit_stop: *int*, model: ARCHModel)

Results from estimation of an ARCHModel model

Parameters

params: ndarray
Estimated parameters

param_cov: ndarray | None
Estimated variance-covariance matrix of params. If none, calls method to compute variance from model when parameter covariance is first used from result

r2: float
Model R-squared

resid: ndarray
Residuals from model. Residuals have same shape as original data and contain nan-values in locations not used in estimation

volatility: ndarray
Conditional volatility from model

cov_type: str
String describing the covariance estimator used

dep_var: pandas.Series
Dependent variable

names: list[str]
Model parameter names

loglikelihood: float
Loglikelihood at estimated parameters

is_pandas: bool
Whether the original input was pandas

optim_output: OptimizeResult
Result of log-likelihood optimization

fit_start: int
Integer index of the first observation used to fit the model

fit_stop: int
Integer index of the last observation used to fit the model using slice notation *fit_start:fit_stop*

model: *ARCHModel*
The model object used to estimate the parameters

Methods

arch_lm_test([lags, standardized])	ARCH LM test for conditional heteroskedasticity
conf_int([alpha])	Parameter confidence intervals
forecast([params, horizon, start, align, ...])	Construct forecasts from estimated model
hedgehog_plot([params, horizon, step, ...])	Plot forecasts from estimated model
plot([annualize, scale])	Plot standardized residuals and conditional volatility
summary()	Constructs a summary of the results from a fit model.

arch.univariate.base.ARCHModelResult.arch_lm_test

`ARCHModelResult.`**`arch_lm_test`**(lags: *int* | *None* = **None**, standardized: *bool* = **False**) → *WaldTestStatistic*

ARCH LM test for conditional heteroskedasticity

Parameters

lags: int | None = None
Number of lags to include in the model. If not specified,

standardized: bool = False
Flag indicating to test the model residuals divided by their conditional standard deviations. If False, directly tests the estimated residuals.

Returns
result – Result of ARCH-LM test

Return type
`WaldTestStatistic`

arch.univariate.base.ARCHModelResult.conf_int

`ARCHModelResult.`**`conf_int`**(alpha: *float* = **0.05**) → pd.DataFrame

Parameter confidence intervals

Parameters

alpha: float = 0.05
Size (prob.) to use when constructing the confidence interval.

Returns
ci – Array where the ith row contains the confidence interval for the ith parameter

Return type
`pandas.DataFrame`

arch.univariate.base.ARCHModelResult.forecast

ARCHModelResult.**forecast**(params: *ArrayLike1D | None* = **None**, horizon: *int* = **1**, start: *int | DateLike | None* = **None**, align: *'origin' | 'target'* = 'origin', method: *ForecastingMethod* = 'analytic', simulations: *int* = **1000**, rng: *collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None* = **None**, random_state: *np.random.RandomState | None* = **None**, *, reindex: *bool* = **False**, x: *None | dict[Label, ArrayLike] | ArrayLike* = **None**) → *ARCHModelForecast*

Construct forecasts from estimated model

Parameters

params: ArrayLike1D | None = None
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 = 1
Number of steps to forecast

start: int | DateLike | None = None
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' = 'origin'
Either 'origin' or 'target'. When set of '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' simplified computing forecast errors since the realization and h-step forecast are aligned.

method: ForecastingMethod = 'analytic'
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 = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None = None
Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

random_state: np.random.RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

reindex: bool = False

Whether to reindex the forecasts to have the same dimension as the series being forecast.

Changed in version 6.2: The default has been changed to False.

x: None | dict[Label, ArrayLike] | ArrayLike = None

Values to use for exogenous regressors if any are included in the model. Three formats are accepted:

- 2-d array-like: This format can be used when there is a single exogenous variable. The input must have shape (nforecast, horizon) or (nobs, horizon) where nforecast is the number of forecasting periods and nobs is the original shape of y. For example, if a single series of forecasts are made from the end of the sample with a horizon of 10, then the input can be (1, 10). Alternatively, if the original data had 1000 observations, then the input can be (1000, 10), and only the final row is used to produce forecasts.

- A dictionary of 2-d array-like: This format is identical to the previous except that the dictionary keys must match the names of the exog variables. Requires that the exog variables were pass as a pandas DataFrame.
- A 3-d NumPy array (or equivalent). In this format, each panel (0th axis) is a 2-d array that must have shape (nforecast, horizon) or (nobs,horizon). The array x[j] corresponds to the j-th column of the exogenous variables.

Due to the complexity required to accommodate all scenarios, please see the example notebook that demonstrates the valid formats for x.

New in version 4.19.

Returns

Container for forecasts. Key properties are `mean`, `variance` and `residual_variance`.

Return type

arch.univariate.base.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.

arch.univariate.base.ARCHModelResult.hedgehog_plot

ARCHModelResult.**hedgehog_plot**(params: *ndarray | Series | None* = **None**, horizon: *int* = **10**, step: *int* = **10**, start: *int | str | datetime | datetime64 | Timestamp | None* = **None**, plot_type: *'volatility' | 'mean'* = `'volatility'`, method: *'analytic' | 'simulation' | 'bootstrap'* = `'analytic'`, simulations: *int* = **1000**) → matplotlib.figure.Figure

Plot forecasts from estimated model

Parameters

params: ndarray | Series | None = None

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 = 10

Number of steps to forecast

step: int = 10
Non-negative number of forecasts to skip between spines

start: int | str | datetime | datetime64 | Timestamp | None = None
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.

plot_type: 'volatility' | 'mean' = 'volatility'
Quantity to plot, the forecast volatility or the forecast mean

method: 'analytic' | 'simulation' | 'bootstrap' = 'analytic'
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 = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

Returns
fig – Handle to the figure

Return type
`figure`

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],  vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot(plot_type='mean')
```

arch.univariate.base.ARCHModelResult.plot

`ARCHModelResult.plot`(annualize: *str* | *None* = ***None***, scale: *float* | *None* = ***None***) → matplotlib.figure.Figure

Plot standardized residuals and conditional volatility

Parameters

annualize: str | None = None
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 | None = None
Value to use when scaling returns to annualize. If scale is provided, annualize is ignored and the value in scale is used.

Returns
fig – Handle to the figure

Return type
`figure`

Examples

```
>>> from arch import arch_model
>>> am = arch_model(None)
>>> sim_data = am.simulate([0.0, 0.01, 0.07, 0.92], 2520)
>>> am = arch_model(sim_data['data'])
>>> res = am.fit(update_freq=0, disp='off')
>>> fig = res.plot()
```

Produce a plot with annualized volatility

```
>>> fig = res.plot(annualize='D')
```

Override the usual scale of 252 to use 360 for an asset that trades most days of the year

```
>>> fig = res.plot(scale=360)
```

arch.univariate.base.ARCHModelResult.summary

`ARCHModelResult.summary()` → Summary

Constructs a summary of the results from a fit model.

Returns
summary – Object that contains tables and facilitated export to text, html or latex

Return type
`Summary instance`

Properties

aic	Akaike Information Criteria
bic	Schwarz/Bayesian Information Criteria
conditional_volatility	Estimated conditional volatility
convergence_flag	scipy.optimize.minimize result flag
fit_start	Start of sample used to estimate parameters
fit_stop	End of sample used to estimate parameters
loglikelihood	Model loglikelihood
model	Model instance used to produce the fit
nobs	Number of data points used to estimate model
num_params	Number of parameters in model
optimization_result	Information about the convergence of the loglikelihood optimization
param_cov	Parameter covariance
params	Model Parameters
pvalues	Array of p-values for the t-statistics
resid	Model residuals
rsquared	R-squared
rsquared_adj	Degree of freedom adjusted R-squared
scale	The scale applied to the original data before estimating the model.
std_err	Array of parameter standard errors
std_resid	Residuals standardized by conditional volatility
tvalues	Array of t-statistics testing the null that the coefficient are 0

arch.univariate.base.ARCHModelResult.aic

property ARCHModelResult.**aic** : float

Akaike Information Criteria

-2 * loglikelihood + 2 * num_params

arch.univariate.base.ARCHModelResult.bic

property ARCHModelResult.**bic** : float

Schwarz/Bayesian Information Criteria

-2 * loglikelihood + log(nobs) * num_params

arch.univariate.base.ARCHModelResult.conditional_volatility

property ARCHModelResult.**conditional_volatility** : pd.Series | Float64Array

Estimated conditional volatility

Returns

conditional_volatility – nobs element array containing the conditional volatility (square root of conditional variance). The values are aligned with the input data so that the value in the t-th position is the variance of t-th error, which is computed using time-(t-1) information.

Return type
`{ndarray, Series}`

arch.univariate.base.ARCHModelResult.convergence_flag

property `ARCHModelResult.convergence_flag` : int

scipy.optimize.minimize result flag

arch.univariate.base.ARCHModelResult.fit_start

property `ARCHModelResult.fit_start` : int

Start of sample used to estimate parameters

arch.univariate.base.ARCHModelResult.fit_stop

property `ARCHModelResult.fit_stop` : int

End of sample used to estimate parameters

arch.univariate.base.ARCHModelResult.loglikelihood

property `ARCHModelResult.loglikelihood` : float

Model loglikelihood

arch.univariate.base.ARCHModelResult.model

property `ARCHModelResult.model` : *ARCHModel*

Model instance used to produce the fit

arch.univariate.base.ARCHModelResult.nobs

property `ARCHModelResult.nobs` : int

Number of data points used to estimate model

arch.univariate.base.ARCHModelResult.num_params

property `ARCHModelResult.num_params` : int

Number of parameters in model

arch.univariate.base.ARCHModelResult.optimization_result

property `ARCHModelResult.optimization_result` : OptimizeResult

Information about the convergence of the loglikelihood optimization

Returns

optim_result – Result from numerical optimization of the log-likelihood.

Return type

`OptimizeResult`

arch.univariate.base.ARCHModelResult.param_cov

property `ARCHModelResult.param_cov` : pd.DataFrame

Parameter covariance

arch.univariate.base.ARCHModelResult.params

property `ARCHModelResult.params` : pandas.Series

Model Parameters

arch.univariate.base.ARCHModelResult.pvalues

property `ARCHModelResult.pvalues` : pandas.Series

Array of p-values for the t-statistics

arch.univariate.base.ARCHModelResult.resid

property `ARCHModelResult.resid` : Float64Array | pd.Series

Model residuals

arch.univariate.base.ARCHModelResult.rsquared

property `ARCHModelResult.rsquared` : float

R-squared

arch.univariate.base.ARCHModelResult.rsquared_adj

property `ARCHModelResult.rsquared_adj` : float

Degree of freedom adjusted R-squared

arch.univariate.base.ARCHModelResult.scale

property `ARCHModelResult.scale` : float

The scale applied to the original data before estimating the model.

If scale=1.0, the the data have not been rescaled. Otherwise, the model parameters have been estimated on scale * y.

arch.univariate.base.ARCHModelResult.std_err

property `ARCHModelResult.std_err` : pandas.Series

Array of parameter standard errors

arch.univariate.base.ARCHModelResult.std_resid

property `ARCHModelResult.std_resid` : Float64Array | pd.Series

Residuals standardized by conditional volatility

arch.univariate.base.ARCHModelResult.tvalues

property `ARCHModelResult.tvalues` : pandas.Series

Array of t-statistics testing the null that the coefficient are 0

1.12.2 arch.univariate.base.ARCHModelFixedResult

class `arch.univariate.base.ARCHModelFixedResult`(params: *ndarray*, resid: *ndarray*, volatility: *ndarray*, dep_var: *pandas.Series*, names: *list[str]*, loglikelihood: *float*, is_pandas: *bool*, model: ARCHModel)

Results for fixed parameters for an ARCHModel model

Parameters

params: ndarray
Estimated parameters

resid: ndarray
Residuals from model. Residuals have same shape as original data and contain nan-values in locations not used in estimation

volatility: ndarray
Conditional volatility from model

dep_var: pandas.Series
Dependent variable

names: list[str]
Model parameter names

loglikelihood: float
Loglikelihood at specified parameters

is_pandas: bool
Whether the original input was pandas

model: *ARCHModel*
The model object used to estimate the parameters

Methods

`arch_lm_test`([lags, standardized])	ARCH LM test for conditional heteroskedasticity
`forecast`([params, horizon, start, align, ...])	Construct forecasts from estimated model
`hedgehog_plot`([params, horizon, step, ...])	Plot forecasts from estimated model
`plot`([annualize, scale])	Plot standardized residuals and conditional volatility
`summary`()	Constructs a summary of the results from a fit model.

arch.univariate.base.ARCHModelFixedResult.arch_lm_test

`ARCHModelFixedResult.arch_lm_test`(lags: *int* | *None* = **None**, standardized: *bool* = **False**) → *WaldTestStatistic*

ARCH LM test for conditional heteroskedasticity

Parameters

lags: int | None = None
Number of lags to include in the model. If not specified,

standardized: bool = False
Flag indicating to test the model residuals divided by their conditional standard deviations. If False, directly tests the estimated residuals.

Returns
result – Result of ARCH-LM test

Return type
`WaldTestStatistic`

arch.univariate.base.ARCHModelFixedResult.forecast

`ARCHModelFixedResult.forecast`(params: *ArrayLike1D* | *None* = **None**, horizon: *int* = **1**, start: *int* | *DateLike* | *None* = **None**, align: *'origin'* | *'target'* = `'origin'`, method: *ForecastingMethod* = `'analytic'`, simulations: *int* = **1000**, rng: *collections.abc.Callable[[int* | *tuple[int, ...]], Float64Array]* | *None* = **None**, random_state: *np.random.RandomState* | *None* = **None**, *, reindex: *bool* = **False**, x: *None* | *dict[Label, ArrayLike]* | *ArrayLike* = **None**) → *ARCHModelForecast*

Construct forecasts from estimated model

Parameters

params: ArrayLike1D | None = None
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 = 1
Number of steps to forecast

start: int | DateLike | None = None
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' = 'origin'
Either 'origin' or 'target'. When set of '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' simplified computing forecast errors since the realization and h-step forecast are aligned.

method: ForecastingMethod = 'analytic'
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 = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

rng: collections.abc.Callable[[int | tuple[int, ...]], Float64Array] | None = None
Custom random number generator to use in simulation-based forecasts. Must produce random samples using the syntax *rng(size)* where size the 2-element tuple (simulations, horizon).

random_state: np.random.RandomState | None = None
NumPy RandomState instance to use when method is 'bootstrap'

reindex: bool = False

Whether to reindex the forecasts to have the same dimension as the series being forecast.

Changed in version 6.2: The default has been changed to False.

x: None | dict[Label, ArrayLike] | ArrayLike = None

Values to use for exogenous regressors if any are included in the model. Three formats are accepted:

- 2-d array-like: This format can be used when there is a single exogenous variable. The input must have shape (nforecast, horizon) or (nobs, horizon) where nforecast is the number of forecasting periods and nobs is the original shape of y. For example, if a single series of forecasts are made from the end of the sample with a horizon of 10, then the input can be (1, 10). Alternatively, if the original data had 1000 observations, then the input can be (1000, 10), and only the final row is used to produce forecasts.
- A dictionary of 2-d array-like: This format is identical to the previous except that the dictionary keys must match the names of the exog variables. Requires that the exog variables were pass as a pandas DataFrame.
- A 3-d NumPy array (or equivalent). In this format, each panel (0th axis) is a 2-d array that must have shape (nforecast, horizon) or (nobs,horizon). The array x[j] corresponds to the j-th column of the exogenous variables.

Due to the complexity required to accommodate all scenarios, please see the example notebook that demonstrates the valid formats for x.

New in version 4.19.

Returns
Container for forecasts. Key properties are `mean`, `variance` and `residual_variance`.

Return type
arch.univariate.base.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.

arch.univariate.base.ARCHModelFixedResult.hedgehog_plot

ARCHModelFixedResult.**hedgehog_plot**(params: *ndarray | Series | None* = **None**, horizon: *int* = **10**, step: *int* = **10**, start: *int | str | datetime | datetime64 | Timestamp | None* = **None**, plot_type: *'volatility' | 'mean'* = 'volatility', method: *'analytic' | 'simulation' | 'bootstrap'* = 'analytic', simulations: *int* = **1000**) → matplotlib.figure.Figure

Plot forecasts from estimated model

Parameters

params: ndarray | Series | None = None
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 = 10
Number of steps to forecast

step: int = 10
Non-negative number of forecasts to skip between spines

start: int | str | datetime | datetime64 | Timestamp | None = None
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.

plot_type: 'volatility' | 'mean' = 'volatility'
Quantity to plot, the forecast volatility or the forecast mean

method: 'analytic' | 'simulation' | 'bootstrap' = 'analytic'
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 = 1000
Number of simulations to run when computing the forecast using either simulation or bootstrap.

Returns
fig – Handle to the figure

Return type
`figure`

Examples

```
>>> import pandas as pd
>>> from arch import arch_model
>>> am = arch_model(None,mean='HAR',lags=[1,5,22],vol='Constant')
>>> sim_data = am.simulate([0.1,0.4,0.3,0.2,1.0], 250)
>>> sim_data.index = pd.date_range('2000-01-01',periods=250)
>>> am = arch_model(sim_data['data'],mean='HAR',lags=[1,5,22],  vol='Constant')
>>> res = am.fit()
>>> fig = res.hedgehog_plot(plot_type='mean')
```

arch.univariate.base.ARCHModelFixedResult.plot

ARCHModelFixedResult.**plot**(annualize: *str* | *None* = ***None***, scale: *float* | *None* = ***None***) → matplotlib.figure.Figure

Plot standardized residuals and conditional volatility

Parameters

annualize: str | None = None
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 | None = None
Value to use when scaling returns to annualize. If scale is provided, annualize is ignored and the value in scale is used.

Returns
fig – Handle to the figure

Return type
`figure`

Examples

```
>>> from arch import arch_model
>>> am = arch_model(None)
>>> sim_data = am.simulate([0.0, 0.01, 0.07, 0.92], 2520)
>>> am = arch_model(sim_data['data'])
>>> res = am.fit(update_freq=0, disp='off')
>>> fig = res.plot()
```

Produce a plot with annualized volatility

```
>>> fig = res.plot(annualize='D')
```

Override the usual scale of 252 to use 360 for an asset that trades most days of the year

```
>>> fig = res.plot(scale=360)
```

arch.univariate.base.ARCHModelFixedResult.summary

ARCHModelFixedResult.**summary**() → Summary

Constructs a summary of the results from a fit model.

Returns
summary – Object that contains tables and facilitated export to text, html or latex

Return type
`Summary instance`

Properties

aic	Akaike Information Criteria
bic	Schwarz/Bayesian Information Criteria
conditional_volatility	Estimated conditional volatility
loglikelihood	Model loglikelihood
model	Model instance used to produce the fit
nobs	Number of data points used to estimate model
num_params	Number of parameters in model
params	Model Parameters
resid	Model residuals
std_resid	Residuals standardized by conditional volatility

arch.univariate.base.ARCHModelFixedResult.aic

property ARCHModelFixedResult.**aic** : float

Akaike Information Criteria

-2 * loglikelihood + 2 * num_params

arch.univariate.base.ARCHModelFixedResult.bic

property ARCHModelFixedResult.**bic** : float

Schwarz/Bayesian Information Criteria

-2 * loglikelihood + log(nobs) * num_params

arch.univariate.base.ARCHModelFixedResult.conditional_volatility

property `ARCHModelFixedResult.conditional_volatility` : pd.Series | Float64Array

Estimated conditional volatility

Returns

conditional_volatility – nobs element array containing the conditional volatility (square root of conditional variance). The values are aligned with the input data so that the value in the t-th position is the variance of t-th error, which is computed using time-(t-1) information.

Return type

`{ndarray, Series}`

arch.univariate.base.ARCHModelFixedResult.loglikelihood

property `ARCHModelFixedResult.loglikelihood` : float

Model loglikelihood

arch.univariate.base.ARCHModelFixedResult.model

property `ARCHModelFixedResult.model` : *ARCHModel*

Model instance used to produce the fit

arch.univariate.base.ARCHModelFixedResult.nobs

property `ARCHModelFixedResult.nobs` : int

Number of data points used to estimate model

arch.univariate.base.ARCHModelFixedResult.num_params

property `ARCHModelFixedResult.num_params` : int

Number of parameters in model

arch.univariate.base.ARCHModelFixedResult.params

property `ARCHModelFixedResult.params` : pandas.Series

Model Parameters

arch.univariate.base.ARCHModelFixedResult.resid

property `ARCHModelFixedResult.resid` : Float64Array | pd.Series

Model residuals

arch.univariate.base.ARCHModelFixedResult.std_resid

property `ARCHModelFixedResult.std_resid` : Float64Array | pd.Series

Residuals standardized by conditional volatility

1.13 Utilities

Utilities that do not fit well on other pages.

1.13.1 Test Results

class `arch.utility.testing.WaldTestStatistic(stat: float, df: int, null: str, alternative: str, name: str = '')`

Test statistic holder for Wald-type tests

Parameters

stat: float
The test statistic

df: int
Degree of freedom.

null: str
A statement of the test's null hypothesis

alternative: str
A statement of the test's alternative hypothesis

name: str = ''
Name of test

property `critical_values` : dict[str, float]

Critical values test for common test sizes

property `null` : str

Null hypothesis

property `pval` : float

P-value of test statistic

property `stat` : float

Test statistic

1.14 Theoretical Background

To be completed

CHAPTER

TWO

BOOTSTRAPPING

The bootstrap module provides both high- and low-level interfaces for bootstrapping data contained in NumPy arrays or pandas Series or DataFrames.

All bootstraps have the same interfaces and only differ in their name, setup parameters and the (internally generated) sampling scheme.

2.1 Bootstrap Examples

This setup code is required to run in an IPython notebook

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn

seaborn.set_style("darkgrid")
plt.rc("figure", figsize=(16, 6))
plt.rc("savefig", dpi=90)
plt.rc("font", family="sans-serif")
plt.rc("font", size=14)
```

2.1.1 Sharpe Ratio

The Sharpe Ratio is an important measure of return per unit of risk. The example shows how to estimate the variance of the Sharpe Ratio and how to construct confidence intervals for the Sharpe Ratio using a long series of U.S. equity data.

[2]:
```
import arch.data.frenchdata
import numpy as np
import pandas as pd

ff = arch.data.frenchdata.load()
```

The data set contains the Fama-French factors, including the excess market return.

[3]:
```
excess_market = ff.iloc[:, 0]
print(ff.describe())
```

```
          Mkt-RF          SMB          HML           RF
count  1109.000000  1109.000000  1109.000000  1109.000000
mean      0.659946     0.206555     0.368864     0.274220
std       5.327524     3.191132     3.482352     0.253377
min     -29.130000   -16.870000   -13.280000    -0.060000
25%      -1.970000    -1.560000    -1.320000     0.030000
50%       1.020000     0.070000     0.140000     0.230000
75%       3.610000     1.730000     1.740000     0.430000
max      38.850000    36.700000    35.460000     1.350000
```

The next step is to construct a function that computes the Sharpe Ratio. This function also return the annualized mean and annualized standard deviation which will allow the covariance matrix of these parameters to be estimated using the bootstrap.

[4]:
```
def sharpe_ratio(x):
    mu, sigma = 12 * x.mean(), np.sqrt(12 * x.var())
    values = np.array([mu, sigma, mu / sigma]).squeeze()
    index = ["mu", "sigma", "SR"]
    return pd.Series(values, index=index)
```

The function can be called directly on the data to show full sample estimates.

[5]:
```
params = sharpe_ratio(excess_market)
params
```

[5]:
```
mu        7.919351
sigma    18.455084
SR        0.429115
dtype: float64
```

2.1.2 Reproducibility

All bootstraps accept the keyword argument `seed` which can contain a NumPy `Generator` or `RandomState` or an `int`. When using an `int`, the argument is passed `np.random.default_rng` to create the core generator. This allows the same pseudo random values to be used across multiple runs.

Warning

The bootstrap chosen must be appropriate for the data. Squared returns are serially correlated, and so a time-series bootstrap is required.

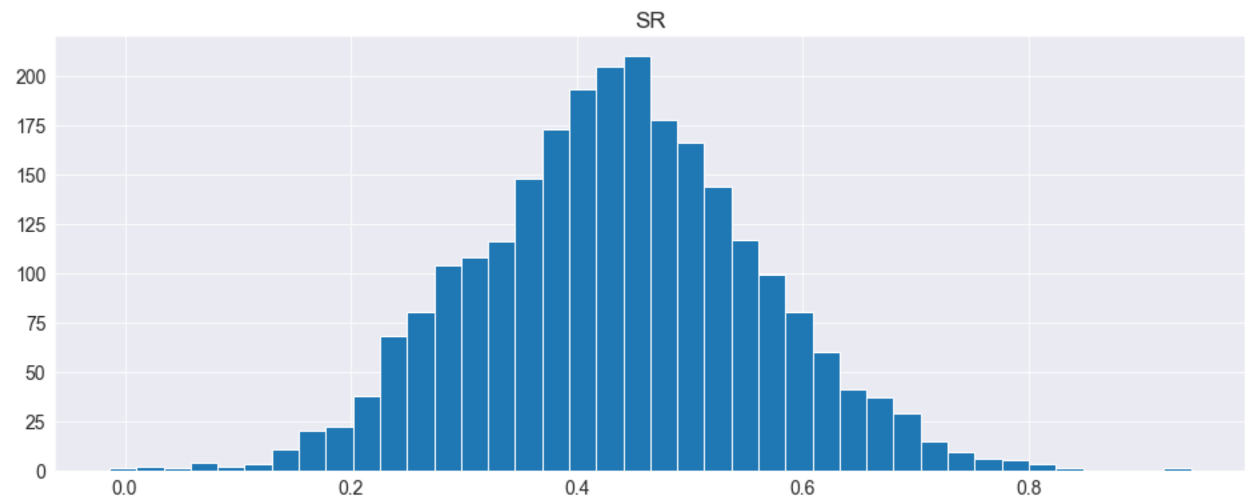
Bootstraps are initialized with any bootstrap specific parameters and the data to be used in the bootstrap. Here the `12` is the average window length in the Stationary Bootstrap, and the next input is the data to be bootstrapped.

[6]:
```
from arch.bootstrap import StationaryBootstrap

# Initialize with entropy from random.org
entropy = [877788388, 418255226, 989657335, 69307515]
seed = np.random.default_rng(entropy)

bs = StationaryBootstrap(12, excess_market, seed=seed)
results = bs.apply(sharpe_ratio, 2500)
```

(continues on next page)

(continued from previous page)

```
SR = pd.DataFrame(results[:, -1:], columns=["SR"])
fig = SR.hist(bins=40)
```



[7]:
```
cov = bs.cov(sharpe_ratio, 1000)
cov = pd.DataFrame(cov, index=params.index, columns=params.index)
print(cov)
se = pd.Series(np.sqrt(np.diag(cov)), index=params.index)
se.name = "Std Errors"
print("\n")
print(se)
```

```
             mu     sigma        SR
mu     3.837196 -0.638431  0.224722
sigma -0.638431  3.019569 -0.105762
SR     0.224722 -0.105762  0.014915


mu       1.958876
sigma    1.737691
SR       0.122126
Name: Std Errors, dtype: float64
```

[8]:
```
ci = bs.conf_int(sharpe_ratio, 1000, method="basic")
ci = pd.DataFrame(ci, index=["Lower", "Upper"], columns=params.index)
print(ci)
```

```
              mu      sigma        SR
Lower   4.367662  14.780547  0.166759
Upper  11.958503  21.735752  0.659350
```

Alternative confidence intervals can be computed using a variety of methods. Setting `reuse=True` allows the previous bootstrap results to be used when constructing confidence intervals using alternative methods.

[9]:
```
ci = bs.conf_int(sharpe_ratio, 1000, method="percentile", reuse=True)
ci = pd.DataFrame(ci, index=["Lower", "Upper"], columns=params.index)
print(ci)
```

```
             mu      sigma        SR
Lower   3.880198  15.174416  0.198880
Upper  11.471040  22.129620  0.691471
```

Optimal Block Length Estimation

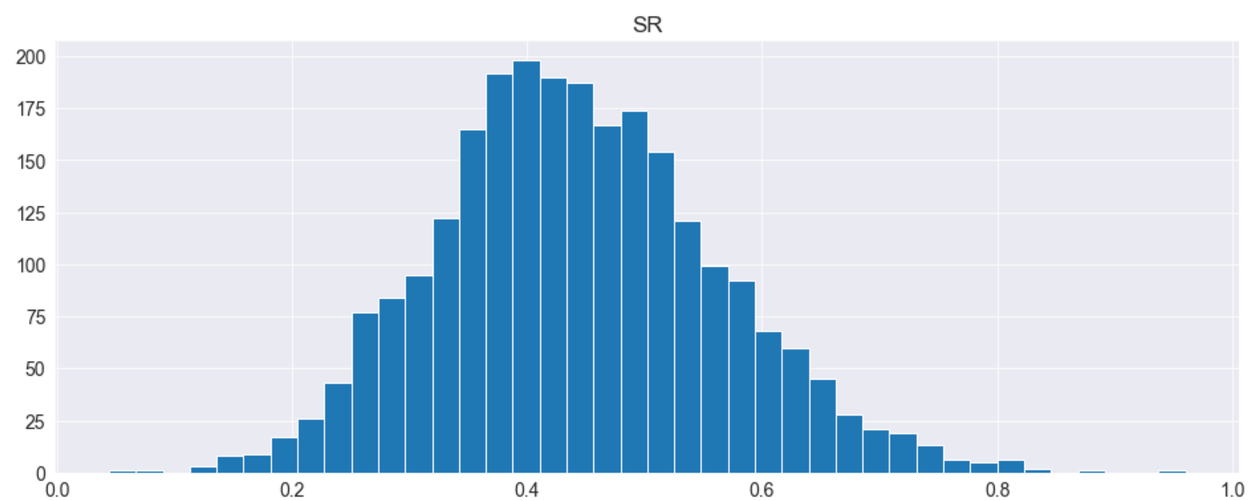
The function `optimal_block_length` can be used to estimate the optimal block lengths for the Stationary and Circular bootstraps. Here we use the squared market return since the Sharpe ratio depends on the mean and the variance, and the autocorrelation in the squares is stronger than in the returns.

[10]:
```
from arch.bootstrap import optimal_block_length

opt = optimal_block_length(excess_market**2)
print(opt)
```

```
        stationary   circular
Mkt-RF   47.766787  54.679322
```

We can repeat the analysis above using the estimated optimal block length. Here we see that the extremes appear to be slightly more extreme.

[11]:
```
# Reinitialize using the same entropy
rs = np.random.default_rng(entropy)

bs = StationaryBootstrap(opt.loc["Mkt-RF", "stationary"], excess_market, seed=seed)
results = bs.apply(sharpe_ratio, 2500)
SR = pd.DataFrame(results[:, -1:], columns=["SR"])
fig = SR.hist(bins=40)
```



2.1.3 Probit (statsmodels)

The second example makes use of a Probit model from statsmodels. The demo data is university admissions data which contains a binary variable for being admitted, GRE score, GPA score and quartile rank. This data is downloaded from the internet and imported using pandas.

[12]:
```
import arch.data.binary

binary = arch.data.binary.load()
binary = binary.dropna()
print(binary.describe())
```

```
            admit         gre         gpa       rank
count  400.000000  400.000000  400.000000  400.00000
mean     0.317500  587.700000    3.389900    2.48500
std      0.466087  115.516536    0.380567    0.94446
min      0.000000  220.000000    2.260000    1.00000
25%      0.000000  520.000000    3.130000    2.00000
50%      0.000000  580.000000    3.395000    2.00000
75%      1.000000  660.000000    3.670000    3.00000
max      1.000000  800.000000    4.000000    4.00000
```

Fitting the model directly

The first steps are to build the regressor and the dependent variable arrays. Then, using these arrays, the model can be estimated by calling fit

[13]:
```
import statsmodels.api as sm

endog = binary[["admit"]]
exog = binary[["gre", "gpa"]]
const = pd.Series(np.ones(exog.shape[0]), index=endog.index)
const.name = "Const"
exog = pd.DataFrame([const, exog.gre, exog.gpa]).T

# Estimate the model
mod = sm.Probit(endog, exog)
fit = mod.fit(disp=0)
params = fit.params
print(params)
```

```
Const   -3.003536
gre      0.001643
gpa      0.454575
dtype: float64
```

The wrapper function

Most models in statsmodels are implemented as classes, require an explicit call to `fit` and return a class containing parameter estimates and other quantities. These classes cannot be directly used with the bootstrap methods. However, a simple wrapper can be written that takes the data as the only inputs and returns parameters estimated using a Statsmodel model.

[14]:
```python
def probit_wrap(endog, exog):
    return sm.Probit(endog, exog).fit(disp=0).params
```

A call to this function should return the same parameter values.

[15]:
```python
probit_wrap(endog, exog)
```

[15]:
```
Const   -3.003536
gre      0.001643
gpa      0.454575
dtype: float64
```

The wrapper can be directly used to estimate the parameter covariance or to construct confidence intervals.

[16]:
```python
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(endog=endog, exog=exog)
cov = bs.cov(probit_wrap, 1000)
cov = pd.DataFrame(cov, index=exog.columns, columns=exog.columns)
print(cov)
```

```
          Const           gre       gpa
Const  0.397473 -6.641971e-05 -0.102525
gre   -0.000066  4.467596e-07 -0.000058
gpa   -0.102525 -5.815162e-05  0.039859
```

[17]:
```python
se = pd.Series(np.sqrt(np.diag(cov)), index=exog.columns)
print(se)
print("T-stats")
print(params / se)
```

```
Const    0.630455
gre      0.000668
gpa      0.199647
dtype: float64
T-stats
Const   -4.764077
gre      2.457413
gpa      2.276894
dtype: float64
```

[18]:
```python
ci = bs.conf_int(probit_wrap, 1000, method="basic")
ci = pd.DataFrame(ci, index=["Lower", "Upper"], columns=exog.columns)
print(ci)
```

```
          Const       gre       gpa
Lower -4.214157  0.000360  0.005706
Upper -1.622607  0.002906  0.871725
```

Speeding things up

Starting values can be provided to `fit` which can save time finding starting values. Since the bootstrap parameter estimates should be close to the original sample estimates, the full sample estimated parameters are reasonable starting values. These can be passed using the `extra_kwargs` dictionary to a modified wrapper that will accept a keyword argument containing starting values.

[19]:
```
def probit_wrap_start_params(endog, exog, start_params=None):
    return sm.Probit(endog, exog).fit(start_params=start_params, disp=0).params
```

[20]:
```
bs.reset()  # Reset to original state for comparability
cov = bs.cov(
    probit_wrap_start_params, 1000, extra_kwargs={"start_params": params.values}
)
cov = pd.DataFrame(cov, index=exog.columns, columns=exog.columns)
print(cov)
```

```
          Const           gre       gpa
Const  0.397473 -6.641971e-05 -0.102525
gre   -0.000066  4.467596e-07 -0.000058
gpa   -0.102525 -5.815162e-05  0.039859
```

2.1.4 Bootstrapping Uneven Length Samples

Independent samples of uneven length are common in experiment settings, e.g., A/B testing of a website. The `IIDBootstrap` allows for arbitrary dependence within an observation index and so cannot be naturally applied to these data sets. The `IndependentSamplesBootstrap` allows datasets with variables of different lengths to be sampled by exploiting the independence of the values to separately bootstrap each component. Below is an example showing how a confidence interval can be constructed for the difference in means of two groups.

[21]:
```
from arch.bootstrap import IndependentSamplesBootstrap


def mean_diff(x, y):
    return x.mean() - y.mean()


rs = np.random.RandomState(0)
treatment = 0.2 + rs.standard_normal(200)
control = rs.standard_normal(800)

bs = IndependentSamplesBootstrap(treatment, control, seed=seed)
print(bs.conf_int(mean_diff, method="studentized"))
```

```
[[0.19450863]
 [0.49723719]]
```

2.2 Confidence Intervals

The confidence interval function allows three types of confidence intervals to be constructed:

- Nonparametric, which only resamples the data
- Semi-parametric, which use resampled residuals
- Parametric, which simulate residuals

Confidence intervals can then be computed using one of 6 methods:

- Basic (`basic`)
- Percentile (`percentile`)
- Studentized (`studentized`)
- Asymptotic using parameter covariance (`norm`, `var` or `cov`)
- Bias-corrected (`bc`, `bias-corrected` or `debiased`)
- Bias-corrected and accelerated (`bca`)

- *Setup*
- *Confidence Interval Types*
  - *Nonparametric Confidence Intervals*
  - *Semi-parametric Confidence Intervals*
  - *Parametric Confidence Intervals*
- *Confidence Interval Methods*
  - *Basic (`basic`)*
  - *Percentile (`percentile`)*
  - *Asymptotic Normal Approximation (`norm`, `cov` or `var`)*
  - *Studentized (`studentized`)*
  - *Bias-corrected (`bc`, `bias-corrected` or `debiased`)*
  - *Bias-corrected and accelerated (`bca`)*

2.2.1 Setup

All examples will construct confidence intervals for the Sharpe ratio of the S&P 500, which is the ratio of the annualized mean to the annualized standard deviation. The parameters will be the annualized mean, the annualized standard deviation and the Sharpe ratio.

The setup makes use of return data downloaded from Yahoo!

```
import datetime as dt

import pandas as pd
import pandas_datareader.data as web
```

(continues on next page)

(continued from previous page)

```
start = dt.datetime(1951, 1, 1)
end = dt.datetime(2014, 1, 1)
sp500 = web.DataReader('^GSPC', 'yahoo', start=start, end=end)
low = sp500.index.min()
high = sp500.index.max()
monthly_dates = pd.date_range(low, high, freq='M')
monthly = sp500.reindex(monthly_dates, method='ffill')
returns = 100 * monthly['Adj Close'].pct_change().dropna()
```

The main function used will return a 3-element array containing the parameters.

```
def sharpe_ratio(x):
    mu, sigma = 12 * x.mean(), np.sqrt(12 * x.var())
    return np.array([mu, sigma, mu / sigma])
```

Note

Functions must return 1-d NumPy arrays or Pandas Series.

2.2.2 Confidence Interval Types

Three types of confidence intervals can be computed. The simplest are non-parametric; these only make use of parameter estimates from both the original data as well as the resampled data. Semi-parametric mix the original data with a limited form of resampling, usually for residuals. Finally, parametric bootstrap confidence intervals make use of a parametric distribution to construct "as-if" exact confidence intervals.

Nonparametric Confidence Intervals

Non-parametric sampling is the simplest method to construct confidence intervals.

This example makes use of the percentile bootstrap which is conceptually the simplest method - it constructs many bootstrap replications and returns order statistics from these empirical distributions.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='percentile')
```

Note

While returns have little serial correlation, squared returns are highly persistent. The IID bootstrap is not a good choice here. Instead a time-series bootstrap with an appropriately chosen block size should be used.

Semi-parametric Confidence Intervals

See *Semiparametric Bootstraps*

Parametric Confidence Intervals

See *Parametric Bootstraps*

2.2.3 Confidence Interval Methods

Note

`conf_int` can construct two-sided, upper or lower (one-sided) confidence intervals. All examples use two-sided, 95% confidence intervals (the default). This can be modified using the keyword inputs `type` (`'upper'`, `'lower'` or `'two-sided'`) and `size`.

Basic (`basic`)

Basic confidence intervals construct many bootstrap replications $\hat{\theta}_b^\star$ and then constructs the confidence interval as

$$\left[\hat{\theta}+\left(\hat{\theta}-\hat{\theta}_u^\star\right),\hat{\theta}+\left(\hat{\theta}-\hat{\theta}_l^\star\right)\right]$$

where $\hat{\theta}_l^\star$ and $\hat{\theta}_u^\star$ are the $\alpha/2$ and $1-\alpha/2$ empirical quantiles of the bootstrap distribution. When θ is a vector, the empirical quantiles are computed element-by-element.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='basic')
```

Percentile (`percentile`)

The percentile method directly constructs confidence intervals from the empirical CDF of the bootstrap parameter estimates, $\hat{\theta}_b^\star$. The confidence interval is then defined.

$$\left[\hat{\theta}_l^\star,\hat{\theta}_u^\star\right]$$

where $\hat{\theta}_l^\star$ and $\hat{\theta}_u^\star$ are the $\alpha/2$ and $1-\alpha/2$ empirical quantiles of the bootstrap distribution.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='percentile')
```

Asymptotic Normal Approximation (`norm`, `cov` or `var`)

The asymptotic normal approximation method estimates the covariance of the parameters and then combines this with the usual quantiles from a normal distribution. The confidence interval is then

$$\left[\hat{\theta}+\hat{\sigma}\Phi^{-1}\left(\alpha/2\right),\hat{\theta}-\hat{\sigma}\Phi^{-1}\left(\alpha/2\right),\right]$$

where $\hat{\sigma}$ is the bootstrap estimate of the parameter standard error.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='norm')
```

Studentized (`studentized`)

The studentized bootstrap may be more accurate than some of the other methods. The studentized bootstrap makes use of either a standard error function, when parameter standard errors can be analytically computed, or a nested bootstrap, to bootstrap studentized versions of the original statistic. This can produce higher-order refinements in some circumstances.

The confidence interval is then

$$\left[\hat{\theta}+\hat{\sigma}\hat{G}^{-1}\left(\alpha/2\right),\hat{\theta}+\hat{\sigma}\hat{G}^{-1}\left(1-\alpha/2\right),\right]$$

where $\hat{G}$ is the estimated quantile function for the studentized data and where $\hat{\sigma}$ is a bootstrap estimate of the parameter standard error.

The version that uses a nested bootstrap is simple to implement although it can be slow since it requires B inner bootstraps of each of the B outer bootstraps.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='studentized')
```

In order to use the standard error function, it is necessary to estimate the standard error of the parameters. In this example, this can be done using a method-of-moments argument and the delta-method. A detailed description of the mathematical formula is beyond the intent of this document.

```
def sharpe_ratio_se(params, x):
    mu, sigma, sr = params
    y = 12 * x
    e1 = y - mu
    e2 = y ** 2.0 - sigma ** 2.0
    errors = np.vstack((e1, e2)).T
    t = errors.shape[0]
    vcv = errors.T.dot(errors) / t
    D = np.array([[1, 0],
                  [0, 0.5 * 1 / sigma],
                  [1.0 / sigma, - mu / (2.0 * sigma**3)]
                  ])
    avar = D.dot(vcv /t).dot(D.T)
    return np.sqrt(np.diag(avar))
```

The studentized bootstrap can then be implemented using the standard error function.

```
from arch.bootstrap import IIDBootstrap
bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='studentized',
                 std_err_func=sharpe_ratio_se)
```

Note

Standard error functions must return a 1-d array with the same number of element as params.

Note

Standard error functions must match the patters `std_err_func(params, *args, **kwargs)` where `params` is an array of estimated parameters constructed using `*args` and `**kwargs`.

Bias-corrected (bc, bias-corrected or debiased)

The bias corrected bootstrap makes use of a bootstrap estimate of the bias to improve confidence intervals.

```
from arch.bootstrap import IIDBootstrap
bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='bc')
```

The bias-corrected confidence interval is identical to the bias-corrected and accelerated where $a = 0$.

Bias-corrected and accelerated (bca)

Bias-corrected and accelerated confidence intervals make use of both a bootstrap bias estimate and a jackknife acceleration term. BCa intervals may offer higher-order accuracy if some conditions are satisfied. Bias-corrected confidence intervals are a special case of BCa intervals where the acceleration parameter is set to 0.

```
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
ci = bs.conf_int(sharpe_ratio, 1000, method='bca')
```

The confidence interval is based on the empirical distribution of the bootstrap parameter estimates, $\hat{\theta}_b^\star$, where the percentiles used are

$$\Phi\left(\Phi^{-1}\left(\hat{b}\right) + \frac{\Phi^{-1}\left(\hat{b}\right) + z_\alpha}{1 - \hat{a}\left(\Phi^{-1}\left(\hat{b}\right) + z_\alpha\right)}\right)$$

where z_α is the usual quantile from the normal distribution and b is the empirical bias estimate,

$$\hat{b} = \#\left\{\hat{\theta}_b^\star < \hat{\theta}\right\}/B$$

a is a skewness-like estimator using a leave-one-out jackknife.

2.3 Covariance Estimation

The bootstrap can be used to estimate parameter covariances in applications where analytical computation is challenging, or simply as an alternative to traditional estimators.

This example estimates the covariance of the mean, standard deviation and Sharpe ratio of the S&P 500 using Yahoo! Finance data.

```
import datetime as dt
import pandas as pd
import pandas_datareader.data as web

start = dt.datetime(1951, 1, 1)
end = dt.datetime(2014, 1, 1)
sp500 = web.DataReader('^GSPC', 'yahoo', start=start, end=end)
low = sp500.index.min()
high = sp500.index.max()
monthly_dates = pd.date_range(low, high, freq='M')
monthly = sp500.reindex(monthly_dates, method='ffill')
returns = 100 * monthly['Adj Close'].pct_change().dropna()
```

The function that returns the parameters.

```
def sharpe_ratio(r):
    mu = 12 * r.mean(0)
    sigma = np.sqrt(12 * r.var(0))
    sr = mu / sigma
    return np.array([mu, sigma, sr])
```

Like all applications of the bootstrap, it is important to choose a bootstrap that captures the dependence in the data. This example uses the stationary bootstrap with an average block size of 12.

```
import pandas as pd
from arch.bootstrap import StationaryBootstrap

bs = StationaryBootstrap(12, returns)
param_cov = bs.cov(sharpe_ratio)
index = ['mu', 'sigma', 'SR']
params = sharpe_ratio(returns)
params = pd.Series(params, index=index)
param_cov = pd.DataFrame(param_cov, index=index, columns=index)
```

The output is

```
>>> params
mu        8.148534
sigma    14.508540
SR        0.561637
dtype: float64

>>> param_cov
             mu     sigma        SR
mu     3.729435 -0.442891  0.273945
```

(continues on next page)

(continued from previous page)

```
sigma -0.442891  0.495087 -0.049454
SR     0.273945 -0.049454  0.020830
```

Note

The covariance estimator is centered using the average of the bootstrapped estimators. The original sample estimator can be used to center using the keyword argument `recenter=False`.

2.4 Low-level Interfaces

2.4.1 Constructing Parameter Estimates

The bootstrap method apply can be use to directly compute parameter estimates from a function and the bootstrapped data.

This example makes use of monthly S&P 500 data.

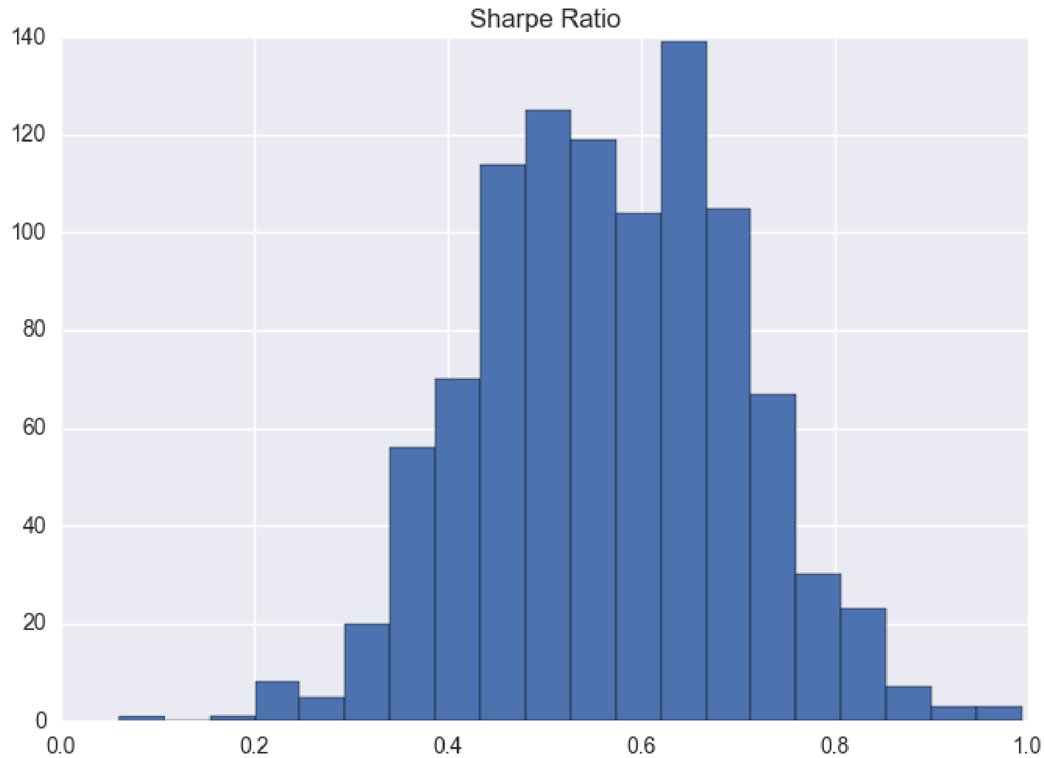
```
import datetime as dt

import pandas as pd
import pandas_datareader.data as web

start = dt.datetime(1951, 1, 1)
end = dt.datetime(2014, 1, 1)
sp500 = web.DataReader('^GSPC', 'yahoo', start=start, end=end)
low = sp500.index.min()
high = sp500.index.max()
monthly_dates = pd.date_range(low, high, freq='M')
monthly = sp500.reindex(monthly_dates, method='ffill')
returns = 100 * monthly['Adj Close'].pct_change().dropna()
```

The function will compute the Sharpe ratio – the (annualized) mean divided by the (annualized) standard deviation.

```
import numpy as np
def sharpe_ratio(x):
    return np.array([12 * x.mean() / np.sqrt(12 * x.var())])
```

The bootstrapped Sharpe ratios can be directly computed using *apply*.

```
import seaborn
from arch.bootstrap import IIDBootstrap
bs = IIDBootstrap(returns)
sharpe_ratios = bs.apply(sr, 1000)
sharpe_ratios = pd.DataFrame(sharp_ratios, columns=['Sharpe Ratio'])
sharpe_ratios.hist(bins=20)
```



2.4.2 The Bootstrap Iterator

The lowest-level method to use a bootstrap is the iterator. This is used internally in all higher-level methods that estimate a function using multiple bootstrap replications. The iterator returns a two-element tuple where the first element contains all positional arguments (in the order input) passed when constructing the bootstrap instance, and the second contains the all keyword arguments passed when constructing the instance.

This example makes uses of simulated data to demonstrate how to use the bootstrap iterator.

```
import pandas as pd
import numpy as np

from arch.bootstrap import IIDBootstrap

x = np.random.randn(1000, 2)
y = pd.DataFrame(np.random.randn(1000, 3))
z = np.random.rand(1000, 10)
bs = IIDBootstrap(x, y=y, z=z)

for pos, kw in bs.bootstrap(1000):
    xstar = pos[0]  # pos is always a tuple, even when a singleton
    ystar = kw['y']  # A dictionary
    zstar = kw['z']  # A dictionary
```

2.5 Semiparametric Bootstraps

Functions for semi-parametric bootstraps differ from those used in nonparametric bootstraps. At a minimum they must accept the keyword argument `params` which will contain the parameters estimated on the original (non-bootstrap) data. This keyword argument must be optional so that the function can be called without the keyword argument to estimate parameters. In most applications other inputs will also be needed to perform the semi-parametric step - these can be input using the `extra_kwargs` keyword input.

For simplicity, consider a semiparametric bootstrap of an OLS regression. The bootstrap step will combine the original parameter estimates and original regressors with bootstrapped residuals to construct a bootstrapped regressand. The bootstrap regressand and regressors can then be used to produce a bootstrapped parameter estimate.

The user-provided function must:

- Estimate the parameters when `params` is not provided
- Estimate residuals from bootstrapped data when `params` is provided to construct bootstrapped residuals, simulate the regressand, and then estimate the bootstrapped parameters

```
import numpy as np
def ols(y, x, params=None, x_orig=None):
    if params is None:
        return np.linalg.pinv(x).dot(y).ravel()

    # When params is not None
    # Bootstrap residuals
    resids = y - x.dot(params)
    # Simulated data
    y_star = x_orig.dot(params) + resids
    # Parameter estimates
    return np.linalg.pinv(x_orig).dot(y_star).ravel()
```

Note

The function should return a 1-dimensional array. `ravel` is used above to ensure that the parameters estimated are 1d.

This function can then be used to perform a semiparametric bootstrap

```
from arch.bootstrap import IIDBootstrap
x = np.random.randn(100, 3)
e = np.random.randn(100, 1)
b = np.arange(1, 4)[:, None]
y = x.dot(b) + e
bs = IIDBootstrap(y, x)
ci = bs.conf_int(ols, 1000, method='percentile',
                 sampling='semi', extra_kwargs={'x_orig': x})
```

2.5.1 Using `partial` instead of `extra_kwargs`

`functools.partial` can be used instead to provide a wrapper function which can then be used in the bootstrap. This example fixed the value of `x_orig` so that it is not necessary to use `extra_kwargs`.

```
from functools import partial
ols_partial = partial(ols, x_orig=x)
ci = bs.conf_int(ols_partial, 1000, sampling='semi')
```

2.5.2 Semiparametric Bootstrap (Alternative Method)

Since semiparametric bootstraps are effectively bootstrapping residuals, an alternative method can be used to conduct a semiparametric bootstrap. This requires passing both the data and the estimated residuals when initializing the bootstrap.

First, the function used must be account for this structure.

```
def ols_semi_v2(y, x, resids=None, params=None, x_orig=None):
    if params is None:
        return np.linalg.pinv(x).dot(y).ravel()

    # Simulated data if params provided
    y_star = x_orig.dot(params) + resids
    # Parameter estimates
    return np.linalg.pinv(x_orig).dot(y_star).ravel()
```

This version can then be used to *directly* implement a semiparametric bootstrap, although ultimately it is not meaningfully simpler than the previous method.

```
resids = y - x.dot(ols_semi_v2(y,x))
bs = IIDBootstrap(y, x, resids=resids)
bs.conf_int(ols_semi_v2, 1000, sampling='semi', extra_kwargs={'x_orig': x})
```

Note

This alternative method is more useful when computing residuals is relatively expensive when compared to simulating data or estimating parameters. These circumstances are rarely encountered in actual problems.

2.6 Parametric Bootstraps

Parametric bootstraps are meaningfully different from their nonparametric or semiparametric cousins. Instead of sampling the data to simulate the data (or residuals, in the case of a semiparametric bootstrap), a parametric bootstrap makes use of a fully parametric model to simulate data using a pseudo-random number generator.

Warning

Parametric bootstraps are model-based methods to construct exact confidence intervals through integration. Since these confidence intervals should be exact, bootstrap methods which make use of asymptotic normality are required (and may not be desirable).

Implementing a parametric bootstrap, like implementing a semi-parametric bootstrap, requires specific keyword arguments. The first is `params`, which, when present, will contain the parameters estimated on the original data. The second is `rng` which will contain the `numpy.random.RandomState` instance that is used by the bootstrap. This is provided to facilitate simulation in a reproducible manner.

A parametric bootstrap function must:

- Estimate the parameters when `params` is not provided
- Simulate data when `params` is provided and then estimate the bootstrapped parameters on the simulated data

This example continues the OLS example from the semiparametric example, only assuming that residuals are normally distributed. The variance estimator is the MLE.

```
def ols_para(y, x, params=None, state=None, x_orig=None):
    if params is None:
        beta = np.linalg.pinv(x).dot(y)
        e = y - x.dot(beta)
        sigma2 = e.T.dot(e) / e.shape[0]
        return np.r_[beta.ravel(), sigma2.ravel()]

    beta = params[:-1]
    sigma2 = params[-1]
    e = state.standard_normal(x_orig.shape[0])
    ystar = x_orig.dot(beta) + np.sqrt(sigma2) * e

    # Use the plain function to compute parameters
    return ols_para(ystar, x_orig)
```

This function can then be used to form parametric bootstrap confidence intervals.

```
bs = IIDBootstrap(y,x)
ci = bs.conf_int(ols_para, 1000, method='percentile',
                 sampling='parametric', extra_kwargs={'x_orig': x})
```

Note

The parameter vector in this example includes the variance since this is required when specifying a complete model.

2.7 Independent, Identical Distributed Data (i.i.d.)

IIDBootstrap is the standard bootstrap that is appropriate for data that is either i.i.d. or at least not serially dependant.

IIDBootstrap(*args[, random_state, seed])	Bootstrap using uniform resampling

2.7.1 arch.bootstrap.IIDBootstrap

`class arch.bootstrap.IIDBootstrap(*args: ArrayLike, random_state: RandomState | None = None, seed: None | int | Generator | RandomState = None, **kwargs: ArrayLike)`

Bootstrap using uniform resampling

Parameters

***args: ArrayLike**
Positional arguments to bootstrap

seed: None | int | Generator | RandomState = None
Seed to use to ensure reproducable results. If an int, passes the value to value to `np.random.default_rng`. If None, a fresh Generator is constructed with system-provided entropy.

random_state: RandomState | None = None

`RandomState` to use to ensure reproducable results. Cannot be used with `seed`

Deprecated since version 5.0: The random_state keyword argument has been deprecated. Use seed instead.

****kwargs: ArrayLike**
Keyword arguments to bootstrap

data

Two-element tuple with the pos_data in the first position and kw_data in the second (pos_data, kw_data)

Type
`tuple`

pos_data

Tuple containing the positional arguments (in the order entered)

Type
`tuple`

kw_data

Dictionary containing the keyword arguments

Type
`dict`

Notes

Supports numpy arrays and pandas Series and DataFrames. Data returned has the same type as the input date.

Data entered using keyword arguments is directly accessibly as an attribute.

To ensure a reproducible bootstrap, you must set the `seed` attribute after the bootstrap has been created. See the example below. Note that `seed` is a reserved keyword and any variable passed using this keyword must be an integer, a `Generator` or a `RandomState`.

Examples

Data can be accessed in a number of ways. Positional data is retained in the same order as it was entered when the bootstrap was initialized. Keyword data is available both as an attribute or using a dictionary syntax on kw_data.

```
>>> from arch.bootstrap import IIDBootstrap
>>> from numpy.random import standard_normal
>>> y = standard_normal((500, 1))
>>> x = standard_normal((500,2))
>>> z = standard_normal(500)
>>> bs = IIDBootstrap(x, y=y, z=z)
>>> for data in bs.bootstrap(100):
...     bs_x = data[0][0]
...     bs_y = data[1]['y']
...     bs_z = bs.z
```

Set the seed if reproducibility is required

```
>>> from numpy.random import default_rng
>>> seed = default_rng(1234)
>>> bs = IIDBootstrap(x, y=y, z=z, seed=seed)
```

This is equivalent to

```
>>> bs = IIDBootstrap(x, y=y, z=z, seed=1234)
```

See also

arch.bootstrap.IndependentSamplesBootstrap

Methods

apply(func[, reps, extra_kwargs])	Applies a function to bootstrap replicated data
bootstrap(reps)	Iterator for use when bootstrapping
clone(*args[, seed])	Clones the bootstrap using different data with a fresh prng.
conf_int(func[, reps, method, size, tail, ...])	**param func** Function the computes parameter values. See Notes for requirements
cov(func[, reps, recenter, extra_kwargs])	Compute parameter covariance using bootstrap
get_state()	Gets the state of the bootstrap's random number generator
reset([use_seed])	Resets the bootstrap to either its initial state or the last seed.
seed(value)	Reseeds the bootstrap's random number generator
set_state(state)	Sets the state of the bootstrap's random number generator
update_indices()	Update indices for the next iteration of the bootstrap.
var(func[, reps, recenter, extra_kwargs])	Compute parameter variance using bootstrap

arch.bootstrap.IIDBootstrap.apply

IIDBootstrap.**apply**(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = *1000*, extra_kwargs: *dict[str, Any] | None* = *None*) → ndarray

Applies a function to bootstrap replicated data

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Function the computes parameter values. See Notes for requirements

reps: int = 1000
Number of bootstrap replications

extra_kwargs: dict[str, Any] | None = None
Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

Returns
reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

Return type
`numpy.ndarray`

Notes

When there are no extra keyword arguments, the function is called

```
func(params, *args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func

Examples

```
>>> import numpy as np
>>> x = np.random.randn(1000,2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(x)
>>> def func(y):
...     return y.mean(0)
>>> results = bs.apply(func, 100)
```

arch.bootstrap.IIDBootstrap.bootstrap

IIDBootstrap.**bootstrap**(reps: *int*) → Generator[tuple[tuple[ndarray | DataFrame | Series, ...], dict[str, ndarray | DataFrame | Series]], None, None]

Iterator for use when bootstrapping

Parameters

reps: int
Number of bootstrap replications

Returns
Generator to iterate over in bootstrap calculations

Return type
generator

Examples

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> bs = IIDBootstrap(np.arange(100), x=np.random.randn(100))
>>> for posdata, kwdata in bs.bootstrap(1000):
...     # Do something with the positional data and/or keyword data
...     pass
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

---

Notes

The iterator returns a tuple containing the data entered in positional arguments as a tuple and the data entered using keywords as a dictionary

arch.bootstrap.IIDBootstrap.clone

IIDBootstrap.**clone**(*args: *ArrayLike*, seed: *None* | *int* | *Generator* | *RandomState* = ***None***, **kwargs: *ArrayLike*) → *IIDBootstrap*

Clones the bootstrap using different data with a fresh prng.

Parameters

***args: ArrayLike**
Positional arguments to bootstrap

seed: None | int | Generator | RandomState = None
The seed value to pass to the closed generator

****kwargs: ArrayLike**
Keyword arguments to bootstrap

Returns
Bootstrap instance

Return type
bs

arch.bootstrap.IIDBootstrap.conf_int

`IIDBootstrap.conf_int`(func: *Callable[[...], ndarray]*, reps: *int* = 1000, method: *'basic' | 'percentile' | 'studentized' | 'norm' | 'bc' | 'bca'* = 'basic', size: *float* = 0.95, tail: *'two' | 'upper' | 'lower'* = 'two', extra_kwargs: *dict[str, Any] | None* = None, reuse: *bool* = False, sampling: *'nonparametric' | 'semi-parametric' | 'semi' | 'parametric' | 'semiparametric'* = 'nonparametric', std_err_func: *Callable[[...], ndarray | DataFrame | Series] | None* = None, studentize_reps: *int* = 1000) → ndarray

Parameters

func: Callable[[...], ndarray]
Function the computes parameter values. See Notes for requirements

reps: int = 1000
Number of bootstrap replications

method: 'basic' | 'percentile' | 'studentized' | 'norm' | 'bc' | 'bca' = 'basic'
One of 'basic', 'percentile', 'studentized', 'norm' (identical to 'var', 'cov'), 'bc' (identical to 'debiased', 'bias-corrected'), or 'bca'

size: float = 0.95
Coverage of confidence interval

tail: 'two' | 'upper' | 'lower' = 'two'
One of 'two', 'upper' or 'lower'.

reuse: bool = False
Flag indicating whether to reuse previously computed bootstrap results. This allows alternative methods to be compared without rerunning the bootstrap simulation. Reuse is ignored if reps is not the same across multiple runs, func changes across calls, or method is 'studentized'.

sampling: 'nonparametric' | 'semi-parametric' | 'semi' | 'parametric' | 'semiparametric' = 'nonparametric'
Type of sampling to use: 'nonparametric', 'semi-parametric' (or 'semi') or 'parametric'. The default is 'nonparametric'. See notes about the changes to func required when using 'semi' or 'parametric'.

extra_kwargs: dict[str, Any] | None = None
Extra keyword arguments to use when calling func and std_err_func, when appropriate

std_err_func: Callable[[...], ndarray | DataFrame | Series] | None = None
Function to use when standardizing estimated parameters when using the studentized bootstrap. Providing an analytical function eliminates the need for a nested bootstrap

studentize_reps: int = 1000
Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when `std_err_func` is provided

Returns

Computed confidence interval. Row 0 contains the lower bounds, and row 1 contains the upper bounds. Each column corresponds to a parameter. When tail is 'lower', all upper bounds are inf. Similarly, 'upper' sets all lower bounds to -inf.

Return type

`numpy.ndarray`

Examples

```
>>> import numpy as np
>>> def func(x):
...     return x.mean(0)
>>> y = np.random.randn(1000, 2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(y)
>>> ci = bs.conf_int(func, 1000)
```

Notes

When there are no extra keyword arguments, the function is called

```
func(*args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func.

The standard error function, if provided, must return a vector of parameter standard errors and is called

```
std_err_func(params, *args, **kwargs)
```

where `params` is the vector of estimated parameters using the same bootstrap data as in args and kwargs.

The bootstraps are:

- 'basic' - Basic confidence using the estimated parameter and difference between the estimated parameter and the bootstrap parameters
- 'percentile' - Direct use of bootstrap percentiles
- 'norm' - Makes use of normal approximation and bootstrap covariance estimator
- 'studentized' - Uses either a standard error function or a nested bootstrap to estimate percentiles and the bootstrap covariance for scale
- 'bc' - Bias corrected using estimate bootstrap bias correction
- 'bca' - Bias corrected and accelerated, adding acceleration parameter to 'bc' method

arch.bootstrap.IIDBootstrap.cov

`IIDBootstrap.cov`(func: *Callable[[...], ndarray* | *DataFrame* | *Series]*, reps: *int* = **1000**, recenter: *bool* = **True**, extra_kwargs: *dict[str, Any]* | *None* = **None**) → float | ndarray

Compute parameter covariance using bootstrap

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Callable function that returns the statistic of interest as a 1-d array

reps: int = 1000
Number of bootstrap replications

recenter: bool = True
Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.

extra_kwargs: dict[str, Any] | None = None
Dictionary of extra keyword arguments to pass to func

Returns
Bootstrap covariance estimator

Return type
numpy.ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Examples

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> cov = bs.cov(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> cov = bs.cov(func, 1000, extra_kwargs={'stat':'var'})
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

---

arch.bootstrap.IIDBootstrap.get_state

`IIDBootstrap.get_state()` → tuple[str, ndarray, int, int, float] | Mapping[str, Any]

Gets the state of the bootstrap's random number generator

Returns
Dictionary containing the state.

Return type
`dict`

arch.bootstrap.IIDBootstrap.reset

`IIDBootstrap.reset(use_seed: bool = True)` → None

Resets the bootstrap to either its initial state or the last seed.

Parameters

use_seed: bool = True
Flag indicating whether to use the last seed if provided. If False or if no seed has been set, the bootstrap will be reset to the initial state. Default is True

arch.bootstrap.IIDBootstrap.seed

`IIDBootstrap.seed(value: int | list[int] | ndarray)` → None

Reseeds the bootstrap's random number generator

Parameters

value: int | list[int] | ndarray
Value to use as the seed.

arch.bootstrap.IIDBootstrap.set_state

`IIDBootstrap.set_state(state: tuple[str, ndarray, int, int, float] | dict[str, Any])` → None

Sets the state of the bootstrap's random number generator

Parameters

state: tuple[str, ndarray, int, int, float] | dict[str, Any]
Dictionary or tuple containing the state.

arch.bootstrap.IIDBootstrap.update_indices

`IIDBootstrap.update_indices()` → ndarray | tuple[ndarray, ...] | tuple[list[ndarray], dict[str, ndarray]]

Update indices for the next iteration of the bootstrap. This must be overridden when creating new bootstraps.

arch.bootstrap.IIDBootstrap.var

IIDBootstrap.**var**(func*: Callable[[...], ndarray | DataFrame | Series]*, reps*: int = 1000*, recenter*: bool = True*, extra_kwargs*: dict[str, Any] | None = None*) → float | ndarray

Compute parameter variance using bootstrap

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Callable function that returns the statistic of interest as a 1-d array

reps: int = 1000
Number of bootstrap replications

recenter: bool = True
Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.

extra_kwargs: dict[str, Any] | None = None
Dictionary of extra keyword arguments to pass to func

Returns
Bootstrap variance estimator

Return type
numpy.ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Examples

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> variances = bs.var(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
```

(continues on next page)

(continued from previous page)

```
...     elif stat=='var':
...         return x.var(axis=0)
>>> variances = bs.var(func, 1000, extra_kwargs={'stat': 'var'})
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

Properties

generator	Set or get the instance PRNG
index	The current index of the bootstrap
random_state	Set or get the instance random state
state	Set or get the generator's state

arch.bootstrap.IIDBootstrap.generator

property `IIDBootstrap.generator` : Generator | RandomState

Set or get the instance PRNG

Parameters

seed : {Generator, RandomState}, ***optional***
Generator or RandomState used to produce the pseudo-random values used in the bootstrap

Returns
The instance of the Generator or RandomState instance used by bootstrap

Return type
`{Generator, RandomState}`

arch.bootstrap.IIDBootstrap.index

property `IIDBootstrap.index` : ndarray | tuple[ndarray, ...] | tuple[list[ndarray], dict[str, ndarray]]

The current index of the bootstrap

arch.bootstrap.IIDBootstrap.random_state

property `IIDBootstrap.random_state` : Generator | RandomState

Set or get the instance random state

Parameters

random_state : `numpy.random.RandomState`
RandomState instance used by bootstrap

Returns
RandomState instance used by bootstrap

Return type
numpy.random.RandomState

arch.bootstrap.IIDBootstrap.state

property IIDBootstrap.**state** : tuple[str, ndarray, int, int, float] | Mapping[str, Any]

Set or get the generator's state

Returns
A tuple or dictionary containing the generator's state. If using a `RandomState`, the value returned is a tuple. Otherwise it is a dictionary.

Return type
`{tuple, dict}`

2.8 Independent Samples

IndependentSamplesBootstrap is a bootstrap that is appropriate for data is totally independent, and where each variable may have a different sample size. This type of data arises naturally in experimental settings, e.g., website A/B testing.

IndependentSamplesBootstrap(*args[, ...])	Bootstrap where each input is independently resampled

2.8.1 arch.bootstrap.IndependentSamplesBootstrap

class arch.bootstrap.**IndependentSamplesBootstrap**(**args: ArrayLike*, *random_state: RandomState | None = None*, *seed: None | int | Generator | RandomState = None*, ***kwargs: ArrayLike*)

Bootstrap where each input is independently resampled

Parameters

***args: ArrayLike**
Positional arguments to bootstrap

****kwargs: ArrayLike**
Keyword arguments to bootstrap

data
Two-element tuple with the pos_data in the first position and kw_data in the second (pos_data, kw_data)

Type
tuple

pos_data
Tuple containing the positional arguments (in the order entered)

Type
tuple

kw_data
Dictionary containing the keyword arguments

Type
dict

Notes

This bootstrap independently resamples each input and so is only appropriate when the inputs are independent. This structure allows bootstrapping statistics that depend on samples with unequal length, as is common in some experiments. If data have cross-sectional dependence, so that observation `i` is related across all inputs, this bootstrap is inappropriate.

Supports numpy arrays and pandas Series and DataFrames. Data returned has the same type as the input date.

Data entered using keyword arguments is directly accessibly as an attribute.

To ensure a reproducible bootstrap, you must set the `random_state` attribute after the bootstrap has been created. See the example below. Note that `random_state` is a reserved keyword and any variable passed using this keyword must be an instance of `RandomState`.

Examples

Data can be accessed in a number of ways. Positional data is retained in the same order as it was entered when the bootstrap was initialized. Keyword data is available both as an attribute or using a dictionary syntax on kw_data.

```
>>> from arch.bootstrap import IndependentSamplesBootstrap
>>> from numpy.random import standard_normal
>>> y = standard_normal(500)
>>> x = standard_normal(200)
>>> z = standard_normal(2000)
>>> bs = IndependentSamplesBootstrap(x, y=y, z=z)
>>> for data in bs.bootstrap(100):
...     bs_x = data[0][0]
...     bs_y = data[1]['y']
...     bs_z = bs.z
```

Set the random_state if reproducibility is required

```
>>> from numpy.random import RandomState
>>> rs = RandomState(1234)
>>> bs = IndependentSamplesBootstrap(x, y=y, z=z, random_state=rs)
```

See also

arch.bootstrap.IIDBootstrap

Methods

apply(func[, reps, extra_kwargs])	Applies a function to bootstrap replicated data
bootstrap(reps)	Iterator for use when bootstrapping
clone(*args[, seed])	Clones the bootstrap using different data with a fresh prng.
conf_int(func[, reps, method, size, tail, ...])	**param func** Function the computes parameter values. See Notes for requirements
cov(func[, reps, recenter, extra_kwargs])	Compute parameter covariance using bootstrap
get_state()	Gets the state of the bootstrap's random number generator
reset([use_seed])	Resets the bootstrap to either its initial state or the last seed.
seed(value)	Reseeds the bootstrap's random number generator
set_state(state)	Sets the state of the bootstrap's random number generator
update_indices()	Update indices for the next iteration of the bootstrap.
var(func[, reps, recenter, extra_kwargs])	Compute parameter variance using bootstrap

arch.bootstrap.IndependentSamplesBootstrap.apply

IndependentSamplesBootstrap.**apply**(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = ***1000***, extra_kwargs: *dict[str, Any] | None* = ***None***) → ndarray

Applies a function to bootstrap replicated data

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Function the computes parameter values. See Notes for requirements

reps: int = 1000
Number of bootstrap replications

extra_kwargs: dict[str, Any] | None = None
Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

Returns

reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

Return type

numpy.ndarray

Notes

When there are no extra keyword arguments, the function is called

```
func(params, *args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func

Examples

```
>>> import numpy as np
>>> x = np.random.randn(1000,2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(x)
>>> def func(y):
...     return y.mean(0)
>>> results = bs.apply(func, 100)
```

arch.bootstrap.IndependentSamplesBootstrap.bootstrap

`IndependentSamplesBootstrap.`**`bootstrap`**(reps: *int*) → Generator[tuple[tuple[ndarray | DataFrame | Series, ...], dict[str, ndarray | DataFrame | Series]], None, None]

Iterator for use when bootstrapping

Parameters

reps: int
Number of bootstrap replications

Returns
Generator to iterate over in bootstrap calculations

Return type
generator

Examples

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> bs = IIDBootstrap(np.arange(100), x=np.random.randn(100))
>>> for posdata, kwdata in bs.bootstrap(1000):
...     # Do something with the positional data and/or keyword data
...     pass
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

Notes

The iterator returns a tuple containing the data entered in positional arguments as a tuple and the data entered using keywords as a dictionary

arch.bootstrap.IndependentSamplesBootstrap.clone

IndependentSamplesBootstrap.**clone**(*args: *ArrayLike*, seed: *None* | *int* | *Generator* | *RandomState* = *None*, **kwargs: *ArrayLike*) → *IIDBootstrap*

Clones the bootstrap using different data with a fresh prng.

Parameters

***args: ArrayLike**
Positional arguments to bootstrap

seed: None | int | Generator | RandomState = None
The seed value to pass to the closed generator

****kwargs: ArrayLike**
Keyword arguments to bootstrap

Returns
Bootstrap instance

Return type
bs

arch.bootstrap.IndependentSamplesBootstrap.conf_int

IndependentSamplesBootstrap.**conf_int**(func: *Callable[[...], ndarray]*, reps: *int* = *1000*, method: *'basic'* | *'percentile'* | *'studentized'* | *'norm'* | *'bc'* | *'bca'* = *'basic'*, size: *float* = *0.95*, tail: *'two'* | *'upper'* | *'lower'* = *'two'*, extra_kwargs: *dict[str, Any]* | *None* = *None*, reuse: *bool* = *False*, sampling: *'nonparametric'* | *'semi-parametric'* | *'semi'* | *'parametric'* | *'semiparametric'* = *'nonparametric'*, std_err_func: *Callable[[...], ndarray* | *DataFrame* | *Series]* | *None* = *None*, studentize_reps: *int* = *1000*) → ndarray

Parameters

func: Callable[[...], ndarray]
Function the computes parameter values. See Notes for requirements

reps: int = 1000
Number of bootstrap replications

method: 'basic' | 'percentile' | 'studentized' | 'norm' | 'bc' | 'bca' = 'basic'
One of 'basic', 'percentile', 'studentized', 'norm' (identical to 'var', 'cov'), 'bc' (identical to 'debiased', 'bias-corrected'), or 'bca'

size: float = 0.95
Coverage of confidence interval

tail: 'two' | 'upper' | 'lower' = 'two'
One of 'two', 'upper' or 'lower'.

reuse: bool = False
Flag indicating whether to reuse previously computed bootstrap results. This allows alternative methods to be compared without rerunning the bootstrap simulation. Reuse is ignored if reps is not the same across multiple runs, func changes across calls, or method is 'studentized'.

sampling: 'nonparametric' | 'semi-parametric' | 'semi' | 'parametric' | 'semiparametric' = 'nonparametric'
Type of sampling to use: 'nonparametric', 'semi-parametric' (or 'semi') or 'parametric'. The default is 'nonparametric'. See notes about the changes to func required when using 'semi' or 'parametric'.

extra_kwargs: dict[str, Any] | None = None
Extra keyword arguments to use when calling func and std_err_func, when appropriate

std_err_func: Callable[[...], ndarray | DataFrame | Series] | None = None
Function to use when standardizing estimated parameters when using the studentized bootstrap. Providing an analytical function eliminates the need for a nested bootstrap

studentize_reps: int = 1000
Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when `std_err_func` is provided

Returns
Computed confidence interval. Row 0 contains the lower bounds, and row 1 contains the upper bounds. Each column corresponds to a parameter. When tail is 'lower', all upper bounds are inf. Similarly, 'upper' sets all lower bounds to -inf.

Return type
`numpy.ndarray`

Examples

```
>>> import numpy as np
>>> def func(x):
...     return x.mean(0)
>>> y = np.random.randn(1000, 2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(y)
>>> ci = bs.conf_int(func, 1000)
```

Notes

When there are no extra keyword arguments, the function is called

```
func(*args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func.

The standard error function, if provided, must return a vector of parameter standard errors and is called

```
std_err_func(params, *args, **kwargs)
```

where `params` is the vector of estimated parameters using the same bootstrap data as in args and kwargs.

The bootstraps are:

- 'basic' - Basic confidence using the estimated parameter and difference between the estimated parameter and the bootstrap parameters
- 'percentile' - Direct use of bootstrap percentiles
- 'norm' - Makes use of normal approximation and bootstrap covariance estimator
- 'studentized' - Uses either a standard error function or a nested bootstrap to estimate percentiles and the bootstrap covariance for scale
- 'bc' - Bias corrected using estimate bootstrap bias correction
- 'bca' - Bias corrected and accelerated, adding acceleration parameter to 'bc' method

arch.bootstrap.IndependentSamplesBootstrap.cov

`IndependentSamplesBootstrap.cov`(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = ***1000***, recenter: *bool* = ***True***, extra_kwargs: *dict[str, Any] | None* = ***None***) → float | ndarray

Compute parameter covariance using bootstrap

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Callable function that returns the statistic of interest as a 1-d array

reps: int = 1000
Number of bootstrap replications

recenter: bool = True
Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.

extra_kwargs: dict[str, Any] | None = None
Dictionary of extra keyword arguments to pass to func

Returns
Bootstrap covariance estimator

Return type
`numpy.ndarray`

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Examples

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> cov = bs.cov(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> cov = bs.cov(func, 1000, extra_kwargs={'stat':'var'})
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

arch.bootstrap.IndependentSamplesBootstrap.get_state

`IndependentSamplesBootstrap.get_state()` → tuple[str, ndarray, int, int, float] | Mapping[str, Any]

Gets the state of the bootstrap's random number generator

Returns

Dictionary containing the state.

Return type

`dict`

arch.bootstrap.IndependentSamplesBootstrap.reset

`IndependentSamplesBootstrap.reset(use_seed: bool = True)` → None

Resets the bootstrap to either its initial state or the last seed.

Parameters

use_seed: bool = True

Flag indicating whether to use the last seed if provided. If False or if no seed has been set, the bootstrap will be reset to the initial state. Default is True

arch.bootstrap.IndependentSamplesBootstrap.seed

`IndependentSamplesBootstrap.seed(value: int | list[int] | ndarray) → None`

Reseeds the bootstrap's random number generator

Parameters

value: int | list[int] | ndarray
Value to use as the seed.

arch.bootstrap.IndependentSamplesBootstrap.set_state

`IndependentSamplesBootstrap.set_state(state: tuple[str, ndarray, int, int, float] | dict[str, Any]) → None`

Sets the state of the bootstrap's random number generator

Parameters

state: tuple[str, ndarray, int, int, float] | dict[str, Any]
Dictionary or tuple containing the state.

arch.bootstrap.IndependentSamplesBootstrap.update_indices

`IndependentSamplesBootstrap.update_indices() → tuple[list[ndarray], dict[str, ndarray]]`

Update indices for the next iteration of the bootstrap. This must be overridden when creating new bootstraps.

arch.bootstrap.IndependentSamplesBootstrap.var

`IndependentSamplesBootstrap.var(func: Callable[[...], ndarray | DataFrame | Series], reps: int = 1000, recenter: bool = True, extra_kwargs: dict[str, Any] | None = None) → float | ndarray`

Compute parameter variance using bootstrap

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Callable function that returns the statistic of interest as a 1-d array

reps: int = 1000
Number of bootstrap replications

recenter: bool = True
Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.

extra_kwargs: dict[str, Any] | None = None
Dictionary of extra keyword arguments to pass to func

Returns
Bootstrap variance estimator

Return type
`numpy.ndarray`

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Examples

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> variances = bs.var(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> variances = bs.var(func, 1000, extra_kwargs={'stat': 'var'})
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

Properties

generator	Set or get the instance PRNG
index	Returns the current index of the bootstrap
random_state	Set or get the instance random state
state	Set or get the generator's state

arch.bootstrap.IndependentSamplesBootstrap.generator

property `IndependentSamplesBootstrap.generator` : Generator | RandomState

Set or get the instance PRNG

Parameters

seed : {Generator, RandomState}, ***optional***
Generator or RandomState used to produce the pseudo-random values used in the bootstrap

Returns
The instance of the Generator or RandomState instance used by bootstrap

Return type
`{Generator, RandomState}`

arch.bootstrap.IndependentSamplesBootstrap.index

property `IndependentSamplesBootstrap.index` : ndarray | tuple[ndarray, ...] | tuple[list[ndarray], dict[str, ndarray]]

Returns the current index of the bootstrap

Returns
2-element tuple containing a list and a dictionary. The list contains indices for each of the positional arguments. The dictionary contains the indices of keyword arguments.

Return type
`tuple[list[ndarray], dict[str, ndarray]]`

arch.bootstrap.IndependentSamplesBootstrap.random_state

property `IndependentSamplesBootstrap.random_state` : Generator | RandomState

Set or get the instance random state

Parameters

random_state : `numpy.random.RandomState`
RandomState instance used by bootstrap

Returns
RandomState instance used by bootstrap

Return type
`numpy.random.RandomState`

arch.bootstrap.IndependentSamplesBootstrap.state

property `IndependentSamplesBootstrap.state` : tuple[str, ndarray, int, int, float] | Mapping[str, Any]

Set or get the generator's state

Returns
A tuple or dictionary containing the generator's state. If using a `RandomState`, the value returned is a tuple. Otherwise it is a dictionary.

Return type
`{tuple, dict}`

2.9 Time-series Bootstraps

Bootstraps for time-series data come in a variety of forms. The three contained in this package are the stationary bootstrap (*StationaryBootstrap*), which uses blocks with an exponentially distributed lengths, the circular block bootstrap (*CircularBlockBootstrap*), which uses fixed length blocks, and the moving block bootstrap which also uses fixed length blocks (*MovingBlockBootstrap*). The moving block bootstrap does *not* wrap around and so observations near the start or end of the series will be systematically under-sampled. It is not recommended for this reason.

StationaryBootstrap(block_size, *args[, ...])	Politis and Romano (1994) bootstrap with expon distributed block sizes
CircularBlockBootstrap(block_size, *args[, ...])	Bootstrap using blocks of the same length with end-to-start wrap around
MovingBlockBootstrap(block_size, *args[, ...])	Bootstrap using blocks of the same length without wrap around
optimal_block_length(x)	Estimate optimal window length for time-series bootstraps

2.9.1 arch.bootstrap.StationaryBootstrap

class arch.bootstrap.**StationaryBootstrap**(block_size: *int*, *args: *ArrayLike*, random_state: *RandomState | None* = **None**, seed: *None | int | Generator | RandomState* = **None**, **kwargs: *ArrayLike*)

Politis and Romano (1994) bootstrap with expon distributed block sizes

Parameters

block_size: int
Average size of block to use

***args: ArrayLike**
Positional arguments to bootstrap

seed: None | int | Generator | RandomState = None
Seed to use to ensure reproducable results. If an int, passes the value to value to `np.random.default_rng`. If None, a fresh Generator is constructed with system-provided entropy.

random_state: RandomState | None = None

`RandomState` to use to ensure reproducable results. Cannot be used with `seed`

Deprecated since version 5.0: The random_state keyword argument has been deprecated. Use seed instead.

****kwargs: ArrayLike**
Keyword arguments to bootstrap

data

Two-element tuple with the pos_data in the first position and kw_data in the second (pos_data, kw_data)

Type
`tuple`

pos_data

Tuple containing the positional arguments (in the order entered)

Type
`tuple`

kw_data
Dictionary containing the keyword arguments

Type
`dict`

Notes

Supports numpy arrays and pandas Series and DataFrames. Data returned has the same type as the input date.

Data entered using keyword arguments is directly accessibly as an attribute.

To ensure a reproducible bootstrap, you must set the `random_state` attribute after the bootstrap has been created. See the example below. Note that `random_state` is a reserved keyword and any variable passed using this keyword must be an instance of `RandomState`.

See also

arch.bootstrap.optimal_block_length
Optimal block length estimation

arch.bootstrap.CircularBlockBootstrap
Circular (wrap-around) bootstrap

Examples

Data can be accessed in a number of ways. Positional data is retained in the same order as it was entered when the bootstrap was initialized. Keyword data is available both as an attribute or using a dictionary syntax on kw_data.

```
>>> from arch.bootstrap import StationaryBootstrap
>>> from numpy.random import standard_normal
>>> y = standard_normal((500, 1))
>>> x = standard_normal((500,2))
>>> z = standard_normal(500)
>>> bs = StationaryBootstrap(12, x, y=y, z=z)
>>> for data in bs.bootstrap(100):
...     bs_x = data[0][0]
...     bs_y = data[1]['y']
...     bs_z = bs.z
```

Set the random_state if reproducibility is required

```
>>> from numpy.random import RandomState
>>> rs = RandomState(1234)
>>> bs = StationaryBootstrap(12, x, y=y, z=z, random_state=rs)
```

Methods

apply(func[, reps, extra_kwargs])	Applies a function to bootstrap replicated data
bootstrap(reps)	Iterator for use when bootstrapping
clone(*args[, seed])	Clones the bootstrap using different data with a fresh prng.
conf_int(func[, reps, method, size, tail, ...])	**param func** Function the computes parameter values. See Notes for requirements
cov(func[, reps, recenter, extra_kwargs])	Compute parameter covariance using bootstrap
get_state()	Gets the state of the bootstrap's random number generator
reset([use_seed])	Resets the bootstrap to either its initial state or the last seed.
seed(value)	Reseeds the bootstrap's random number generator
set_state(state)	Sets the state of the bootstrap's random number generator
update_indices()	Update indices for the next iteration of the bootstrap.
var(func[, reps, recenter, extra_kwargs])	Compute parameter variance using bootstrap

arch.bootstrap.StationaryBootstrap.apply

StationaryBootstrap.**apply**(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = *1000*, extra_kwargs: *dict[str, Any] | None* = *None*) → ndarray

Applies a function to bootstrap replicated data

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Function the computes parameter values. See Notes for requirements

reps: int = 1000
Number of bootstrap replications

extra_kwargs: dict[str, Any] | None = None
Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

Returns

reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

Return type

`numpy.ndarray`

Notes

When there are no extra keyword arguments, the function is called

```
func(params, *args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func

Examples

```
>>> import numpy as np
>>> x = np.random.randn(1000,2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(x)
>>> def func(y):
...     return y.mean(0)
>>> results = bs.apply(func, 100)
```

arch.bootstrap.StationaryBootstrap.bootstrap

StationaryBootstrap.**bootstrap**(reps: *int*) → Generator[tuple[tuple[ndarray | DataFrame | Series, ...], dict[str, ndarray | DataFrame | Series]], None, None]

Iterator for use when bootstrapping

Parameters

reps: int
Number of bootstrap replications

Returns
Generator to iterate over in bootstrap calculations

Return type
generator

Examples

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> bs = IIDBootstrap(np.arange(100), x=np.random.randn(100))
>>> for posdata, kwdata in bs.bootstrap(1000):
...     # Do something with the positional data and/or keyword data
...     pass
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

Notes

The iterator returns a tuple containing the data entered in positional arguments as a tuple and the data entered using keywords as a dictionary

arch.bootstrap.StationaryBootstrap.clone

StationaryBootstrap.**clone**(**args: ArrayLike*, *seed: None | int | Generator | RandomState =* **None**, ***kwargs: ArrayLike*) → *CircularBlockBootstrap*

Clones the bootstrap using different data with a fresh prng.

Parameters

***args: ArrayLike**
Positional arguments to bootstrap

seed: None | int | Generator | RandomState = None
The seed value to pass to the closed generator

****kwargs: ArrayLike**
Keyword arguments to bootstrap

Returns
Bootstrap instance

Return type
bs

arch.bootstrap.StationaryBootstrap.conf_int

StationaryBootstrap.**conf_int**(*func: Callable[[...], ndarray]*, *reps: int =* **1000**, *method: 'basic' | 'percentile' | 'studentized' | 'norm' | 'bc' | 'bca' =* `'basic'`, *size: float =* **0.95**, *tail: 'two' | 'upper' | 'lower' =* `'two'`, *extra_kwargs: dict[str, Any] | None =* **None**, *reuse: bool =* **False**, *sampling: 'nonparametric' | 'semi-parametric' | 'semi' | 'parametric' | 'semiparametric' =* `'nonparametric'`, *std_err_func: Callable[[...], ndarray | DataFrame | Series] | None =* **None**, *studentize_reps: int =* **1000**) → ndarray

Parameters

func: Callable[[...], ndarray]
Function the computes parameter values. See Notes for requirements

reps: int = 1000
Number of bootstrap replications

method: 'basic' | 'percentile' | 'studentized' | 'norm' | 'bc' | 'bca' = 'basic'
One of 'basic', 'percentile', 'studentized', 'norm' (identical to 'var', 'cov'), 'bc' (identical to 'debiased', 'bias-corrected'), or 'bca'

size: float = 0.95
Coverage of confidence interval

tail: 'two' | 'upper' | 'lower' = 'two'
One of 'two', 'upper' or 'lower'.

reuse: bool = False
Flag indicating whether to reuse previously computed bootstrap results. This allows alternative methods to be compared without rerunning the bootstrap simulation. Reuse is ignored if reps is not the same across multiple runs, func changes across calls, or method is 'studentized'.

sampling: 'nonparametric' | 'semi-parametric' | 'semi' | 'parametric' | 'semiparametric' = 'nonparametric'
Type of sampling to use: 'nonparametric', 'semi-parametric' (or 'semi') or 'parametric'. The default is 'nonparametric'. See notes about the changes to func required when using 'semi' or 'parametric'.

extra_kwargs: dict[str, Any] | None = None
Extra keyword arguments to use when calling func and std_err_func, when appropriate

std_err_func: Callable[[...], ndarray | DataFrame | Series] | None = None
Function to use when standardizing estimated parameters when using the studentized bootstrap. Providing an analytical function eliminates the need for a nested bootstrap

studentize_reps: int = 1000
Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when `std_err_func` is provided

Returns
Computed confidence interval. Row 0 contains the lower bounds, and row 1 contains the upper bounds. Each column corresponds to a parameter. When tail is 'lower', all upper bounds are inf. Similarly, 'upper' sets all lower bounds to -inf.

Return type
`numpy.ndarray`

Examples

```
>>> import numpy as np
>>> def func(x):
...     return x.mean(0)
>>> y = np.random.randn(1000, 2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(y)
>>> ci = bs.conf_int(func, 1000)
```

Notes

When there are no extra keyword arguments, the function is called

```
func(*args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func.

The standard error function, if provided, must return a vector of parameter standard errors and is called

```
std_err_func(params, *args, **kwargs)
```

where `params` is the vector of estimated parameters using the same bootstrap data as in args and kwargs.

The bootstraps are:

- 'basic' - Basic confidence using the estimated parameter and difference between the estimated parameter and the bootstrap parameters
- 'percentile' - Direct use of bootstrap percentiles
- 'norm' - Makes use of normal approximation and bootstrap covariance estimator
- 'studentized' - Uses either a standard error function or a nested bootstrap to estimate percentiles and the bootstrap covariance for scale
- 'bc' - Bias corrected using estimate bootstrap bias correction
- 'bca' - Bias corrected and accelerated, adding acceleration parameter to 'bc' method

arch.bootstrap.StationaryBootstrap.cov

StationaryBootstrap.**cov**(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = **1000**, recenter: *bool* = **True**, extra_kwargs: *dict[str, Any] | None* = **None**) → float | ndarray

Compute parameter covariance using bootstrap

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Callable function that returns the statistic of interest as a 1-d array

reps: int = 1000
Number of bootstrap replications

recenter: bool = True
Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.

extra_kwargs: dict[str, Any] | None = None
Dictionary of extra keyword arguments to pass to func

Returns
Bootstrap covariance estimator

Return type
`numpy.ndarray`

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Examples

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> cov = bs.cov(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> cov = bs.cov(func, 1000, extra_kwargs={'stat':'var'})
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

arch.bootstrap.StationaryBootstrap.get_state

`StationaryBootstrap.get_state()` → tuple[str, ndarray, int, int, float] | Mapping[str, Any]

Gets the state of the bootstrap's random number generator

Returns

Dictionary containing the state.

Return type

`dict`

arch.bootstrap.StationaryBootstrap.reset

`StationaryBootstrap.reset(use_seed: bool = True)` → None

Resets the bootstrap to either its initial state or the last seed.

Parameters

use_seed: bool = True

Flag indicating whether to use the last seed if provided. If False or if no seed has been set, the bootstrap will be reset to the initial state. Default is True

arch.bootstrap.StationaryBootstrap.seed

StationaryBootstrap.**seed**(value: *int | list[int] | ndarray*) → None

Reseeds the bootstrap's random number generator

Parameters

value: int | list[int] | ndarray
Value to use as the seed.

arch.bootstrap.StationaryBootstrap.set_state

StationaryBootstrap.**set_state**(state: *tuple[str, ndarray, int, int, float] | dict[str, Any]*) → None

Sets the state of the bootstrap's random number generator

Parameters

state: tuple[str, ndarray, int, int, float] | dict[str, Any]
Dictionary or tuple containing the state.

arch.bootstrap.StationaryBootstrap.update_indices

StationaryBootstrap.**update_indices**() → ndarray

Update indices for the next iteration of the bootstrap. This must be overridden when creating new bootstraps.

arch.bootstrap.StationaryBootstrap.var

StationaryBootstrap.**var**(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = *1000*, recenter: *bool* = *True*, extra_kwargs: *dict[str, Any] | None* = *None*) → float | ndarray

Compute parameter variance using bootstrap

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Callable function that returns the statistic of interest as a 1-d array

reps: int = 1000
Number of bootstrap replications

recenter: bool = True
Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.

extra_kwargs: dict[str, Any] | None = None
Dictionary of extra keyword arguments to pass to func

Returns
Bootstrap variance estimator

Return type
`numpy.ndarray`

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Examples

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> variances = bs.var(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> variances = bs.var(func, 1000, extra_kwargs={'stat': 'var'})
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

Properties

generator	Set or get the instance PRNG
index	The current index of the bootstrap
random_state	Set or get the instance random state
state	Set or get the generator's state

arch.bootstrap.StationaryBootstrap.generator

property `StationaryBootstrap.generator` : Generator | RandomState

Set or get the instance PRNG

Parameters

seed : {Generator, RandomState}, *optional*
Generator or RandomState used to produce the pseudo-random values used in the bootstrap

Returns
The instance of the Generator or RandomState instance used by bootstrap

Return type
{Generator, RandomState}

arch.bootstrap.StationaryBootstrap.index

property `StationaryBootstrap.index` : ndarray | tuple[ndarray, ...] | tuple[list[ndarray], dict[str, ndarray]]

The current index of the bootstrap

arch.bootstrap.StationaryBootstrap.random_state

property `StationaryBootstrap.random_state` : Generator | RandomState

Set or get the instance random state

Parameters

random_state : numpy.random.RandomState
RandomState instance used by bootstrap

Returns
RandomState instance used by bootstrap

Return type
numpy.random.RandomState

arch.bootstrap.StationaryBootstrap.state

property `StationaryBootstrap.state` : tuple[str, ndarray, int, int, float] | Mapping[str, Any]

Set or get the generator's state

Returns
A tuple or dictionary containing the generator's state. If using a `RandomState`, the value returned is a tuple. Otherwise it is a dictionary.

Return type
{tuple, dict}

2.9.2 arch.bootstrap.CircularBlockBootstrap

`class arch.bootstrap.CircularBlockBootstrap`(block_size: *int*, *args: *ArrayLike*, random_state: *RandomState* | *None* = **None**, seed: *None* | *int* | *Generator* | *RandomState* = **None**, **kwargs: *ArrayLike*)

Bootstrap using blocks of the same length with end-to-start wrap around

Parameters

block_size: int
Size of block to use

***args: ArrayLike**
Positional arguments to bootstrap

seed: None | int | Generator | RandomState = None
Seed to use to ensure reproducable results. If an int, passes the value to value to `np.random.default_rng`. If None, a fresh Generator is constructed with system-provided entropy.

random_state: RandomState | None = None

`RandomState` to use to ensure reproducable results. Cannot be used with `seed`

Deprecated since version 5.0: The random_state keyword argument has been deprecated. Use seed instead.

****kwargs: ArrayLike**
Keyword arguments to bootstrap

`data`
Two-element tuple with the pos_data in the first position and kw_data in the second (pos_data, kw_data)

Type
`tuple`

`pos_data`
Tuple containing the positional arguments (in the order entered)

Type
`tuple`

`kw_data`
Dictionary containing the keyword arguments

Type
`dict`

Notes

Supports numpy arrays and pandas Series and DataFrames. Data returned has the same type as the input date.

Data entered using keyword arguments is directly accessibly as an attribute.

To ensure a reproducible bootstrap, you must set the `random_state` attribute after the bootstrap has been created. See the example below. Note that `random_state` is a reserved keyword and any variable passed using this keyword must be an instance of `RandomState`.

See also

arch.bootstrap.optimal_block_length
Optimal block length estimation

arch.bootstrap.StationaryBootstrap
Politis and Romano's bootstrap with exp. distributed block lengths

Examples

Data can be accessed in a number of ways. Positional data is retained in the same order as it was entered when the bootstrap was initialized. Keyword data is available both as an attribute or using a dictionary syntax on kw_data.

```
>>> from arch.bootstrap import CircularBlockBootstrap
>>> from numpy.random import standard_normal
>>> y = standard_normal((500, 1))
>>> x = standard_normal((500, 2))
>>> z = standard_normal(500)
>>> bs = CircularBlockBootstrap(17, x, y=y, z=z)
>>> for data in bs.bootstrap(100):
...     bs_x = data[0][0]
...     bs_y = data[1]['y']
...     bs_z = bs.z
```

Set the random_state if reproducibility is required

```
>>> from numpy.random import RandomState
>>> rs = RandomState(1234)
>>> bs = CircularBlockBootstrap(17, x, y=y, z=z, random_state=rs)
```

Methods

apply(func[, reps, extra_kwargs])	Applies a function to bootstrap replicated data
bootstrap(reps)	Iterator for use when bootstrapping
clone(*args[, seed])	Clones the bootstrap using different data with a fresh prng.
conf_int(func[, reps, method, size, tail, ...])	**param func** Function the computes parameter values. See Notes for requirements
cov(func[, reps, recenter, extra_kwargs])	Compute parameter covariance using bootstrap
get_state()	Gets the state of the bootstrap's random number generator
reset([use_seed])	Resets the bootstrap to either its initial state or the last seed.
seed(value)	Reseeds the bootstrap's random number generator
set_state(state)	Sets the state of the bootstrap's random number generator
update_indices()	Update indices for the next iteration of the bootstrap.
var(func[, reps, recenter, extra_kwargs])	Compute parameter variance using bootstrap

arch.bootstrap.CircularBlockBootstrap.apply

CircularBlockBootstrap.**apply**(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = *1000*, extra_kwargs: *dict[str, Any] | None* = *None*) → ndarray

Applies a function to bootstrap replicated data

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Function the computes parameter values. See Notes for requirements

reps: int = 1000
Number of bootstrap replications

extra_kwargs: dict[str, Any] | None = None
Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

Returns
reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

Return type
`numpy.ndarray`

Notes

When there are no extra keyword arguments, the function is called

```
func(params, *args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func

Examples

```
>>> import numpy as np
>>> x = np.random.randn(1000,2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(x)
>>> def func(y):
...     return y.mean(0)
>>> results = bs.apply(func, 100)
```

arch.bootstrap.CircularBlockBootstrap.bootstrap

CircularBlockBootstrap.**bootstrap**(reps: *int*) → Generator[tuple[tuple[ndarray | DataFrame | Series, ...], dict[str, ndarray | DataFrame | Series]], None, None]

Iterator for use when bootstrapping

Parameters

reps: int
Number of bootstrap replications

Returns
Generator to iterate over in bootstrap calculations

Return type
generator

Examples

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> bs = IIDBootstrap(np.arange(100), x=np.random.randn(100))
>>> for posdata, kwdata in bs.bootstrap(1000):
...     # Do something with the positional data and/or keyword data
...     pass
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

---

Notes

The iterator returns a tuple containing the data entered in positional arguments as a tuple and the data entered using keywords as a dictionary

arch.bootstrap.CircularBlockBootstrap.clone

CircularBlockBootstrap.**clone**(*args: *ArrayLike*, seed: *None* | *int* | *Generator* | *RandomState* = **None**, **kwargs: *ArrayLike*) → *CircularBlockBootstrap*

Clones the bootstrap using different data with a fresh prng.

Parameters

***args: ArrayLike**
Positional arguments to bootstrap

seed: None | int | Generator | RandomState = None
The seed value to pass to the closed generator

****kwargs: ArrayLike**
Keyword arguments to bootstrap

Returns
Bootstrap instance

Return type
bs

arch.bootstrap.CircularBlockBootstrap.conf_int

`CircularBlockBootstrap.conf_int(func: Callable[[...], ndarray], reps: int = 1000, method: 'basic' | 'percentile' | 'studentized' | 'norm' | 'bc' | 'bca' = 'basic', size: float = 0.95, tail: 'two' | 'upper' | 'lower' = 'two', extra_kwargs: dict[str, Any] | None = None, reuse: bool = False, sampling: 'nonparametric' | 'semi-parametric' | 'semi' | 'parametric' | 'semiparametric' = 'nonparametric', std_err_func: Callable[[...], ndarray | DataFrame | Series] | None = None, studentize_reps: int = 1000) → ndarray`

Parameters

func: Callable[[...], ndarray]
Function the computes parameter values. See Notes for requirements

reps: int = 1000
Number of bootstrap replications

method: 'basic' | 'percentile' | 'studentized' | 'norm' | 'bc' | 'bca' = 'basic'
One of 'basic', 'percentile', 'studentized', 'norm' (identical to 'var', 'cov'), 'bc' (identical to 'debiased', 'bias-corrected'), or 'bca'

size: float = 0.95
Coverage of confidence interval

tail: 'two' | 'upper' | 'lower' = 'two'
One of 'two', 'upper' or 'lower'.

reuse: bool = False
Flag indicating whether to reuse previously computed bootstrap results. This allows alternative methods to be compared without rerunning the bootstrap simulation. Reuse is ignored if reps is not the same across multiple runs, func changes across calls, or method is 'studentized'.

sampling: 'nonparametric' | 'semi-parametric' | 'semi' | 'parametric' | 'semiparametric' = 'nonparametric'
Type of sampling to use: 'nonparametric', 'semi-parametric' (or 'semi') or 'parametric'. The default is 'nonparametric'. See notes about the changes to func required when using 'semi' or 'parametric'.

extra_kwargs: dict[str, Any] | None = None
Extra keyword arguments to use when calling func and std_err_func, when appropriate

std_err_func: Callable[[...], ndarray | DataFrame | Series] | None = None
Function to use when standardizing estimated parameters when using the studentized bootstrap. Providing an analytical function eliminates the need for a nested bootstrap

studentize_reps: int = 1000
Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when `std_err_func` is provided

Returns

Computed confidence interval. Row 0 contains the lower bounds, and row 1 contains the upper bounds. Each column corresponds to a parameter. When tail is 'lower', all upper bounds are inf. Similarly, 'upper' sets all lower bounds to -inf.

Return type

`numpy.ndarray`

Examples

```
>>> import numpy as np
>>> def func(x):
...     return x.mean(0)
>>> y = np.random.randn(1000, 2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(y)
>>> ci = bs.conf_int(func, 1000)
```

Notes

When there are no extra keyword arguments, the function is called

```
func(*args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func.

The standard error function, if provided, must return a vector of parameter standard errors and is called

```
std_err_func(params, *args, **kwargs)
```

where `params` is the vector of estimated parameters using the same bootstrap data as in args and kwargs.

The bootstraps are:

- 'basic' - Basic confidence using the estimated parameter and difference between the estimated parameter and the bootstrap parameters
- 'percentile' - Direct use of bootstrap percentiles
- 'norm' - Makes use of normal approximation and bootstrap covariance estimator
- 'studentized' - Uses either a standard error function or a nested bootstrap to estimate percentiles and the bootstrap covariance for scale
- 'bc' - Bias corrected using estimate bootstrap bias correction
- 'bca' - Bias corrected and accelerated, adding acceleration parameter to 'bc' method

arch.bootstrap.CircularBlockBootstrap.cov

CircularBlockBootstrap.**cov**(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = *1000*, recenter: *bool* = *True*, extra_kwargs: *dict[str, Any] | None* = *None*) → float | ndarray

Compute parameter covariance using bootstrap

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Callable function that returns the statistic of interest as a 1-d array

reps: int = 1000
Number of bootstrap replications

recenter: bool = True
Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.

extra_kwargs: dict[str, Any] | None = None
Dictionary of extra keyword arguments to pass to func

Returns
Bootstrap covariance estimator

Return type
numpy.ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Examples

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> cov = bs.cov(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> cov = bs.cov(func, 1000, extra_kwargs={'stat':'var'})
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

arch.bootstrap.CircularBlockBootstrap.get_state

CircularBlockBootstrap.**get_state**() → tuple[str, ndarray, int, int, float] | Mapping[str, Any]

Gets the state of the bootstrap's random number generator

Returns
Dictionary containing the state.

Return type
dict

arch.bootstrap.CircularBlockBootstrap.reset

CircularBlockBootstrap.**reset**(use_seed: *bool* = ***True***) → None

Resets the bootstrap to either its initial state or the last seed.

Parameters

use_seed: bool = True
Flag indicating whether to use the last seed if provided. If False or if no seed has been set, the bootstrap will be reset to the initial state. Default is True

arch.bootstrap.CircularBlockBootstrap.seed

CircularBlockBootstrap.**seed**(value: *int* | *list[int]* | *ndarray*) → None

Reseeds the bootstrap's random number generator

Parameters

value: int | list[int] | ndarray
Value to use as the seed.

arch.bootstrap.CircularBlockBootstrap.set_state

CircularBlockBootstrap.**set_state**(state: *tuple[str, ndarray, int, int, float]* | *dict[str, Any]*) → None

Sets the state of the bootstrap's random number generator

Parameters

state: tuple[str, ndarray, int, int, float] | dict[str, Any]
Dictionary or tuple containing the state.

arch.bootstrap.CircularBlockBootstrap.update_indices

CircularBlockBootstrap.**update_indices**() → ndarray

Update indices for the next iteration of the bootstrap. This must be overridden when creating new bootstraps.

arch.bootstrap.CircularBlockBootstrap.var

CircularBlockBootstrap.**var**(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = ***1000***, recenter: *bool* = ***True***, extra_kwargs: *dict[str, Any] | None* = ***None***) → float | ndarray

Compute parameter variance using bootstrap

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Callable function that returns the statistic of interest as a 1-d array

reps: int = 1000
Number of bootstrap replications

recenter: bool = True
Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.

extra_kwargs: dict[str, Any] | None = None
Dictionary of extra keyword arguments to pass to func

Returns
Bootstrap variance estimator

Return type
numpy.ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Examples

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> variances = bs.var(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
```

(continues on next page)

(continued from previous page)

```
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> variances = bs.var(func, 1000, extra_kwargs={'stat': 'var'})
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

Properties

generator	Set or get the instance PRNG
index	The current index of the bootstrap
random_state	Set or get the instance random state
state	Set or get the generator's state

arch.bootstrap.CircularBlockBootstrap.generator

property `CircularBlockBootstrap.`**`generator`** : Generator | RandomState

Set or get the instance PRNG

Parameters

seed : **{Generator, RandomState}**, *optional*
Generator or RandomState used to produce the pseudo-random values used in the bootstrap

Returns
The instance of the Generator or RandomState instance used by bootstrap

Return type
`{Generator, RandomState}`

arch.bootstrap.CircularBlockBootstrap.index

property `CircularBlockBootstrap.`**`index`** : ndarray | tuple[ndarray, ...] | tuple[list[ndarray], dict[str, ndarray]]

The current index of the bootstrap

arch.bootstrap.CircularBlockBootstrap.random_state

property `CircularBlockBootstrap.`**`random_state`** : Generator | RandomState

Set or get the instance random state

Parameters

random_state : `numpy.random.RandomState`
RandomState instance used by bootstrap

Returns
RandomState instance used by bootstrap

Return type
`numpy.random.RandomState`

arch.bootstrap.CircularBlockBootstrap.state

property `CircularBlockBootstrap.state` : tuple[str, ndarray, int, int, float] | Mapping[str, Any]

Set or get the generator's state

Returns
A tuple or dictionary containing the generator's state. If using a `RandomState`, the value returned is a tuple. Otherwise it is a dictionary.

Return type
`{tuple, dict}`

2.9.3 arch.bootstrap.MovingBlockBootstrap

class `arch.bootstrap.MovingBlockBootstrap`(block_size: *int*, *args: *ArrayLike*, random_state: *RandomState* | *None* = **None**, seed: *None* | *int* | *Generator* | *RandomState* = **None**, **kwargs: *ArrayLike*)

Bootstrap using blocks of the same length without wrap around

Parameters

block_size: int
Size of block to use

***args: ArrayLike**
Positional arguments to bootstrap

seed: None | int | Generator | RandomState = None
Seed to use to ensure reproducable results. If an int, passes the value to value to `np.random.default_rng`. If None, a fresh Generator is constructed with system-provided entropy.

random_state: RandomState | None = None

`RandomState` to use to ensure reproducable results. Cannot be used with `seed`

Deprecated since version 5.0: The random_state keyword argument has been deprecated. Use seed instead.

****kwargs: ArrayLike**
Keyword arguments to bootstrap

data

Two-element tuple with the pos_data in the first position and kw_data in the second (pos_data, kw_data)

Type
`tuple`

pos_data

Tuple containing the positional arguments (in the order entered)

Type
`tuple`

kw_data

Dictionary containing the keyword arguments

Type

dict

Notes

Supports numpy arrays and pandas Series and DataFrames. Data returned has the same type as the input date.

Data entered using keyword arguments is directly accessibly as an attribute.

To ensure a reproducible bootstrap, you must set the `random_state` attribute after the bootstrap has been created. See the example below. Note that `random_state` is a reserved keyword and any variable passed using this keyword must be an instance of `RandomState`.

See also

arch.bootstrap.optimal_block_length
Optimal block length estimation

arch.bootstrap.StationaryBootstrap
Politis and Romano's bootstrap with exp. distributed block lengths

arch.bootstrap.CircularBlockBootstrap
Circular (wrap-around) bootstrap

Examples

Data can be accessed in a number of ways. Positional data is retained in the same order as it was entered when the bootstrap was initialized. Keyword data is available both as an attribute or using a dictionary syntax on kw_data.

```
>>> from arch.bootstrap import MovingBlockBootstrap
>>> from numpy.random import standard_normal
>>> y = standard_normal((500, 1))
>>> x = standard_normal((500,2))
>>> z = standard_normal(500)
>>> bs = MovingBlockBootstrap(7, x, y=y, z=z)
>>> for data in bs.bootstrap(100):
...     bs_x = data[0][0]
...     bs_y = data[1]['y']
...     bs_z = bs.z
```

Set the random_state if reproducibility is required

```
>>> from numpy.random import RandomState
>>> rs = RandomState(1234)
>>> bs = MovingBlockBootstrap(7, x, y=y, z=z, random_state=rs)
```

Methods

apply(func[, reps, extra_kwargs])	Applies a function to bootstrap replicated data
bootstrap(reps)	Iterator for use when bootstrapping
clone(*args[, seed])	Clones the bootstrap using different data with a fresh prng.
conf_int(func[, reps, method, size, tail, ...])	**param func** Function the computes parameter values. See Notes for requirements
cov(func[, reps, recenter, extra_kwargs])	Compute parameter covariance using bootstrap
get_state()	Gets the state of the bootstrap's random number generator
reset([use_seed])	Resets the bootstrap to either its initial state or the last seed.
seed(value)	Reseeds the bootstrap's random number generator
set_state(state)	Sets the state of the bootstrap's random number generator
update_indices()	Update indices for the next iteration of the bootstrap.
var(func[, reps, recenter, extra_kwargs])	Compute parameter variance using bootstrap

arch.bootstrap.MovingBlockBootstrap.apply

MovingBlockBootstrap.**apply**(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = *1000*, extra_kwargs: *dict[str, Any] | None* = *None*) → ndarray

Applies a function to bootstrap replicated data

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Function the computes parameter values. See Notes for requirements

reps: int = 1000
Number of bootstrap replications

extra_kwargs: dict[str, Any] | None = None
Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

Returns
reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

Return type
`numpy.ndarray`

Notes

When there are no extra keyword arguments, the function is called

```
func(params, *args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func

Examples

```
>>> import numpy as np
>>> x = np.random.randn(1000,2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(x)
>>> def func(y):
...     return y.mean(0)
>>> results = bs.apply(func, 100)
```

arch.bootstrap.MovingBlockBootstrap.bootstrap

`MovingBlockBootstrap.bootstrap`(reps: *int*) → Generator[tuple[tuple[ndarray | DataFrame | Series, ...], dict[str, ndarray | DataFrame | Series]], None, None]

Iterator for use when bootstrapping

Parameters

reps: int
Number of bootstrap replications

Returns
Generator to iterate over in bootstrap calculations

Return type
generator

Examples

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> bs = IIDBootstrap(np.arange(100), x=np.random.randn(100))
>>> for posdata, kwdata in bs.bootstrap(1000):
...     # Do something with the positional data and/or keyword data
...     pass
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

Notes

The iterator returns a tuple containing the data entered in positional arguments as a tuple and the data entered using keywords as a dictionary

arch.bootstrap.MovingBlockBootstrap.clone

MovingBlockBootstrap.**clone**(**args: ArrayLike*, *seed: None | int | Generator | RandomState* = **None**, ***kwargs: ArrayLike*) → *CircularBlockBootstrap*

Clones the bootstrap using different data with a fresh prng.

Parameters

***args: ArrayLike**
Positional arguments to bootstrap

seed: None | int | Generator | RandomState = None
The seed value to pass to the closed generator

****kwargs: ArrayLike**
Keyword arguments to bootstrap

Returns
Bootstrap instance

Return type
`bs`

arch.bootstrap.MovingBlockBootstrap.conf_int

MovingBlockBootstrap.**conf_int**(*func: Callable[[...], ndarray]*, *reps: int* = **1000**, *method: 'basic' | 'percentile' | 'studentized' | 'norm' | 'bc' | 'bca'* = `'basic'`, *size: float* = **0.95**, *tail: 'two' | 'upper' | 'lower'* = `'two'`, *extra_kwargs: dict[str, Any] | None* = **None**, *reuse: bool* = **False**, *sampling: 'nonparametric' | 'semi-parametric' | 'semi' | 'parametric' | 'semiparametric'* = `'nonparametric'`, *std_err_func: Callable[[...], ndarray | DataFrame | Series] | None* = **None**, *studentize_reps: int* = **1000**) → ndarray

Parameters

func: Callable[[...], ndarray]
Function the computes parameter values. See Notes for requirements

reps: int = 1000
Number of bootstrap replications

method: 'basic' | 'percentile' | 'studentized' | 'norm' | 'bc' | 'bca' = 'basic'
One of 'basic', 'percentile', 'studentized', 'norm' (identical to 'var', 'cov'), 'bc' (identical to 'debiased', 'bias-corrected'), or 'bca'

size: float = 0.95
Coverage of confidence interval

tail: 'two' | 'upper' | 'lower' = 'two'
One of 'two', 'upper' or 'lower'.

reuse: bool = False
Flag indicating whether to reuse previously computed bootstrap results. This allows alternative methods to be compared without rerunning the bootstrap simulation. Reuse is ignored if reps is not the same across multiple runs, func changes across calls, or method is 'studentized'.

sampling: 'nonparametric' | 'semi-parametric' | 'semi' | 'parametric' | 'semiparametric' = 'nonparametric'
Type of sampling to use: 'nonparametric', 'semi-parametric' (or 'semi') or 'parametric'. The default is 'nonparametric'. See notes about the changes to func required when using 'semi' or 'parametric'.

extra_kwargs: dict[str, Any] | None = None
Extra keyword arguments to use when calling func and std_err_func, when appropriate

std_err_func: Callable[[...], ndarray | DataFrame | Series] | None = None
Function to use when standardizing estimated parameters when using the studentized bootstrap. Providing an analytical function eliminates the need for a nested bootstrap

studentize_reps: int = 1000
Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when `std_err_func` is provided

Returns
Computed confidence interval. Row 0 contains the lower bounds, and row 1 contains the upper bounds. Each column corresponds to a parameter. When tail is 'lower', all upper bounds are inf. Similarly, 'upper' sets all lower bounds to -inf.

Return type
`numpy.ndarray`

Examples

```
>>> import numpy as np
>>> def func(x):
...     return x.mean(0)
>>> y = np.random.randn(1000, 2)
>>> from arch.bootstrap import IIDBootstrap
>>> bs = IIDBootstrap(y)
>>> ci = bs.conf_int(func, 1000)
```

Notes

When there are no extra keyword arguments, the function is called

```
func(*args, **kwargs)
```

where args and kwargs are the bootstrap version of the data provided when setting up the bootstrap. When extra keyword arguments are used, these are appended to kwargs before calling func.

The standard error function, if provided, must return a vector of parameter standard errors and is called

```
std_err_func(params, *args, **kwargs)
```

where `params` is the vector of estimated parameters using the same bootstrap data as in args and kwargs.

The bootstraps are:

- 'basic' - Basic confidence using the estimated parameter and difference between the estimated parameter and the bootstrap parameters
- 'percentile' - Direct use of bootstrap percentiles
- 'norm' - Makes use of normal approximation and bootstrap covariance estimator
- 'studentized' - Uses either a standard error function or a nested bootstrap to estimate percentiles and the bootstrap covariance for scale
- 'bc' - Bias corrected using estimate bootstrap bias correction
- 'bca' - Bias corrected and accelerated, adding acceleration parameter to 'bc' method

arch.bootstrap.MovingBlockBootstrap.cov

MovingBlockBootstrap.**cov**(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = ***1000***, recenter: *bool* = ***True***, extra_kwargs: *dict[str, Any] | None* = ***None***) → float | ndarray

Compute parameter covariance using bootstrap

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Callable function that returns the statistic of interest as a 1-d array

reps: int = 1000
Number of bootstrap replications

recenter: bool = True
Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.

extra_kwargs: dict[str, Any] | None = None
Dictionary of extra keyword arguments to pass to func

Returns
Bootstrap covariance estimator

Return type
`numpy.ndarray`

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Examples

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> cov = bs.cov(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> cov = bs.cov(func, 1000, extra_kwargs={'stat':'var'})
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

arch.bootstrap.MovingBlockBootstrap.get_state

`MovingBlockBootstrap.get_state()` → tuple[str, ndarray, int, int, float] | Mapping[str, Any]

Gets the state of the bootstrap's random number generator

Returns

Dictionary containing the state.

Return type

`dict`

arch.bootstrap.MovingBlockBootstrap.reset

`MovingBlockBootstrap.reset(use_seed: bool = True)` → None

Resets the bootstrap to either its initial state or the last seed.

Parameters

use_seed: bool = True

Flag indicating whether to use the last seed if provided. If False or if no seed has been set, the bootstrap will be reset to the initial state. Default is True

arch.bootstrap.MovingBlockBootstrap.seed

MovingBlockBootstrap.**seed**(value: *int* | *list[int]* | *ndarray*) → None

Reseeds the bootstrap's random number generator

Parameters

value: int | list[int] | ndarray
Value to use as the seed.

arch.bootstrap.MovingBlockBootstrap.set_state

MovingBlockBootstrap.**set_state**(state: *tuple[str, ndarray, int, int, float]* | *dict[str, Any]*) → None

Sets the state of the bootstrap's random number generator

Parameters

state: tuple[str, ndarray, int, int, float] | dict[str, Any]
Dictionary or tuple containing the state.

arch.bootstrap.MovingBlockBootstrap.update_indices

MovingBlockBootstrap.**update_indices**() → ndarray

Update indices for the next iteration of the bootstrap. This must be overridden when creating new bootstraps.

arch.bootstrap.MovingBlockBootstrap.var

MovingBlockBootstrap.**var**(func: *Callable[[...], ndarray | DataFrame | Series]*, reps: *int* = *1000*, recenter: *bool* = *True*, extra_kwargs: *dict[str, Any] | None* = *None*) → float | ndarray

Compute parameter variance using bootstrap

Parameters

func: Callable[[...], ndarray | DataFrame | Series]
Callable function that returns the statistic of interest as a 1-d array

reps: int = 1000
Number of bootstrap replications

recenter: bool = True
Whether to center the bootstrap variance estimator on the average of the bootstrap samples (True) or to center on the original sample estimate (False). Default is True.

extra_kwargs: dict[str, Any] | None = None
Dictionary of extra keyword arguments to pass to func

Returns
Bootstrap variance estimator

Return type
`numpy.ndarray`

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and **args* and ***kwargs* are data used in the the bootstrap. The first argument, params, will be none when called using the original data, and will contain the estimate computed using the original data in bootstrap replications. This parameter is passed to allow parametric bootstrap simulation.

Examples

Bootstrap covariance of the mean

```
>>> from arch.bootstrap import IIDBootstrap
>>> import numpy as np
>>> def func(x):
...     return x.mean(axis=0)
>>> y = np.random.randn(1000, 3)
>>> bs = IIDBootstrap(y)
>>> variances = bs.var(func, 1000)
```

Bootstrap covariance using a function that takes additional input

```
>>> def func(x, stat='mean'):
...     if stat=='mean':
...         return x.mean(axis=0)
...     elif stat=='var':
...         return x.var(axis=0)
>>> variances = bs.var(func, 1000, extra_kwargs={'stat': 'var'})
```

Note

Note this is a generic example and so the class used should be the name of the required bootstrap

Properties

generator	Set or get the instance PRNG
index	The current index of the bootstrap
random_state	Set or get the instance random state
state	Set or get the generator's state

arch.bootstrap.MovingBlockBootstrap.generator

property `MovingBlockBootstrap.generator` : Generator | RandomState

Set or get the instance PRNG

Parameters

seed : **{Generator, RandomState}**, ***optional***
Generator or RandomState used to produce the pseudo-random values used in the bootstrap

Returns
The instance of the Generator or RandomState instance used by bootstrap

Return type
`{Generator, RandomState}`

arch.bootstrap.MovingBlockBootstrap.index

property `MovingBlockBootstrap.index` : ndarray | tuple[ndarray, ...] | tuple[list[ndarray], dict[str, ndarray]]

The current index of the bootstrap

arch.bootstrap.MovingBlockBootstrap.random_state

property `MovingBlockBootstrap.random_state` : Generator | RandomState

Set or get the instance random state

Parameters

random_state : `numpy.random.RandomState`
RandomState instance used by bootstrap

Returns
RandomState instance used by bootstrap

Return type
`numpy.random.RandomState`

arch.bootstrap.MovingBlockBootstrap.state

property `MovingBlockBootstrap.state` : tuple[str, ndarray, int, int, float] | Mapping[str, Any]

Set or get the generator's state

Returns
A tuple or dictionary containing the generator's state. If using a `RandomState`, the value returned is a tuple. Otherwise it is a dictionary.

Return type
`{tuple, dict}`

2.9.4 arch.bootstrap.optimal_block_length

`arch.bootstrap.optimal_block_length(x: ArrayLike1D | ArrayLike2D) → pd.DataFrame`

Estimate optimal window length for time-series bootstraps

Parameters

x: ArrayLike1D | ArrayLike2D
A one-dimensional or two-dimensional array-like. Operates columns by column if 2-dimensional.

Returns
A DataFrame with two columns *b_sb*, the estimated optimal block size for the Stationary Bootstrap and *b_cb*, the estimated optimal block size for the circular bootstrap.

Return type
`pandas.DataFrame`

See also

arch.bootstrap.StationaryBootstrap
Politis and Romano's bootstrap with exp. distributed block lengths

arch.bootstrap.CircularBlockBootstrap
Circular (wrap-around) bootstrap

Notes

Algorithm described in ([1]) its correction ([2]) depends on a tuning parameter m, which is chosen as the first value where k_n consecutive autocorrelations of x are all inside a conservative band of $\pm 2\sqrt{\log_{10}(n)/n}$ where n is the sample size. The maximum value of m is set to $\lceil \sqrt{n} + k_n \rceil$ where $k_n = \max(5, \log_{10}(n))$. The block length is then computed as

$$b_i^{OPT} = \left(\frac{2g^2}{d_i} n\right)^{\frac{1}{3}}$$

where

$$\begin{aligned} g &= \sum_{k=-m}^{m} h\left(\frac{k}{m}\right)|k|\hat{\gamma}_k \\ h(x) &= \min(1, 2(1-|x|)) \\ d_i &= c_i \left(\hat{\sigma}^2\right)^2 \\ \hat{\sigma}^2 &= \sum_{k=-m}^{m} h\left(\frac{k}{m}\right)\hat{\gamma}_k \\ \hat{\gamma}_i &= n^{-1}\sum_{k=i+1}^{n}\left(x_k - \bar{x}\right)\left(x_{k-i} - \bar{x}\right) \end{aligned}$$

and the two remaining constants c_i are 2 for the Stationary bootstrap and 4/3 for the Circular bootstrap.

[1] Dimitris N. Politis & Halbert White (2004) Automatic Block-Length Selection for the Dependent Bootstrap, Econometric Reviews, 23:1, 53-70, DOI: 10.1081/ETC-120028836.

[2] Andrew Patton , Dimitris N. Politis & Halbert White (2009) Correction to "Automatic Block-Length Selection for the Dependent Bootstrap" by D. Politis and H. White, Econometric Reviews, 28:4, 372-375, DOI: 10.1080/07474930802459016.

Some of the tuning parameters are taken from Andrew Patton's MATLAB program that computes the optimal block length. The block lengths do not match this implementation since the autocovariances and autocorrelations are all computed using the maximum sample length rather than a common sampling length.

References

2.10 References

The bootstrap is a large area with a number of high-quality books. Leading references include

References

Articles used in the creation of this module include

CHAPTER

THREE

MULTIPLE COMPARISON PROCEDURES

This module contains a set of bootstrap-based multiple comparison procedures. These are designed to allow multiple models to be compared while controlling a the Familywise Error Rate, which is similar to the size of a test.

3.1 Multiple Comparisons

This setup code is required to run in an IPython notebook

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn

seaborn.set_style("darkgrid")
plt.rc("figure", figsize=(16, 6))
plt.rc("savefig", dpi=90)
plt.rc("font", family="sans-serif")
plt.rc("font", size=14)
```

[2]:
```
# Reproducability
import numpy as np

gen = np.random.default_rng(23456)
# Common seed used throughout
seed = gen.integers(0, 2**31 - 1)
```

The multiple comparison procedures all allow for examining aspects of superior predictive ability. There are three available:

- `SPA` - The test of Superior Predictive Ability, also known as the Reality Check (and accessible as `RealityCheck`) or the bootstrap data snooper, examines whether any model in a set of models can outperform a benchmark.
- `StepM` - The stepwise multiple testing procedure uses sequential testing to determine which models are superior to a benchmark.
- `MCS` - The model confidence set which computes the set of models which with performance indistinguishable from others in the set.

All procedures take **losses** as inputs. That is, smaller values are preferred to larger values. This is common when evaluating forecasting models where the loss function is usually defined as a positive function of the forecast error that is increasing in the absolute error. Leading examples are Mean Square Error (MSE) and Mean Absolute Deviation (MAD).

3.1.1 The test of Superior Predictive Ability (SPA)

This procedure requires a t-element array of benchmark losses and a t by k-element array of model losses. The null hypothesis is that no model is better than the benchmark, or

$$H_0 : \max_i E[L_i] \geq E[L_{bm}]$$

where L_i is the loss from model i and L_{bm} is the loss from the benchmark model.

This procedure is normally used when there are many competing forecasting models such as in the study of technical trading rules. The example below will make use of a set of models which are all equivalently good to a benchmark model and will serve as a *size study*.

Study Design

The study will make use of a measurement error in predictors to produce a large set of correlated variables that all have equal expected MSE. The benchmark will have identical measurement error and so all models have the same expected loss, although will have different forecasts.

The first block computed the series to be forecast.

[3]:
```
import statsmodels.api as sm
from numpy.random import randn

t = 1000
factors = randn(t, 3)
beta = np.array([1, 0.5, 0.1])
e = randn(t)
y = factors.dot(beta)
```

The next block computes the benchmark factors and the model factors by contaminating the original factors with noise. The models are estimated on the first 500 observations and predictions are made for the second 500. Finally, losses are constructed from these predictions.

[4]:
```
# Measurement noise
bm_factors = factors + randn(t, 3)
# Fit using first half, predict second half
bm_beta = sm.OLS(y[:500], bm_factors[:500]).fit().params
# MSE loss
bm_losses = (y[500:] - bm_factors[500:].dot(bm_beta)) ** 2.0
# Number of models
k = 500
model_factors = np.zeros((k, t, 3))
model_losses = np.zeros((500, k))
for i in range(k):
    # Add measurement noise
    model_factors[i] = factors + randn(1000, 3)
    # Compute regression parameters
    model_beta = sm.OLS(y[:500], model_factors[i, :500]).fit().params
    # Prediction and losses
    model_losses[:, i] = (y[500:] - model_factors[i, 500:].dot(model_beta)) ** 2.0
```

Finally the SPA can be used. The SPA requires the **losses** from the benchmark and the models as inputs. Other inputs allow the bootstrap sued to be changed or for various options regarding studentization of the losses. `compute` does the real work, and then `pvalues` contains the probability that the null is true given the realizations.

In this case, one would not reject. The three p-values correspond to different re-centerings of the losses. In general, the `consistent` p-value should be used. It should always be the case that

$$lower \leq consistent \leq upper.$$

See the original papers for more details.

[5]:
```
from arch.bootstrap import SPA

spa = SPA(bm_losses, model_losses, seed=seed)
spa.compute()
spa.pvalues
```

[5]:
```
lower         0.005
consistent    0.005
upper         0.005
dtype: float64
```

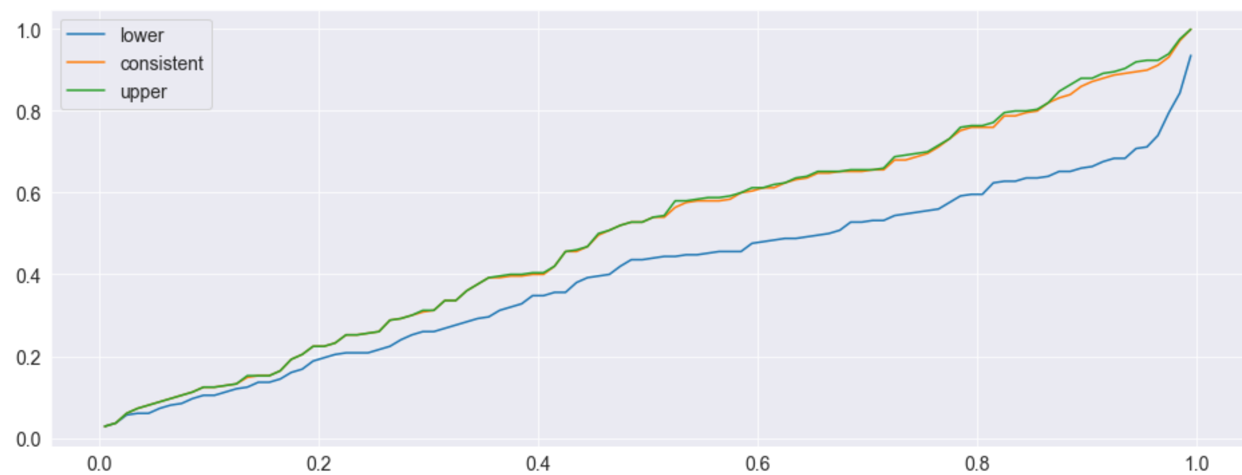
The same blocks can be repeated to perform a simulation study. Here I only use 100 replications since this should complete in a reasonable amount of time. Also I set `reps=250` to limit the number of bootstrap replications in each application of the SPA (the default is a more reasonable 1000).

[6]:
```
# Save the pvalues
pvalues = []
b = 100
seeds = gen.integers(0, 2**31 - 1, b)
# Repeat 100 times
for j in range(b):
    if j % 10 == 0:
        print(j)
    factors = randn(t, 3)
    beta = np.array([1, 0.5, 0.1])
    e = randn(t)
    y = factors.dot(beta)

    # Measurement noise
    bm_factors = factors + randn(t, 3)
    # Fit using first half, predict second half
    bm_beta = sm.OLS(y[:500], bm_factors[:500]).fit().params
    # MSE loss
    bm_losses = (y[500:] - bm_factors[500:].dot(bm_beta)) ** 2.0
    # Number of models
    k = 500
    model_factors = np.zeros((k, t, 3))
    model_losses = np.zeros((500, k))
    for i in range(k):
        model_factors[i] = factors + randn(1000, 3)
        model_beta = sm.OLS(y[:500], model_factors[i, :500]).fit().params
        # MSE loss
        model_losses[:, i] = (y[500:] - model_factors[i, 500:].dot(model_beta)) ** 2.0
    # Lower the bootstrap replications to 250
    spa = SPA(bm_losses, model_losses, reps=250, seed=seeds[j])
    spa.compute()
    pvalues.append(spa.pvalues)
```

```
0
10
20
30
40
50
60
70
80
90
```

Finally the pvalues can be plotted. Ideally they should form a 45^o line indicating the size is correct. Both the consistent and upper perform well. The lower has too many small p-values.

[7]:
```
import pandas as pd

pvalues = pd.DataFrame(pvalues)
for col in pvalues:
    values = pvalues[col].values
    values.sort()
    pvalues[col] = values
# Change the index so that the x-values are between 0 and 1
pvalues.index = np.linspace(0.005, 0.995, 100)
fig = pvalues.plot()
```



Power

The SPA also has power to reject then the null is violated. The simulation will be modified so that the amount of measurement error differs across models, and so that some models are actually better than the benchmark. The p-values should be small indicating rejection of the null.

[8]:
```
# Number of models
k = 500
model_factors = np.zeros((k, t, 3))
model_losses = np.zeros((500, k))
for i in range(k):
```

(continues on next page)

(continued from previous page)

```
    scale = (2500.0 - i) / 2500.0
    model_factors[i] = factors + scale * randn(1000, 3)
    model_beta = sm.OLS(y[:500], model_factors[i, :500]).fit().params
    # MSE loss
    model_losses[:, i] = (y[500:] - model_factors[i, 500:].dot(model_beta)) ** 2.0

spa = SPA(bm_losses, model_losses, seed=seed)
spa.compute()
spa.pvalues
```

[8]:

```
lower         0.039
consistent    0.049
upper         0.050
dtype: float64
```

Here the average losses are plotted. The higher index models are clearly better than the lower index models – and the benchmark model (which is identical to model.0).

[9]:

```
model_losses = pd.DataFrame(model_losses, columns=["model." + str(i) for i in range(k)])
avg_model_losses = pd.DataFrame(model_losses.mean(0), columns=["Average loss"])
fig = avg_model_losses.plot(style=["o"])
```

3.1.2 Stepwise Multiple Testing (StepM)

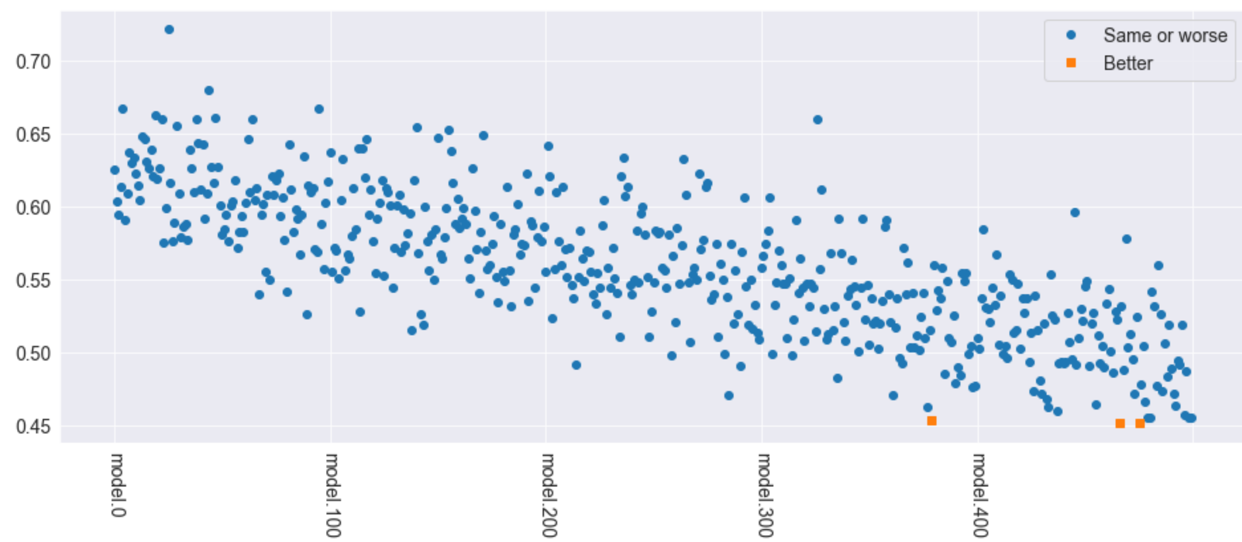
Stepwise Multiple Testing is similar to the SPA and has the same null. The primary difference is that it identifies the set of models which are better than the benchmark, rather than just asking the basic question if any model is better.

[10]:

```
from arch.bootstrap import StepM

stepm = StepM(bm_losses, model_losses)
stepm.compute()
print("Model indices:")
print([model.split(".")[1] for model in stepm.superior_models])
```

```
Model indices:
['379', '466', '475']
```

[11]:
```
better_models = pd.concat([model_losses.mean(0), model_losses.mean(0)], 1)
better_models.columns = ["Same or worse", "Better"]
better = better_models.index.isin(stepm.superior_models)
worse = np.logical_not(better)
better_models.loc[better, "Same or worse"] = np.nan
better_models.loc[worse, "Better"] = np.nan
fig = better_models.plot(style=["o", "s"], rot=270)
```



3.1.3 The Model Confidence Set

The model confidence set takes a set of **losses** as its input and finds the set which are not statistically different from each other while controlling the familywise error rate. The primary output is a set of p-values, where models with a pvalue above the size are in the MCS. Small p-values indicate that the model is easily rejected from the set that includes the best.

[12]:
```
from arch.bootstrap import MCS

# Limit the size of the set
losses = model_losses.iloc[:, ::20]
mcs = MCS(losses, size=0.10)
mcs.compute()
print("MCS P-values")
print(mcs.pvalues)
print("Included")
included = mcs.included
print([model.split(".")[1] for model in included])
print("Excluded")
excluded = mcs.excluded
print([model.split(".")[1] for model in excluded])
```

```
MCS P-values
            Pvalue
Model name
model.20     0.001
```

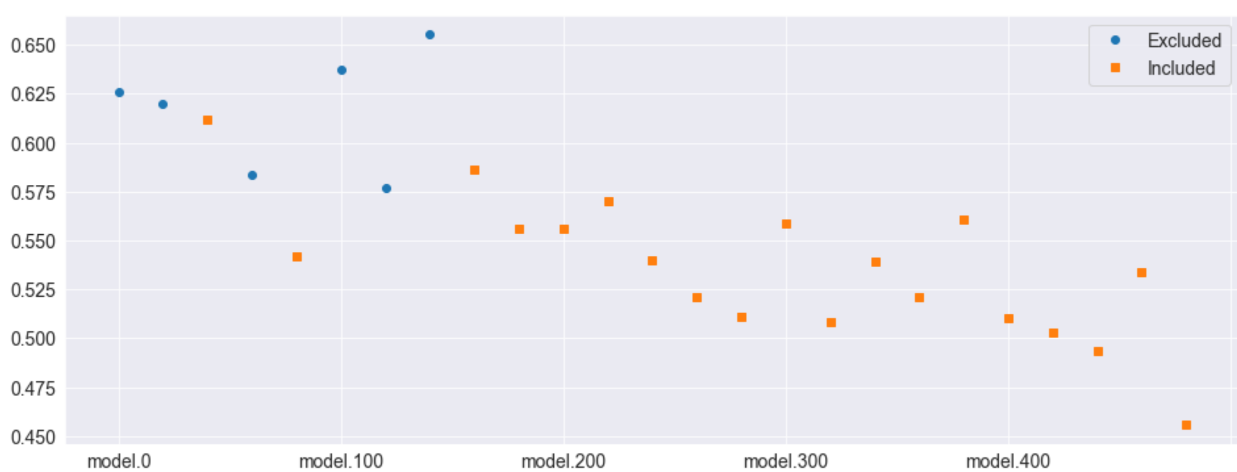
(continues on next page)

(continued from previous page)

```
model.0      0.003
model.120    0.005
model.100    0.005
model.140    0.011
model.60     0.074
model.40     0.101
model.160    0.118
model.380    0.216
model.300    0.287
model.80     0.443
model.180    0.443
model.220    0.443
model.460    0.506
model.340    0.536
model.200    0.619
model.240    0.740
model.360    0.840
model.400    0.840
model.320    0.840
model.260    0.840
model.420    0.840
model.280    0.840
model.440    0.840
model.480    1.000
Included
['160', '180', '200', '220', '240', '260', '280', '300', '320', '340', '360', '380', '40
↪', '400', '420', '440', '460', '480', '80']
Excluded
['0', '100', '120', '140', '20', '60']
```

[13]:

```
status = pd.DataFrame(
    [losses.mean(0), losses.mean(0)], index=["Excluded", "Included"]
).T
status.loc[status.index.isin(included), "Excluded"] = np.nan
status.loc[status.index.isin(excluded), "Included"] = np.nan
fig = status.plot(style=["o", "s"])
```



3.2 Module Reference

3.2.1 Test of Superior Predictive Ability (SPA), Reality Check

The test of Superior Predictive Ability (Hansen 2005), or SPA, is an improved version of the Reality Check (White 2000). It tests whether the best forecasting performance from a set of models is better than that of the forecasts from a benchmark model. A model is "better" if its losses are smaller than those from the benchmark. Formally, it tests the null

$$H_0 : \max_i E[L_i] \geq E[L_{bm}]$$

where L_i is the loss from model i and L_{bm} is the loss from the benchmark model. The alternative is

$$H_1 : \min_i E[L_i] < E[L_{bm}]$$

This procedure accounts for dependence between the losses and the fact that there are potentially alternative models being considered.

Note: Also callable using `RealityCheck`

SPA(benchmark, models[, block_size, reps, ...])	Test of Superior Predictive Ability (SPA) of White and Hansen.

arch.bootstrap.SPA

class arch.bootstrap.SPA(benchmark: *ArrayLike*, models: *ArrayLike*, block_size: *int | None* = `None`, reps: *int* = `1000`, bootstrap: *'stationary' | 'sb' | 'circular' | 'cbb' | 'moving block' | 'mbb'* = `'stationary'`, studentize: *bool* = `True`, nested: *bool* = `False`, *, seed: *None | int | np.random.Generator | np.random.RandomState* = `None`)

Test of Superior Predictive Ability (SPA) of White and Hansen.

The SPA is also known as the Reality Check or Bootstrap Data Snooper.

Parameters

benchmark: ArrayLike
T element array of benchmark model *losses*

models: ArrayLike
T by k element array of alternative model *losses*

block_size: int | None = None
Length of window to use in the bootstrap. If not provided, sqrt(T) is used. In general, this should be provided and chosen to be appropriate for the data.

reps: int = 1000
Number of bootstrap replications to uses. Default is 1000.

bootstrap: 'stationary' | 'sb' | 'circular' | 'cbb' | 'moving block' | 'mbb' = 'stationary'
Bootstrap to use. Options are 'stationary' or 'sb': Stationary bootstrap (Default) 'circular' or 'cbb': Circular block bootstrap 'moving block' or 'mbb': Moving block bootstrap

studentize: bool = True
Flag indicating to studentize loss differentials. Default is True

nested: bool = False
Flag indicating to use a nested bootstrap to compute variances for studentization. Default is False. Note that this can be slow since the procedure requires k extra bootstraps.

seed: None | int | np.random.Generator | np.random.RandomState = None
Seed value to use when creating the bootstrap used in the comparison. If an integer or None, the NumPy default_rng is used with the seed value. If a Generator or a RandomState, the argument is used.

Notes

The three p-value correspond to different re-centering decisions.

- Upper : Never recenter to all models are relevant to distribution
- Consistent : Only recenter if closer than a log(log(t)) bound
- Lower : Never recenter a model if worse than benchmark

See[1] and[2] for details.

See also

StepM

References

Methods

better_models([pvalue, pvalue_type])	Returns set of models rejected as being equal-or-worse than the benchmark
compute()	Compute the bootstrap pvalue.
critical_values([pvalue])	Returns data-dependent critical values
reset()	Reset the bootstrap to its initial state.
seed(value)	Seed the bootstrap's random number generator
subset(selector)	Sets a list of active models to run the SPA on.

arch.bootstrap.SPA.better_models

SPA.**better_models**(pvalue: *float* = 0.05, pvalue_type: *'lower'* | *'consistent'* | *'upper'* = 'consistent') → ndarray | list[Hashable]

Returns set of models rejected as being equal-or-worse than the benchmark

Parameters

pvalue: float = 0.05
P-value in (0,1) to use when computing superior models

pvalue_type: 'lower' | 'consistent' | 'upper' = 'consistent'
String in ‘lower’, ‘consistent’, or ‘upper’ indicating which critical value to use.

[1] Hansen, P. R. (2005). A test for superior predictive ability. Journal of Business & Economic Statistics, 23(4), 365-380.
[2] White, H. (2000). A reality check for data snooping. Econometrica, 68(5), 1097-1126.

Returns
indices – List of column names or indices of the superior models. Column names are returned if models is a DataFrame.

Return type
`list`

Notes

List of superior models returned is always with respect to the initial set of models, even when using subset().

arch.bootstrap.SPA.compute

`SPA.compute()` → None

Compute the bootstrap pvalue.

Notes

Must be called before accessing the pvalue.

arch.bootstrap.SPA.critical_values

`SPA.critical_values(pvalue: float = 0.05)` → pandas.Series

Returns data-dependent critical values

Parameters

pvalue: float = 0.05
P-value in (0,1) to use when computing the critical values.

Returns
crit_vals – Series containing critical values for the lower, consistent and upper methodologies

Return type
`pandas.Series`

arch.bootstrap.SPA.reset

`SPA.reset()` → None

Reset the bootstrap to its initial state.

arch.bootstrap.SPA.seed

`SPA.seed(value: int | list[int] | ndarray)` → None

Seed the bootstrap's random number generator

Parameters

value: int | list[int] | ndarray
Integer to use as the seed

arch.bootstrap.SPA.subset

`SPA.subset(selector: ndarray) → None`

Sets a list of active models to run the SPA on. Primarily for internal use.

Parameters

selector: ndarray
Boolean array indicating which columns to use when computing the p-values. This is primarily for use by StepM.

Properties

pvalues	P-values corresponding to the lower, consistent and upper p-values.

arch.bootstrap.SPA.pvalues

`property SPA.pvalues : pandas.Series`

P-values corresponding to the lower, consistent and upper p-values.

Returns
pvals – Three p-values corresponding to the lower bound, the consistent estimator, and the upper bound.

Return type
`pandas.Series`

3.2.2 Stepwise Multiple Testing (StepM)

The Stepwise Multiple Testing procedure (Romano & Wolf (2005)) is closely related to the SPA, except that it returns a set of models that are superior to the benchmark model, rather than the p-value from the null. They are so closely related that *StepM* is essentially a wrapper around *SPA* with some small modifications to allow multiple calls.

StepM(benchmark, models[, size, block_size, ...])	StepM multiple comparison procedure of Romano and Wolf.

arch.bootstrap.StepM

`class arch.bootstrap.StepM(benchmark: ArrayLike, models: ArrayLike, size: float = 0.05, block_size: int | None = None, reps: int = 1000, bootstrap: 'stationary' | 'sb' | 'circular' | 'cbb' | 'moving block' | 'mbb' = 'stationary', studentize: bool = True, nested: bool = False, *, seed: None | int | np.random.Generator | np.random.RandomState = None)`

StepM multiple comparison procedure of Romano and Wolf.

Parameters

benchmark: ArrayLike
T element array of benchmark model *losses*

models: ArrayLike
T by k element array of alternative model *losses*

size: float = 0.05
Value in (0,1) to use as the test size when implementing the comparison. Default value is 0.05.

block_size: int | None = None
Length of window to use in the bootstrap. If not provided, sqrt(T) is used. In general, this should be provided and chosen to be appropriate for the data.

reps: int = 1000
Number of bootstrap replications to uses. Default is 1000.

bootstrap: 'stationary' | 'sb' | 'circular' | 'cbb' | 'moving block' | 'mbb' = 'stationary'
Bootstrap to use. Options are 'stationary' or 'sb': Stationary bootstrap (Default) 'circular' or 'cbb': Circular block bootstrap 'moving block' or 'mbb': Moving block bootstrap

studentize: bool = True
Flag indicating to studentize loss differentials. Default is True

nested: bool = False
Flag indicating to use a nested bootstrap to compute variances for studentization. Default is False. Note that this can be slow since the procedure requires k extra bootstraps.

seed: None | int | **np.random.Generator | np.random.RandomState** = None
Seed value to use when creating the bootstrap used in the comparison. If an integer or None, the NumPy default_rng is used with the seed value. If a Generator or a RandomState, the argument is used.

Notes

The size controls the Family Wise Error Rate (FWER) since this is a multiple comparison procedure. Uses SPA and the consistent selection procedure.

See[1] for detail.

See also

SPA

References

Methods

compute()	Compute the set of superior models.
reset()	Reset the bootstrap to it's initial state.
seed(value)	Seed the bootstrap's random number generator

[1] Romano, J. P., & Wolf, M. (2005). Stepwise multiple testing as formalized data snooping. Econometrica, 73(4), 1237-1282.

arch.bootstrap.StepM.compute

`StepM.compute()` → None

Compute the set of superior models.

arch.bootstrap.StepM.reset

`StepM.reset()` → None

Reset the bootstrap to it's initial state.

arch.bootstrap.StepM.seed

`StepM.seed(value: int | list[int] | ndarray)` → None

Seed the bootstrap's random number generator

Parameters

value: int | list[int] | ndarray
Integer to use as the seed

Properties

superior_models	List of the indices or column names of the superior models

arch.bootstrap.StepM.superior_models

property `StepM.superior_models` : list[Hashable]

List of the indices or column names of the superior models

Returns
List of superior models. Contains column indices if models is an array or contains column names if models is a DataFrame.

Return type
`list`

3.2.3 Model Confidence Set (MCS)

The Model Confidence Set (Hansen, Lunde & Nason (2011)) differs from other multiple comparison procedures in that there is no benchmark. The MCS attempts to identify the set of models which produce the same expected loss, while controlling the probability that a model that is worse than the best model is in the model confidence set. Like the other MCPs, it controls the Familywise Error Rate rather than the usual test size.

MCS(losses, size[, reps, block_size, ...])	Model Confidence Set (MCS) of Hansen, Lunde and Nason.

arch.bootstrap.MCS

class arch.bootstrap.**MCS**(losses: *ArrayLike2D*, size: *float*, reps: *int* = 1000, block_size: *int* | *None* = None, method: *'R'* | *'max'* = 'R', bootstrap: *'stationary'* | *'sb'* | *'circular'* | *'cbb'* | *'moving block'* | *'mbb'* = 'stationary', *, seed: *None* | *int* | *np.random.Generator* | *np.random.RandomState* = None)

Model Confidence Set (MCS) of Hansen, Lunde and Nason.

Parameters

losses: ArrayLike2D
T by k array containing losses from a set of models

size: float
Value in (0,1) to use as the test size when implementing the mcs. Default value is 0.05.

block_size: int | None = None
Length of window to use in the bootstrap. If not provided, sqrt(T) is used. In general, this should be provided and chosen to be appropriate for the data.

method: 'R' | 'max' = 'R'
MCS test and elimination implementation method, either 'max' or 'R'. Default is 'R'.

reps: int = 1000
Number of bootstrap replications to uses. Default is 1000.

bootstrap: 'stationary' | 'sb' | 'circular' | 'cbb' | 'moving block' | 'mbb' = 'stationary'
Bootstrap to use. Options are 'stationary' or 'sb': Stationary bootstrap (Default) 'circular' or 'cbb': Circular block bootstrap 'moving block' or 'mbb': Moving block bootstrap

seed: None | int | np.random.Generator | np.random.RandomState = None
Seed value to use when creating the bootstrap used in the comparison. If an integer or None, the NumPy default_rng is used with the seed value. If a Generator or a RandomState, the argument is used.

Notes

See[1] for details.

References

Methods

compute()	Compute the set of models in the confidence set.
reset()	Reset the bootstrap to it's initial state.
seed(value)	Seed the bootstrap's random number generator

[1] Hansen, P. R., Lunde, A., & Nason, J. M. (2011). The model confidence set. Econometrica, 79(2), 453-497.

arch.bootstrap.MCS.compute

`MCS.compute()` → None

Compute the set of models in the confidence set.

arch.bootstrap.MCS.reset

`MCS.reset()` → None

Reset the bootstrap to it's initial state.

arch.bootstrap.MCS.seed

`MCS.seed(value: int | list[int] | ndarray)` → None

Seed the bootstrap's random number generator

Parameters

value: int | list[int] | ndarray
Integer to use as the seed

Properties

excluded	List of model indices that are excluded from the MCS
included	List of model indices that are included in the MCS
pvalues	Model p-values for inclusion in the MCS

arch.bootstrap.MCS.excluded

property `MCS.excluded` : list[Hashable]

List of model indices that are excluded from the MCS

Returns
excluded – List of column indices or names of the excluded models

Return type
`list`

arch.bootstrap.MCS.included

property `MCS.included` : list[Hashable]

List of model indices that are included in the MCS

Returns
included – List of column indices or names of the included models

Return type
`list`

arch.bootstrap.MCS.pvalues

property `MCS.pvalues` : pd.DataFrame

Model p-values for inclusion in the MCS

Returns
pvalues – DataFrame where the index is the model index (column or name) containing the smallest size where the model is in the MCS.

Return type
`pandas.DataFrame`

3.3 References

Articles used in the creation of this module include

CHAPTER

FOUR

UNIT ROOT TESTING

Many time series are highly persistent, and determining whether the data appear to be stationary or contains a unit root is the first step in many analyses. This module contains a number of routines:

- Augmented Dickey-Fuller (`ADF`)
- Dickey-Fuller GLS (`DFGLS`)
- Phillips-Perron (`PhillipsPerron`)
- KPSS (`KPSS`)
- Zivot-Andrews (`ZivotAndrews`)
- Variance Ratio (`VarianceRatio`)
- Automatic Bandwidth Selection (`auto_bandwidth()`)

The first four all start with the null of a unit root and have an alternative of a stationary process. The final test, KPSS, has a null of a stationary process with an alternative of a unit root.

4.1 Introduction

All tests expect a 1-d series as the first input. The input can be any array that can *squeeze* into a 1-d array, a pandas *Series* or a pandas *DataFrame* that contains a single variable.

All tests share a common structure. The key elements are:

- *stat* - Returns the test statistic
- *pvalue* - Returns the p-value of the test statistic
- *lags* - Sets or gets the number of lags used in the model. In most test, can be `None` to trigger automatic selection.
- *trend* - Sets or gets the trend used in the model. Supported trends vary by model, but include:
  - *'nc'*: No constant
  - *'c'*: Constant
  - *'ct'*: Constant and time trend
  - *'ctt'*: Constant, time trend and quadratic time trend
- *summary()* - Returns a summary object that can be printed to get a formatted table

4.1.1 Basic Example

This basic example show the use of the Augmented-Dickey fuller to test whether the default premium, defined as the difference between the yields of large portfolios of BAA and AAA bonds. This example uses a constant and time trend.

```
import datetime as dt

import pandas_datareader.data as web
from arch.unitroot import ADF

start = dt.datetime(1919, 1, 1)
end = dt.datetime(2014, 1, 1)

df = web.DataReader(["AAA", "BAA"], "fred", start, end)
df['diff'] = df['BAA'] - df['AAA']
adf = ADF(df['diff'])
adf.trend = 'ct'

print(adf.summary())
```

which yields

```
   Augmented Dickey-Fuller Results
=====================================
Test Statistic                 -3.448
P-value                         0.045
Lags                               21
-------------------------------------

Trend: Constant and Linear Time Trend
Critical Values: -3.97 (1%), -3.41 (5%), -3.13 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

4.2 Unit Root Testing

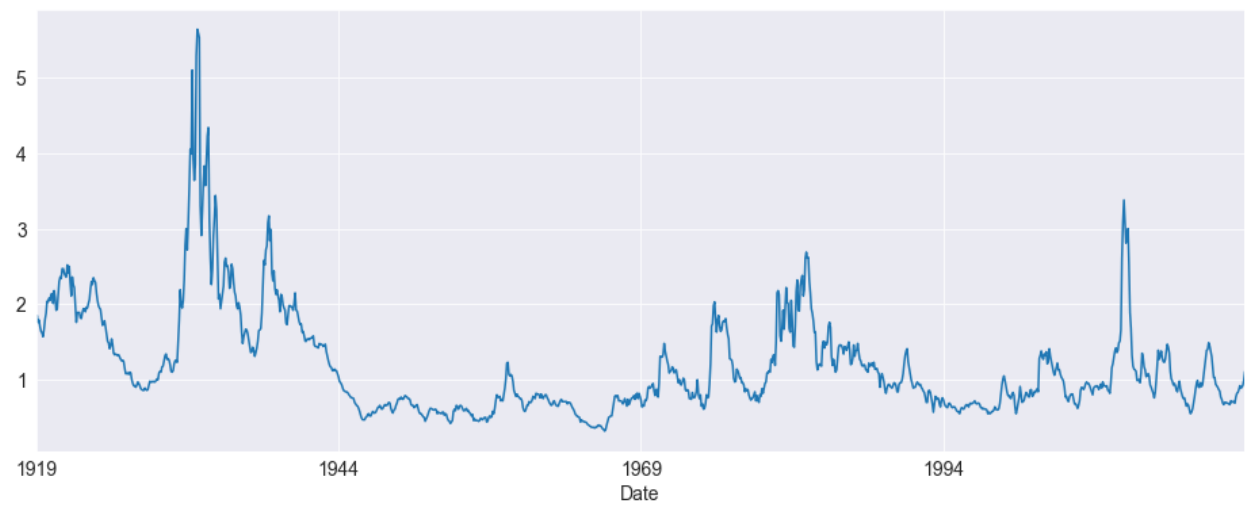
This setup code is required to run in an IPython notebook

[1]:
```
import warnings

warnings.simplefilter("ignore")

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn

seaborn.set_style("darkgrid")
plt.rc("figure", figsize=(16, 6))
plt.rc("savefig", dpi=90)
plt.rc("font", family="sans-serif")
plt.rc("font", size=14)
```

4.2.1 Setup

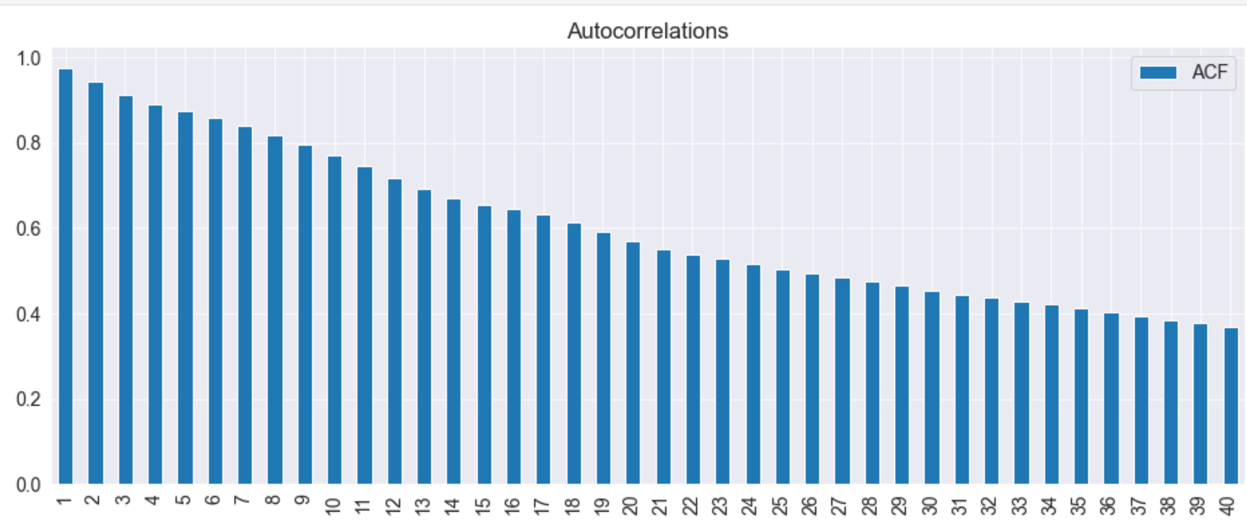
Most examples will make use of the Default premium, which is the difference between the yields of BAA and AAA rated corporate bonds. The data is downloaded from FRED using pandas.

[2]:
```
import arch.data.default
import pandas as pd
import statsmodels.api as sm

default_data = arch.data.default.load()
default = default_data.BAA.copy()
default.name = "default"
default = default - default_data.AAA.values
fig = default.plot()
```



The Default premium is clearly highly persistent. A simple check of the autocorrelations confirms this.

[3]:
```
acf = pd.DataFrame(sm.tsa.stattools.acf(default), columns=["ACF"])
fig = acf[1:].plot(kind="bar", title="Autocorrelations")
```



4.2.2 Augmented Dickey-Fuller Testing

The Augmented Dickey-Fuller test is the most common unit root test used. It is a regression of the first difference of the variable on its lagged level as well as additional lags of the first difference. The null is that the series contains a unit root, and the (one-sided) alternative is that the series is stationary.

By default, the number of lags is selected by minimizing the AIC across a range of lag lengths (which can be set using `max_lag` when initializing the model). Additionally, the basic test includes a constant in the ADF regression.

These results indicate that the Default premium is stationary.

[4]:
```
from arch.unitroot import ADF

adf = ADF(default)
print(adf.summary().as_text())
```

```
   Augmented Dickey-Fuller Results
=====================================
Test Statistic                 -3.356
P-value                         0.013
Lags                               21
-------------------------------------

Trend: Constant
Critical Values: -3.44 (1%), -2.86 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

The number of lags can be directly set using `lags`. Changing the number of lags makes no difference to the conclusion.

Note: The ADF assumes residuals are white noise, and that the number of lags is sufficient to pick up any dependence in the data.

Setting the number of lags

[5]:
```
adf = ADF(default, lags=5)
print(adf.summary().as_text())
```

```
   Augmented Dickey-Fuller Results
=====================================
Test Statistic                 -3.582
P-value                         0.006
Lags                                5
-------------------------------------

Trend: Constant
Critical Values: -3.44 (1%), -2.86 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

Deterministic terms

The deterministic terms can be altered using `trend`. The options are:

- `'nc'` : No deterministic terms
- `'c'` : Constant only
- `'ct'` : Constant and time trend
- `'ctt'` : Constant, time trend and time-trend squared

Changing the type of constant also makes no difference for this data.

[6]:
```
adf = ADF(default, trend="ct", lags=5)
print(adf.summary().as_text())
```

```
   Augmented Dickey-Fuller Results
=====================================
Test Statistic                 -3.786
P-value                         0.017
Lags                                5
-------------------------------------

Trend: Constant and Linear Time Trend
Critical Values: -3.97 (1%), -3.41 (5%), -3.13 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

Regression output

The ADF uses a standard regression when computing results. These can be accesses using `regression`.

[7]:
```
reg_res = adf.regression
print(reg_res.summary().as_text())
```

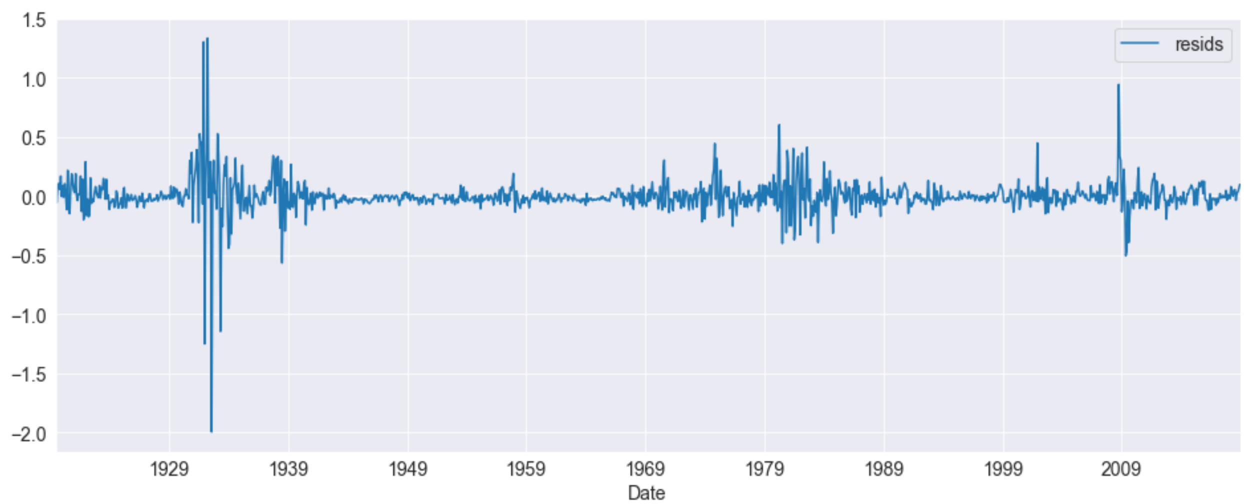
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.095
Model:                            OLS   Adj. R-squared:                  0.090
Method:                 Least Squares   F-statistic:                     17.83
Date:                Tue, 18 May 2021   Prob (F-statistic):           1.30e-22
Time:                        13:22:02   Log-Likelihood:                 630.15
No. Observations:                1194   AIC:                            -1244.
Df Residuals:                    1186   BIC:                            -1204.
Df Model:                           7
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Level.L1      -0.0248      0.007     -3.786      0.000      -0.038      -0.012
Diff.L1        0.2229      0.029      7.669      0.000       0.166       0.280
Diff.L2       -0.0525      0.030     -1.769      0.077      -0.111       0.006
Diff.L3       -0.1363      0.029     -4.642      0.000      -0.194      -0.079
Diff.L4       -0.0510      0.030     -1.727      0.084      -0.109       0.007
Diff.L5        0.0440      0.029      1.516      0.130      -0.013       0.101
```

(continues on next page)

(continued from previous page)

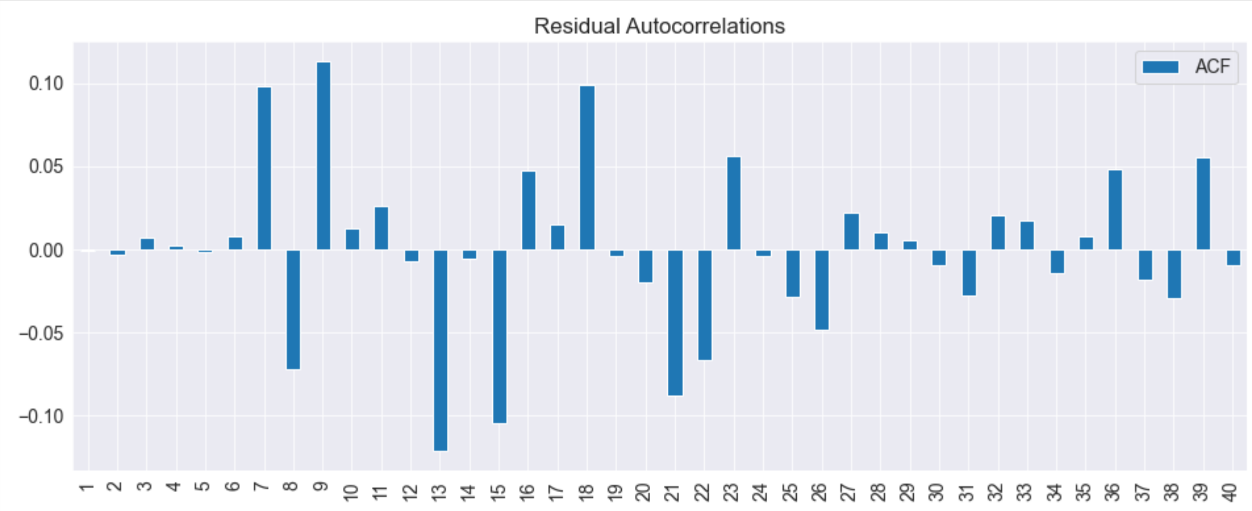
```
const          0.0383      0.013      2.858      0.004      0.012      0.065
trend      -1.586e-05   1.29e-05     -1.230      0.219   -4.11e-05   9.43e-06
==============================================================================
Omnibus:                      665.553   Durbin-Watson:                   2.000
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           146083.295
Skew:                          -1.425   Prob(JB):                         0.00
Kurtosis:                      57.113   Cond. No.                     5.70e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
[2] The condition number is large, 5.7e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

[8]:
```
import matplotlib.pyplot as plt
import pandas as pd

resids = pd.DataFrame(reg_res.resid)
resids.index = default.index[6:]
resids.columns = ["resids"]
fig = resids.plot()
```



Since the number lags was directly set, it is good to check whether the residuals appear to be white noise.

[9]:
```
acf = pd.DataFrame(sm.tsa.stattools.acf(reg_res.resid), columns=["ACF"])
fig = acf[1:].plot(kind="bar", title="Residual Autocorrelations")
```



4.2.3 Dickey-Fuller GLS Testing

The Dickey-Fuller GLS test is an improved version of the ADF which uses a GLS-detrending regression before running an ADF regression with no additional deterministic terms. This test is only available with a constant or constant and time trend (`trend='c'` or `trend='ct'`).

The results of this test agree with the ADF results.

[10]:
```
from arch.unitroot import DFGLS

dfgls = DFGLS(default)
print(dfgls.summary().as_text())
```

```
      Dickey-Fuller GLS Results
=====================================
Test Statistic                 -2.322
P-value                         0.020
Lags                               21
-------------------------------------

Trend: Constant
Critical Values: -2.59 (1%), -1.96 (5%), -1.64 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

The trend can be altered using `trend`. The conclusion is the same.

[11]:
```
dfgls = DFGLS(default, trend="ct")
print(dfgls.summary().as_text())
```

```
      Dickey-Fuller GLS Results
=====================================
Test Statistic                 -3.464
P-value                         0.009
Lags                               21
-------------------------------------
```

(continues on next page)

(continued from previous page)

```
Trend: Constant and Linear Time Trend
Critical Values: -3.43 (1%), -2.86 (5%), -2.58 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

4.2.4 Phillips-Perron Testing

The Phillips-Perron test is similar to the ADF except that the regression run does not include lagged values of the first differences. Instead, the PP test fixed the t-statistic using a long run variance estimation, implemented using a Newey-West covariance estimator.

By default, the number of lags is automatically set, although this can be overridden using `lags`.

[12]:
```
from arch.unitroot import PhillipsPerron

pp = PhillipsPerron(default)
print(pp.summary().as_text())
```

```
     Phillips-Perron Test (Z-tau)
=====================================
Test Statistic                 -3.898
P-value                         0.002
Lags                               23
-------------------------------------

Trend: Constant
Critical Values: -3.44 (1%), -2.86 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

It is important that the number of lags is sufficient to pick up any dependence in the data.

[13]:
```
pp = PhillipsPerron(default, lags=12)
print(pp.summary().as_text())
```

```
     Phillips-Perron Test (Z-tau)
=====================================
Test Statistic                 -4.024
P-value                         0.001
Lags                               12
-------------------------------------

Trend: Constant
Critical Values: -3.44 (1%), -2.86 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

The trend can be changed as well.

[14]:
```
pp = PhillipsPerron(default, trend="ct", lags=12)
print(pp.summary().as_text())
```

```
     Phillips-Perron Test (Z-tau)
=====================================
Test Statistic                 -4.262
P-value                         0.004
Lags                               12
-------------------------------------

Trend: Constant and Linear Time Trend
Critical Values: -3.97 (1%), -3.41 (5%), -3.13 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

Finally, the PP testing framework includes two types of tests. One which uses an ADF-type regression of the first difference on the level, the other which regresses the level on the level. The default is the `tau` test, which is similar to an ADF regression, although this can be changed using `test_type='rho'`.

[15]:
```
pp = PhillipsPerron(default, test_type="rho", trend="ct", lags=12)
print(pp.summary().as_text())
```

```
     Phillips-Perron Test (Z-rho)
=====================================
Test Statistic                -36.114
P-value                         0.002
Lags                               12
-------------------------------------

Trend: Constant and Linear Time Trend
Critical Values: -29.16 (1%), -21.60 (5%), -18.17 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

4.2.5 KPSS Testing

The KPSS test differs from the three previous in that the null is a stationary process and the alternative is a unit root.

Note that here the null is rejected which indicates that the series might be a unit root.

[16]:
```
from arch.unitroot import KPSS

kpss = KPSS(default)
print(kpss.summary().as_text())
```

```
    KPSS Stationarity Test Results
=====================================
Test Statistic                  1.088
P-value                         0.002
Lags                               20
-------------------------------------

Trend: Constant
Critical Values: 0.74 (1%), 0.46 (5%), 0.35 (10%)
Null Hypothesis: The process is weakly stationary.
Alternative Hypothesis: The process contains a unit root.
```

Changing the trend does not alter the conclusion.

[17]:
```
kpss = KPSS(default, trend="ct")
print(kpss.summary().as_text())
```

```
    KPSS Stationarity Test Results
=====================================
Test Statistic                  0.393
P-value                         0.000
Lags                               20
-------------------------------------

Trend: Constant and Linear Time Trend
Critical Values: 0.22 (1%), 0.15 (5%), 0.12 (10%)
Null Hypothesis: The process is weakly stationary.
Alternative Hypothesis: The process contains a unit root.
```

4.2.6 Zivot-Andrews Test

The Zivot-Andrews test allows the possibility of a single structural break in the series. Here we test the default using the test.

[18]:
```
from arch.unitroot import ZivotAndrews

za = ZivotAndrews(default)
print(za.summary().as_text())
```

```
        Zivot-Andrews Results
=====================================
Test Statistic                 -4.900
P-value                         0.040
Lags                               21
-------------------------------------

Trend: Constant
Critical Values: -5.28 (1%), -4.81 (5%), -4.57 (10%)
Null Hypothesis: The process contains a unit root with a single structural break.
Alternative Hypothesis: The process is trend and break stationary.
```

4.2.7 Variance Ratio Testing

Variance ratio tests are not usually used as unit root tests, and are instead used for testing whether a financial return series is a pure random walk versus having some predictability. This example uses the excess return on the market from Ken French's data.

[19]:
```
import arch.data.frenchdata
import numpy as np
import pandas as pd

ff = arch.data.frenchdata.load()
excess_market = ff.iloc[:, 0]  # Excess Market
print(ff.describe())
```

```
           Mkt-RF          SMB          HML           RF
count  1109.000000  1109.000000  1109.000000  1109.000000
mean      0.659946     0.206555     0.368864     0.274220
std       5.327524     3.191132     3.482352     0.253377
min     -29.130000   -16.870000   -13.280000    -0.060000
25%      -1.970000    -1.560000    -1.320000     0.030000
50%       1.020000     0.070000     0.140000     0.230000
75%       3.610000     1.730000     1.740000     0.430000
max      38.850000    36.700000    35.460000     1.350000
```

The variance ratio compares the variance of a 1-period return to that of a multi-period return. The comparison length has to be set when initializing the test.

This example compares 1-month to 12-month returns, and the null that the series is a pure random walk is rejected. Negative values indicate some positive autocorrelation in the returns (momentum).

[20]:
```
from arch.unitroot import VarianceRatio

vr = VarianceRatio(excess_market, 12)
print(vr.summary().as_text())
```

```
     Variance-Ratio Test Results
=====================================
Test Statistic                 -5.029
P-value                         0.000
Lags                               12
-------------------------------------

Computed with overlapping blocks (de-biased)
```

By default the VR test uses all overlapping blocks to estimate the variance of the long period's return. This can be changed by setting `overlap=False`. This lowers the power but does not change the conclusion.

[21]:
```
warnings.simplefilter("always")  # Restore warnings

vr = VarianceRatio(excess_market, 12, overlap=False)
print(vr.summary().as_text())
```

```
     Variance-Ratio Test Results
=====================================
Test Statistic                 -6.206
P-value                         0.000
Lags                               12
-------------------------------------

Computed with non-overlapping blocks
```

```
c:\git\arch\arch\unitroot\unitroot.py:1679: InvalidLengthWarning:
The length of y is not an exact multiple of 12, and so the final
4 observations have been dropped.

  warnings.warn(
```

Note: The warning is intentional. It appears here since when it is not possible to use all data since the data length is not an integer multiple of the long period when using non-overlapping blocks. There is little reason to use `overlap=False`.

4.3 The Unit Root Tests

ADF(y[, lags, trend, max_lags, method, ...])	Augmented Dickey-Fuller unit root test
DFGLS(y[, lags, trend, max_lags, method, ...])	Elliott, Rothenberg and Stock's ([1]_) GLS detrended Dickey-Fuller
PhillipsPerron(y[, lags, trend, test_type])	Phillips-Perron unit root test
ZivotAndrews(y[, lags, trend, trim, ...])	Zivot-Andrews structural-break unit-root test
VarianceRatio(y[, lags, trend, debiased, ...])	Variance Ratio test of a random walk.
KPSS(y[, lags, trend])	Kwiatkowski, Phillips, Schmidt and Shin (KPSS) stationarity test

4.3.1 arch.unitroot.ADF

class arch.unitroot.**ADF**(y: *ndarray* | *DataFrame* | *Series*, lags: *int* | *None* = **None**, trend: *'n'* | *'c'* | *'ct'* | *'ctt'* = *'c'*, max_lags: *int* | *None* = **None**, method: *'aic'* | *'bic'* | *'t-stat'* = *'aic'*, low_memory: *bool* | *None* = **None**)

Augmented Dickey-Fuller unit root test

Parameters

y: ndarray | DataFrame | Series
The data to test for a unit root

lags: int | None = None
The number of lags to use in the ADF regression. If omitted or None, *method* is used to automatically select the lag length with no more than *max_lags* are included.

trend: 'n' | 'c' | 'ct' | 'ctt' = 'c'

The trend component to include in the test

- ”n” - No trend components
- ”c” - Include a constant (Default)
- ”ct” - Include a constant and linear time trend
- ”ctt” - Include a constant and linear and quadratic time trends

max_lags: int | None = None
The maximum number of lags to use when selecting lag length

method: 'aic' | 'bic' | 't-stat' = 'aic'

The method to use when selecting the lag length

- ”AIC” - Select the minimum of the Akaike IC
- ”BIC” - Select the minimum of the Schwarz/Bayesian IC
- ”t-stat” - Select the minimum of the Schwarz/Bayesian IC

low_memory: bool | None = None
Flag indicating whether to use a low memory implementation of the lag selection algorithm. The low memory algorithm is slower than the standard algorithm but will use 2-4% of the memory required for the standard algorithm. This options allows automatic lag selection to be used in very long time series. If None, use automatic selection of algorithm.

Notes

The null hypothesis of the Augmented Dickey-Fuller is that there is a unit root, with the alternative that there is no unit root. If the pvalue is above a critical size, then the null cannot be rejected that there and the series appears to be a unit root.

The p-values are obtained through regression surface approximation from MacKinnon (1994) using the updated 2010 tables. If the p-value is close to significant, then the critical values should be used to judge whether to reject the null.

The autolag option and maxlag for it are described in Greene.

Examples

```
>>> from arch.unitroot import ADF
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.macrodata.load().data
>>> inflation = np.diff(np.log(data["cpi"]))
>>> adf = ADF(inflation)
>>> print("{0:0.4f}".format(adf.stat))
-3.0931
>>> print("{0:0.4f}".format(adf.pvalue))
0.0271
>>> adf.lags
2
>>> adf.trend="ct"
>>> print("{0:0.4f}".format(adf.stat))
-3.2111
>>> print("{0:0.4f}".format(adf.pvalue))
0.0822
```

References

Methods

summary()	Summary of test, containing statistic, p-value and critical values

arch.unitroot.ADF.summary

`ADF.summary()` → Summary

Summary of test, containing statistic, p-value and critical values

Properties

`alternative_hypothesis`	The alternative hypothesis
`critical_values`	Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.
`lags`	Sets or gets the number of lags used in the model.
`max_lags`	Sets or gets the maximum lags used when automatically selecting lag length
`nobs`	The number of observations used when computing the test statistic.
`null_hypothesis`	The null hypothesis
`pvalue`	Returns the p-value for the test statistic
`regression`	Returns the OLS regression results from the ADF model estimated
`stat`	The test statistic for a unit root
`trend`	Sets or gets the deterministic trend term used in the test.
`valid_trends`	List of valid trend terms.
`y`	Returns the data used in the test statistic

arch.unitroot.ADF.alternative_hypothesis

property ADF.alternative_hypothesis : str

The alternative hypothesis

arch.unitroot.ADF.critical_values

property ADF.critical_values : dict[str, float]

Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

arch.unitroot.ADF.lags

property ADF.lags : int

Sets or gets the number of lags used in the model. When bootstrap use DF-type regressions, lags is the number of lags in the regression model. When bootstrap use long-run variance estimators, lags is the number of lags used in the long-run variance estimator.

arch.unitroot.ADF.max_lags

property ADF.max_lags : int | None

Sets or gets the maximum lags used when automatically selecting lag length

arch.unitroot.ADF.nobs

property ADF.**nobs** : int

The number of observations used when computing the test statistic. Accounts for loss of data due to lags for regression-based bootstrap.

arch.unitroot.ADF.null_hypothesis

property ADF.**null_hypothesis** : str

The null hypothesis

arch.unitroot.ADF.pvalue

property ADF.**pvalue** : float

Returns the p-value for the test statistic

arch.unitroot.ADF.regression

property ADF.**regression** : RegressionResults

Returns the OLS regression results from the ADF model estimated

arch.unitroot.ADF.stat

property ADF.**stat** : float

The test statistic for a unit root

arch.unitroot.ADF.trend

property ADF.**trend** : str

Sets or gets the deterministic trend term used in the test. See valid_trends for a list of supported trends

arch.unitroot.ADF.valid_trends

property ADF.**valid_trends** : list[str]

List of valid trend terms.

arch.unitroot.ADF.y

property ADF.**y** : ndarray | DataFrame | Series

Returns the data used in the test statistic

4.3.2 arch.unitroot.DFGLS

class arch.unitroot.**DFGLS**(*y: ndarray | DataFrame | Series*, *lags: int | None = None*, *trend: 'c' | 'ct' = 'c'*, *max_lags: int | None = None*, *method: 'aic' | 'bic' | 't-stat' = 'aic'*, *low_memory: bool | None = None*)

Elliott, Rothenberg and Stock's ([1]) GLS detrended Dickey-Fuller

Parameters

y: ndarray | DataFrame | Series
The data to test for a unit root

lags: int | None = None
The number of lags to use in the ADF regression. If omitted or None, *method* is used to automatically select the lag length with no more than *max_lags* are included.

trend: 'c' | 'ct' = 'c'

The trend component to include in the test

- "c" - Include a constant (Default)
- "ct" - Include a constant and linear time trend

max_lags: int | None = None
The maximum number of lags to use when selecting lag length. When using automatic lag length selection, the lag is selected using OLS detrending rather than GLS detrending ([2]).

method: 'aic' | 'bic' | 't-stat' = 'aic'

The method to use when selecting the lag length

- "AIC" - Select the minimum of the Akaike IC
- "BIC" - Select the minimum of the Schwarz/Bayesian IC
- "t-stat" - Select the minimum of the Schwarz/Bayesian IC

Notes

The null hypothesis of the Dickey-Fuller GLS is that there is a unit root, with the alternative that there is no unit root. If the pvalue is above a critical size, then the null cannot be rejected and the series appears to be a unit root.

DFGLS differs from the ADF test in that an initial GLS detrending step is used before a trend-less ADF regression is run.

Critical values and p-values when trend is "c" are identical to the ADF. When trend is set to "ct", they are from …

[1] Elliott, G. R., T. J. Rothenberg, and J. H. Stock. 1996. Efficient bootstrap for an autoregressive unit root. Econometrica 64: 813-836

[2] Perron, P., & Qu, Z. (2007). A simple modification to improve the finite sample properties of Ng and Perron's unit root tests. Economics letters, 94(1), 12-19.

Examples

```
>>> from arch.unitroot import DFGLS
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.macrodata.load().data
>>> inflation = np.diff(np.log(data["cpi"]))
>>> dfgls = DFGLS(inflation)
>>> print("{0:0.4f}".format(dfgls.stat))
-2.7611
>>> print("{0:0.4f}".format(dfgls.pvalue))
0.0059
>>> dfgls.lags
2
>>> dfgls.trend = "ct"
>>> print("{0:0.4f}".format(dfgls.stat))
-2.9036
>>> print("{0:0.4f}".format(dfgls.pvalue))
0.0447
```

References

Methods

summary()	Summary of test, containing statistic, p-value and critical values

arch.unitroot.DFGLS.summary

DFGLS.**summary**() → Summary

Summary of test, containing statistic, p-value and critical values

Properties

alternative_hypothesis	The alternative hypothesis
critical_values	Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.
lags	Sets or gets the number of lags used in the model.
max_lags	Sets or gets the maximum lags used when automatically selecting lag length
nobs	The number of observations used when computing the test statistic.
null_hypothesis	The null hypothesis
pvalue	Returns the p-value for the test statistic
regression	Returns the OLS regression results from the ADF model estimated
stat	The test statistic for a unit root
trend	Sets or gets the deterministic trend term used in the test.
valid_trends	List of valid trend terms.
y	Returns the data used in the test statistic

arch.unitroot.DFGLS.alternative_hypothesis

property DFGLS.**alternative_hypothesis** : str

The alternative hypothesis

arch.unitroot.DFGLS.critical_values

property DFGLS.**critical_values** : dict[str, float]

Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

arch.unitroot.DFGLS.lags

property DFGLS.**lags** : int

Sets or gets the number of lags used in the model. When bootstrap use DF-type regressions, lags is the number of lags in the regression model. When bootstrap use long-run variance estimators, lags is the number of lags used in the long-run variance estimator.

arch.unitroot.DFGLS.max_lags

property DFGLS.**max_lags** : int | None

Sets or gets the maximum lags used when automatically selecting lag length

arch.unitroot.DFGLS.nobs

property `DFGLS.nobs` : int

The number of observations used when computing the test statistic. Accounts for loss of data due to lags for regression-based bootstrap.

arch.unitroot.DFGLS.null_hypothesis

property `DFGLS.null_hypothesis` : str

The null hypothesis

arch.unitroot.DFGLS.pvalue

property `DFGLS.pvalue` : float

Returns the p-value for the test statistic

arch.unitroot.DFGLS.regression

property `DFGLS.regression` : RegressionResults

Returns the OLS regression results from the ADF model estimated

arch.unitroot.DFGLS.stat

property `DFGLS.stat` : float

The test statistic for a unit root

arch.unitroot.DFGLS.trend

property `DFGLS.trend`

Sets or gets the deterministic trend term used in the test. See valid_trends for a list of supported trends

arch.unitroot.DFGLS.valid_trends

property `DFGLS.valid_trends` : list[str]

List of valid trend terms.

arch.unitroot.DFGLS.y

property `DFGLS.y` : ndarray | DataFrame | Series

Returns the data used in the test statistic

4.3.3 arch.unitroot.PhillipsPerron

class arch.unitroot.**PhillipsPerron**(*y: ndarray | DataFrame | Series*, *lags: int | None = None*, *trend: 'n' | 'c' | 'ct' = 'c'*, *test_type: 'tau' | 'rho' = 'tau'*)

Phillips-Perron unit root test

Parameters

y: ndarray | DataFrame | Series
The data to test for a unit root

lags: int | None = None
The number of lags to use in the Newey-West estimator of the long-run covariance. If omitted or None, the lag length is set automatically to 12 * (nobs/100) ** (1/4)

trend: 'n' | 'c' | 'ct' = 'c'

The trend component to include in the test

- ”n” - No trend components
- ”c” - Include a constant (Default)
- ”ct” - Include a constant and linear time trend

test_type: 'tau' | 'rho' = 'tau'
The test to use when computing the test statistic. “tau” is based on the t-stat and “rho” uses a test based on nobs times the re-centered regression coefficient

Notes

The null hypothesis of the Phillips-Perron (PP) test is that there is a unit root, with the alternative that there is no unit root. If the pvalue is above a critical size, then the null cannot be rejected that there and the series appears to be a unit root.

Unlike the ADF test, the regression estimated includes only one lag of the dependant variable, in addition to trend terms. Any serial correlation in the regression errors is accounted for using a long-run variance estimator (currently Newey-West).

The p-values are obtained through regression surface approximation from MacKinnon (1994) using the updated 2010 tables. If the p-value is close to significant, then the critical values should be used to judge whether to reject the null.

Examples

```
>>> from arch.unitroot import PhillipsPerron
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.macrodata.load().data
>>> inflation = np.diff(np.log(data["cpi"]))
>>> pp = PhillipsPerron(inflation)
>>> print("{0:0.4f}".format(pp.stat))
-8.1356
>>> print("{0:0.4f}".format(pp.pvalue))
0.0000
>>> pp.lags
```

(continues on next page)

(continued from previous page)

```
15
>>> pp.trend = "ct"
>>> print("{0:0.4f}".format(pp.stat))
-8.2022
>>> print("{0:0.4f}".format(pp.pvalue))
0.0000
>>> pp.test_type = "rho"
>>> print("{0:0.4f}".format(pp.stat))
-120.3271
>>> print("{0:0.4f}".format(pp.pvalue))
0.0000
```

References

Methods

summary()	Summary of test, containing statistic, p-value and critical values

arch.unitroot.PhillipsPerron.summary

PhillipsPerron.**summary**() → Summary

Summary of test, containing statistic, p-value and critical values

Properties

alternative_hypothesis	The alternative hypothesis
critical_values	Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.
lags	Sets or gets the number of lags used in the model.
nobs	The number of observations used when computing the test statistic.
null_hypothesis	The null hypothesis
pvalue	Returns the p-value for the test statistic
regression	Returns OLS regression results for the specification used in the test
stat	The test statistic for a unit root
test_type	Gets or sets the test type returned by stat.
trend	Sets or gets the deterministic trend term used in the test.
valid_trends	List of valid trend terms.
y	Returns the data used in the test statistic

arch.unitroot.PhillipsPerron.alternative_hypothesis

property PhillipsPerron.**alternative_hypothesis** : str

The alternative hypothesis

arch.unitroot.PhillipsPerron.critical_values

property PhillipsPerron.**critical_values** : dict[str, float]

Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

arch.unitroot.PhillipsPerron.lags

property PhillipsPerron.**lags** : int

Sets or gets the number of lags used in the model. When bootstrap use DF-type regressions, lags is the number of lags in the regression model. When bootstrap use long-run variance estimators, lags is the number of lags used in the long-run variance estimator.

arch.unitroot.PhillipsPerron.nobs

property PhillipsPerron.**nobs** : int

The number of observations used when computing the test statistic. Accounts for loss of data due to lags for regression-based bootstrap.

arch.unitroot.PhillipsPerron.null_hypothesis

property PhillipsPerron.**null_hypothesis** : str

The null hypothesis

arch.unitroot.PhillipsPerron.pvalue

property PhillipsPerron.**pvalue** : float

Returns the p-value for the test statistic

arch.unitroot.PhillipsPerron.regression

property PhillipsPerron.**regression** : RegressionResults

Returns OLS regression results for the specification used in the test

The results returned use a Newey-West covariance matrix with the same number of lags as are used in the test statistic.

arch.unitroot.PhillipsPerron.stat

property `PhillipsPerron.stat` : float

The test statistic for a unit root

arch.unitroot.PhillipsPerron.test_type

property `PhillipsPerron.test_type` : str

Gets or sets the test type returned by stat. Valid values are “tau” or “rho”

arch.unitroot.PhillipsPerron.trend

property `PhillipsPerron.trend` : str

Sets or gets the deterministic trend term used in the test. See valid_trends for a list of supported trends

arch.unitroot.PhillipsPerron.valid_trends

property `PhillipsPerron.valid_trends` : list[str]

List of valid trend terms.

arch.unitroot.PhillipsPerron.y

property `PhillipsPerron.y` : ndarray | DataFrame | Series

Returns the data used in the test statistic

4.3.4 arch.unitroot.ZivotAndrews

class `arch.unitroot.ZivotAndrews`(y: *ndarray* | *DataFrame* | *Series*, lags: *int* | *None* = **None**, trend: *'c'* | *'ct'* | *'t'* = `'c'`, trim: *float* = **0.15**, max_lags: *int* | *None* = **None**, method: *'aic'* | *'bic'* | *'t-stat'* = `'aic'`)

Zivot-Andrews structural-break unit-root test

The Zivot-Andrews test can be used to test for a unit root in a univariate process in the presence of serial correlation and a single structural break.

Parameters

y: ndarray | DataFrame | Series
data series

lags: int | None = None
The number of lags to use in the ADF regression. If omitted or None, *method* is used to automatically select the lag length with no more than *max_lags* are included.

trend: 'c' | 'ct' | 't' = 'c'

The trend component to include in the test

- ”c” - Include a constant (Default)
- ”t” - Include a linear time trend
- ”ct” - Include a constant and linear time trend

trim: float = 0.15
percentage of series at begin/end to exclude from break-period calculation in range [0, 0.333] (default=0.15)

max_lags: int | None = None
The maximum number of lags to use when selecting lag length

method: 'aic' | 'bic' | 't-stat' = 'aic'

The method to use when selecting the lag length

- "AIC" - Select the minimum of the Akaike IC
- "BIC" - Select the minimum of the Schwarz/Bayesian IC
- "t-stat" - Select the minimum of the Schwarz/Bayesian IC

Notes

H0 = unit root with a single structural break

Algorithm follows Baum (2004/2015) approximation to original Zivot-Andrews method. Rather than performing an autolag regression at each candidate break period (as per the original paper), a single autolag regression is run up-front on the base model (constant + trend with no dummies) to determine the best lag length. This lag length is then used for all subsequent break-period regressions. This results in significant run time reduction but also slightly more pessimistic test statistics than the original Zivot-Andrews method,

No attempt has been made to characterize the size/power trade-off.

References

Methods

summary()	Summary of test, containing statistic, p-value and critical values

arch.unitroot.ZivotAndrews.summary

`ZivotAndrews.summary()` → Summary
Summary of test, containing statistic, p-value and critical values

Properties

alternative_hypothesis	The alternative hypothesis
critical_values	Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.
lags	Sets or gets the number of lags used in the model.
nobs	The number of observations used when computing the test statistic.
null_hypothesis	The null hypothesis
pvalue	Returns the p-value for the test statistic
stat	The test statistic for a unit root
trend	Sets or gets the deterministic trend term used in the test.
valid_trends	List of valid trend terms.
y	Returns the data used in the test statistic

arch.unitroot.ZivotAndrews.alternative_hypothesis

property ZivotAndrews.**alternative_hypothesis** : str

The alternative hypothesis

arch.unitroot.ZivotAndrews.critical_values

property ZivotAndrews.**critical_values** : dict[str, float]

Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

arch.unitroot.ZivotAndrews.lags

property ZivotAndrews.**lags** : int

Sets or gets the number of lags used in the model. When bootstrap use DF-type regressions, lags is the number of lags in the regression model. When bootstrap use long-run variance estimators, lags is the number of lags used in the long-run variance estimator.

arch.unitroot.ZivotAndrews.nobs

property ZivotAndrews.**nobs** : int

The number of observations used when computing the test statistic. Accounts for loss of data due to lags for regression-based bootstrap.

arch.unitroot.ZivotAndrews.null_hypothesis

property ZivotAndrews.**null_hypothesis** : str

The null hypothesis

arch.unitroot.ZivotAndrews.pvalue

property ZivotAndrews.**pvalue** : float

Returns the p-value for the test statistic

arch.unitroot.ZivotAndrews.stat

property ZivotAndrews.**stat** : float

The test statistic for a unit root

arch.unitroot.ZivotAndrews.trend

property ZivotAndrews.**trend** : str

Sets or gets the deterministic trend term used in the test. See valid_trends for a list of supported trends

arch.unitroot.ZivotAndrews.valid_trends

property ZivotAndrews.**valid_trends** : list[str]

List of valid trend terms.

arch.unitroot.ZivotAndrews.y

property ZivotAndrews.**y** : ndarray | DataFrame | Series

Returns the data used in the test statistic

4.3.5 arch.unitroot.VarianceRatio

class arch.unitroot.**VarianceRatio**(y: *ndarray* | *DataFrame* | *Series*, lags: *int* = ***2***, trend: *'n'* | *'c'* = `'c'`, debiased: *bool* = ***True***, robust: *bool* = ***True***, overlap: *bool* = ***True***)

Variance Ratio test of a random walk.

Parameters

y: ndarray | DataFrame | Series
The data to test for a random walk

lags: int = 2
The number of periods to used in the multi-period variance, which is the numerator of the test statistic. Must be at least 2

trend: 'n' | 'c' = 'c'
"c" allows for a non-zero drift in the random walk, while "n" requires that the increments to y are mean 0

overlap: bool = True
Indicates whether to use all overlapping blocks. Default is True. If False, the number of observations in y minus 1 must be an exact multiple of lags. If this condition is not satisfied, some values at the end of y will be discarded.

robust: bool = True
Indicates whether to use heteroskedasticity robust inference. Default is True.

debiased: bool = True
Indicates whether to use a debiased version of the test. Default is True. Only applicable if overlap is True.

Notes

The null hypothesis of a VR is that the process is a random walk, possibly plus drift. Rejection of the null with a positive test statistic indicates the presence of positive serial correlation in the time series.

Examples

```
>>> from arch.unitroot import VarianceRatio
>>> import pandas_datareader as pdr
>>> data = pdr.get_data_fred("DJIA", start="2010-1-1", end="2020-12-31")
>>> data = np.log(data.resample("M").last())  # End of month
>>> vr = VarianceRatio(data, lags=12)
>>> print("{0:0.4f}".format(vr.pvalue))
0.1370
```

References

Methods

summary()	Summary of test, containing statistic, p-value and critical values

arch.unitroot.VarianceRatio.summary

VarianceRatio.**summary**() → Summary
Summary of test, containing statistic, p-value and critical values

Properties

alternative_hypothesis	The alternative hypothesis
critical_values	Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.
debiased	Sets of gets the indicator to use debiased variances in the ratio
lags	Sets or gets the number of lags used in the model.
nobs	The number of observations used when computing the test statistic.
null_hypothesis	The null hypothesis
overlap	Sets of gets the indicator to use overlapping returns in the long-period variance estimator
pvalue	Returns the p-value for the test statistic
robust	Sets of gets the indicator to use a heteroskedasticity robust variance estimator
stat	The test statistic for a unit root
trend	Sets or gets the deterministic trend term used in the test.
valid_trends	List of valid trend terms.
vr	The ratio of the long block lags-period variance to the 1-period variance
y	Returns the data used in the test statistic

arch.unitroot.VarianceRatio.alternative_hypothesis

property VarianceRatio.**alternative_hypothesis** : str

The alternative hypothesis

arch.unitroot.VarianceRatio.critical_values

property VarianceRatio.**critical_values** : dict[str, float]

Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

arch.unitroot.VarianceRatio.debiased

property VarianceRatio.**debiased** : bool

Sets of gets the indicator to use debiased variances in the ratio

arch.unitroot.VarianceRatio.lags

property VarianceRatio.**lags** : int

Sets or gets the number of lags used in the model. When bootstrap use DF-type regressions, lags is the number of lags in the regression model. When bootstrap use long-run variance estimators, lags is the number of lags used in the long-run variance estimator.

arch.unitroot.VarianceRatio.nobs

property VarianceRatio.**nobs** : int

The number of observations used when computing the test statistic. Accounts for loss of data due to lags for regression-based bootstrap.

arch.unitroot.VarianceRatio.null_hypothesis

property VarianceRatio.**null_hypothesis** : str

The null hypothesis

arch.unitroot.VarianceRatio.overlap

property VarianceRatio.**overlap** : bool

Sets of gets the indicator to use overlapping returns in the long-period variance estimator

arch.unitroot.VarianceRatio.pvalue

property VarianceRatio.**pvalue** : float

Returns the p-value for the test statistic

arch.unitroot.VarianceRatio.robust

property VarianceRatio.**robust** : bool

Sets of gets the indicator to use a heteroskedasticity robust variance estimator

arch.unitroot.VarianceRatio.stat

property VarianceRatio.**stat** : float

The test statistic for a unit root

arch.unitroot.VarianceRatio.trend

property VarianceRatio.**trend** : str

Sets or gets the deterministic trend term used in the test. See valid_trends for a list of supported trends

arch.unitroot.VarianceRatio.valid_trends

property `VarianceRatio.valid_trends` : list[str]

List of valid trend terms.

arch.unitroot.VarianceRatio.vr

property `VarianceRatio.vr` : float

The ratio of the long block lags-period variance to the 1-period variance

arch.unitroot.VarianceRatio.y

property `VarianceRatio.y` : ndarray | DataFrame | Series

Returns the data used in the test statistic

4.3.6 arch.unitroot.KPSS

class `arch.unitroot.KPSS`(y: *ndarray* | *DataFrame* | *Series*, lags: *int* | *None* = `None`, trend: *'c'* | *'ct'* = `'c'`)

Kwiatkowski, Phillips, Schmidt and Shin (KPSS) stationarity test

Parameters

y: ndarray | DataFrame | Series
The data to test for stationarity

lags: int | None = None
The number of lags to use in the Newey-West estimator of the long-run covariance. If omitted or None, the number of lags is calculated with the data-dependent method of Hobijn et al. (1998). See also Andrews (1991), Newey & West (1994), and Schwert (1989). Set lags=-1 to use the old method that only depends on the sample size, 12 * (nobs/100) ** (1/4).

trend: 'c' | 'ct' = 'c'

The trend component to include in the ADF test
”c” - Include a constant (Default) “ct” - Include a constant and linear time trend

Notes

The null hypothesis of the KPSS test is that the series is weakly stationary and the alternative is that it is non-stationary. If the p-value is above a critical size, then the null cannot be rejected that there and the series appears stationary.

The p-values and critical values were computed using an extensive simulation based on 100,000,000 replications using series with 2,000 observations.

Examples

```
>>> from arch.unitroot import KPSS
>>> import numpy as np
>>> import statsmodels.api as sm
>>> data = sm.datasets.macrodata.load().data
>>> inflation = np.diff(np.log(data["cpi"]))
>>> kpss = KPSS(inflation)
>>> print("{0:0.4f}".format(kpss.stat))
0.2870
>>> print("{0:0.4f}".format(kpss.pvalue))
0.1473
>>> kpss.trend = "ct"
>>> print("{0:0.4f}".format(kpss.stat))
0.2075
>>> print("{0:0.4f}".format(kpss.pvalue))
0.0128
```

References

Methods

summary()	Summary of test, containing statistic, p-value and critical values

arch.unitroot.KPSS.summary

KPSS.**summary**() → Summary

Summary of test, containing statistic, p-value and critical values

Properties

alternative_hypothesis	The alternative hypothesis
critical_values	Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.
lags	Sets or gets the number of lags used in the model.
nobs	The number of observations used when computing the test statistic.
null_hypothesis	The null hypothesis
pvalue	Returns the p-value for the test statistic
stat	The test statistic for a unit root
trend	Sets or gets the deterministic trend term used in the test.
valid_trends	List of valid trend terms.
y	Returns the data used in the test statistic

arch.unitroot.KPSS.alternative_hypothesis

property KPSS.**alternative_hypothesis** : str

The alternative hypothesis

arch.unitroot.KPSS.critical_values

property KPSS.**critical_values** : dict[str, float]

Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

arch.unitroot.KPSS.lags

property KPSS.**lags** : int

Sets or gets the number of lags used in the model. When bootstrap use DF-type regressions, lags is the number of lags in the regression model. When bootstrap use long-run variance estimators, lags is the number of lags used in the long-run variance estimator.

arch.unitroot.KPSS.nobs

property KPSS.**nobs** : int

The number of observations used when computing the test statistic. Accounts for loss of data due to lags for regression-based bootstrap.

arch.unitroot.KPSS.null_hypothesis

property KPSS.**null_hypothesis** : str

The null hypothesis

arch.unitroot.KPSS.pvalue

property KPSS.**pvalue** : float

Returns the p-value for the test statistic

arch.unitroot.KPSS.stat

property KPSS.**stat** : float

The test statistic for a unit root

arch.unitroot.KPSS.trend

property KPSS.**trend** : str

Sets or gets the deterministic trend term used in the test. See valid_trends for a list of supported trends

arch.unitroot.KPSS.valid_trends

property KPSS.**valid_trends** : list[str]

List of valid trend terms.

arch.unitroot.KPSS.y

property KPSS.**y** : ndarray | DataFrame | Series

Returns the data used in the test statistic

4.3.7 Automatic Bandwidth Selection

auto_bandwidth(y[, kernel])	Automatic bandwidth selection of Andrews (1991) and Newey & West (1994).

arch.unitroot.auto_bandwidth

arch.unitroot.**auto_bandwidth**(y: *Sequence[float | int] | ndarray | Series*, kernel: *'ba' | 'bartlett' | 'nw' | 'pa' | 'parzen' | 'gallant' | 'qs' | 'andrews' = 'ba'*) → float

Automatic bandwidth selection of Andrews (1991) and Newey & West (1994).

Parameters

y: Sequence[float | int] | ndarray | Series
Data on which to apply the bandwidth selection

kernel: 'ba' | 'bartlett' | 'nw' | 'pa' | 'parzen' | 'gallant' | 'qs' | 'andrews' = 'ba'

The kernel function to use for selecting the bandwidth

- ”ba”, “bartlett”, “nw”: Bartlett kernel (default)
- ”pa”, “parzen”, “gallant”: Parzen kernel
- ”qs”, “andrews”: Quadratic Spectral kernel

Returns

The estimated optimal bandwidth.

Return type

float

CHAPTER

FIVE

COINTEGRATION ANALYSIS

The module extended the single-series unit root testing to multiple series and cointegration testing and cointegrating vector estimation.

- Cointegrating Testing
  - Engle-Granger Test (*engle_granger*)
  - Phillips-Ouliaris Tests (*phillips_ouliaris*)
- Cointegrating Vector Estimation
  - Dynamic OLS (*DynamicOLS*)
  - Fully Modified OLS (*FullyModifiedOLS*)
  - Canonical Cointegrating Regression (*CanonicalCointegratingReg*)

5.1 Cointegration Testing

This setup code is required to run in an IPython notebook

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn

seaborn.set_style("darkgrid")
plt.rc("figure", figsize=(16, 6))
plt.rc("savefig", dpi=90)
plt.rc("font", family="sans-serif")
plt.rc("font", size=14)
```

We will look at the spot prices of crude oil measured in Cushing, OK for West Texas Intermediate Crude, and Brent Crude. The underlying data in this data set come from the U.S. Energy Information Administration.

[2]:
```
import numpy as np
from arch.data import crude

data = crude.load()
log_price = np.log(data)

ax = log_price.plot()
xl = ax.set_xlim(log_price.index.min(), log_price.index.max())
```

We can verify these both of these series appear to contains unit roots using Augmented Dickey-Fuller tests. The p-values are large indicating that the null cannot be rejected.

[3]:
```
from arch.unitroot import ADF

ADF(log_price.WTI, trend="c")
```

[3]:
```
<class 'arch.unitroot.unitroot.ADF'>
"""
   Augmented Dickey-Fuller Results
=====================================
Test Statistic                 -1.780
P-value                         0.391
Lags                                1
-------------------------------------

Trend: Constant
Critical Values: -3.45 (1%), -2.87 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
"""
```

[4]:
```
ADF(log_price.Brent, trend="c")
```

[4]:
```
<class 'arch.unitroot.unitroot.ADF'>
"""
   Augmented Dickey-Fuller Results
=====================================
Test Statistic                 -1.655
P-value                         0.454
Lags                                1
-------------------------------------

Trend: Constant
Critical Values: -3.45 (1%), -2.87 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

(continues on next page)

(continued from previous page)

```
"""
```

The Engle-Granger test is a 2-step test that first estimates a cross-sectional regression, and then tests the residuals from this regression using an Augmented Dickey-Fuller distribution with modified critical values. The cross-sectional regression is

$$Y_t = X_t\beta + D_t\gamma + \epsilon_t$$

where Y_t and X_t combine to contain the set of variables being tested for cointegration and D_t are a set of deterministic regressors that might include a constant, a time trend, or a quadratic time trend. The trend is specified using `trend` as

- `"n"`: No trend
- `"c"`: Constant
- `"ct"`: Constant and time trend
- `"ctt"`: Constant, time and quadratic trends

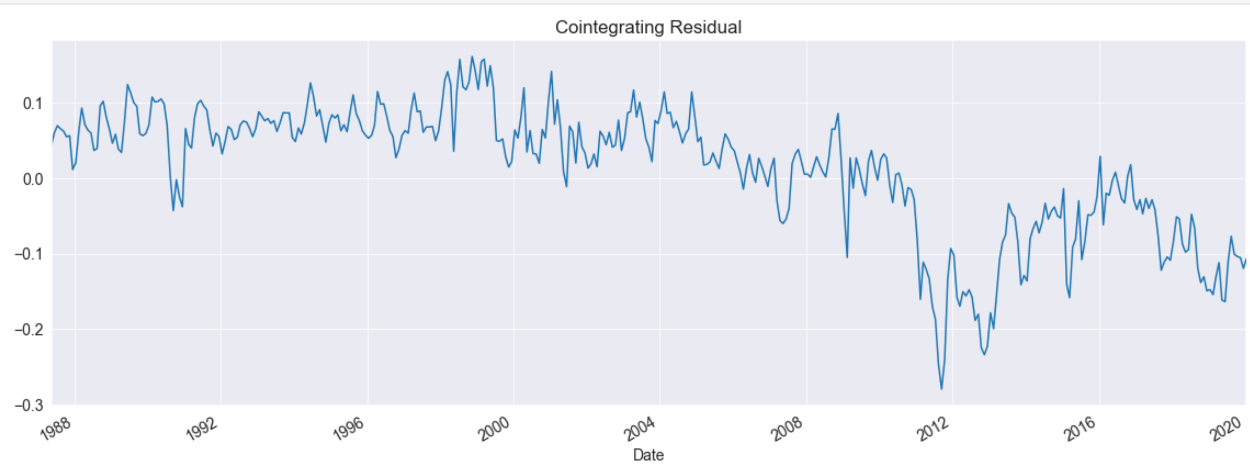
Here we assume that that cointegrating relationship is exact so that no deterministics are needed.

[5]:
```
from arch.unitroot import engle_granger

eg_test = engle_granger(log_price.WTI, log_price.Brent, trend="n")
eg_test
```

[5]:
```
Engle-Granger Cointegration Test
Statistic: -3.4676471998477267
P-value: 0.006860702109284017
Null: No Cointegration, Alternative: Cointegration
ADF Lag length: 0
Trend: c
Estimated Root  (+1): 0.9386946007157646
Distribution Order: 1
ID: 0x2352eeb6e50
```

The `plot` method can be used to plot the model residual. We see that while this appears to be mean 0, it might have a trend in it.

[6]:
```
fig = eg_test.plot()
```



The estimated cointegrating vector is exposed through he `cointegrating_vector` property. Here we see it is very close to $[1, -1]$, indicating a simple no-arbitrage relationship.

[7]:
```
eg_test.cointegrating_vector
```

[7]:
```
WTI       1.000000
Brent    -1.000621
dtype: float64
```

We can rerun the test with both a constant and a time trend to see how this affects the conclusion. We firmly reject the null of no cointegration even with this alternative assumption.

[8]:
```
eg_test = engle_granger(log_price.WTI, log_price.Brent, trend="ct")
eg_test
```

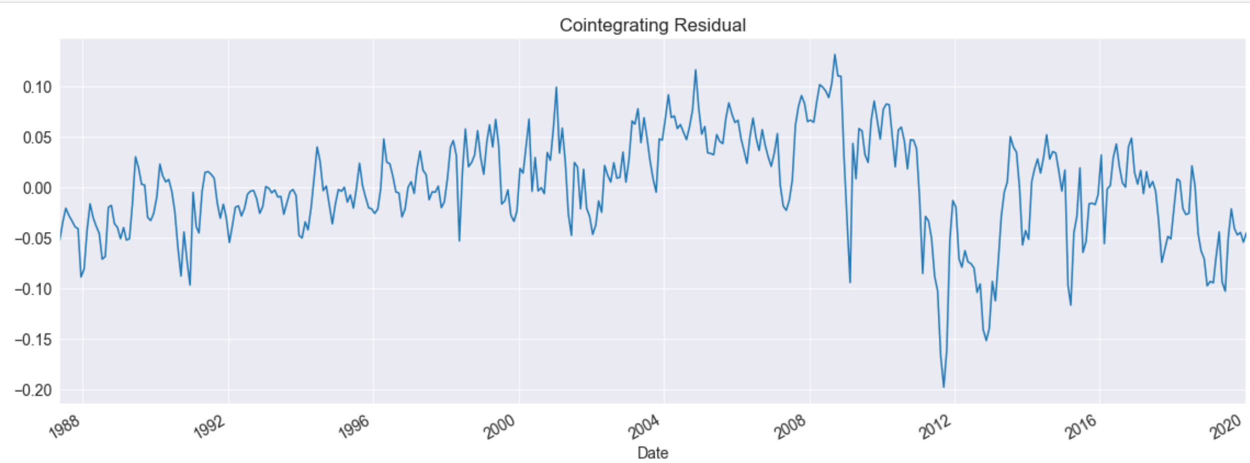
[8]:
```
Engle-Granger Cointegration Test
Statistic: -5.83664970914174
P-value: 2.3286206215070878e-05
Null: No Cointegration, Alternative: Cointegration
ADF Lag length: 0
Trend: c
Estimated Root  (+1): 0.8400729995315473
Distribution Order: 1
ID: 0x235285d7340
```

[9]:
```
eg_test.cointegrating_vector
```

[9]:
```
WTI       1.000000
Brent    -0.931769
const    -0.296939
trend     0.000185
dtype: float64
```

The residuals are clearly mean zero but show evidence of a structural break around the financial crisis of 2008.

[10]:
```
fig = eg_test.plot()
```



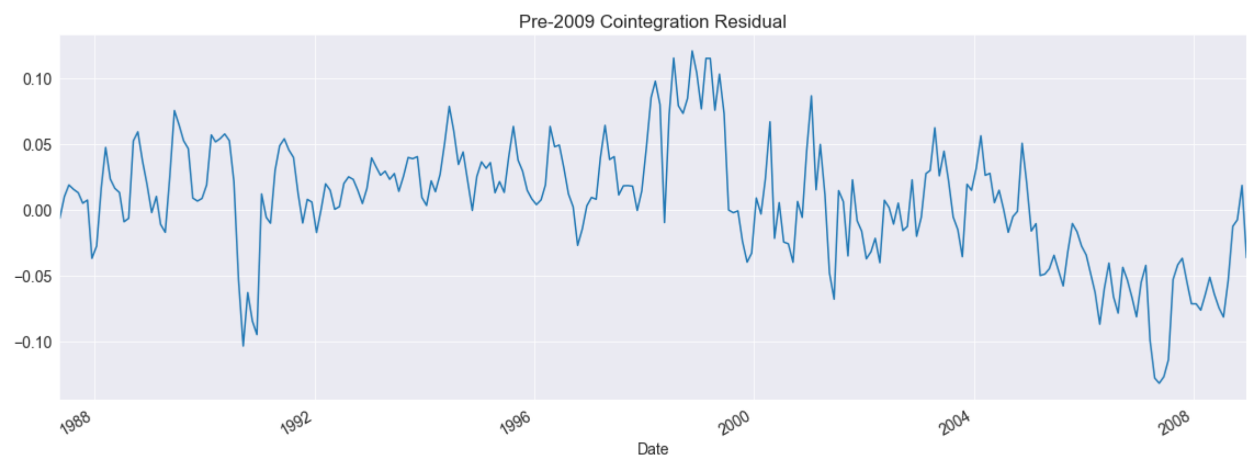
To investigate the changes in the 2008 financial crisis, we can re-run the test on only the pre-crisis period.

[11]:
```
eg_test = engle_granger(log_price[:"2008"].WTI, log_price[:"2008"].Brent, trend="n")
eg_test
```

[11]:
```
Engle-Granger Cointegration Test
Statistic: -4.962489476284803
P-value: 2.054007070920808e-05
Null: No Cointegration, Alternative: Cointegration
ADF Lag length: 0
Trend: c
Estimated Root  (+1): 0.8246009342909095
Distribution Order: 1
ID: 0x2352f312f70
```

These residuals look quite a bit better although it is possible the break in the cointegrating vector happened around 2005 when oil prices first surged.

[12]:
```
fig = eg_test.plot()
ax = fig.get_axes()[0]
title = ax.set_title("Pre-2009 Cointegration Residual")
```



5.1.1 Phillips-Ouliaris

The Phillips-Ouliaris tests consists four distinct tests. Two are similar to the Engle-Granger test, only using a Phillips & Perron-like approach replaces the lags in the ADF test with a long-run variance estimator. The other two use variance-ratio like approaches to test. In both cases the test stabilizes when there is no cointegration and diverges due to singularity of the covariance matrix of the I(1) time series when there is cointegration.

- Z_t - Like PP using the t-stat of the AR(1) coefficient in an AR(1) of the residual from the cross-sectional regression.
- Z_α - Like PP using $T(\alpha - 1)$ and a bias term from the same AR(1)
- P_u - A univariate variance ratio test.
- P_z - A multivariate variance ratio test.

The four test statistics all agree on the crude oil data.

The Z_t and Z_α test statistics are both based on the quantity $\gamma = \rho - 1$ from the regression $y_t = d_t\Delta + \rho y_{t-1} + \epsilon_t$. The null is rejected in favor of the alternative when $\gamma < 0$ so that the test statistic is *below* its critical value.

[13]:
```
from arch.unitroot.cointegration import phillips_ouliaris
```

(continues on next page)

(continued from previous page)

```
po_zt_test = phillips_ouliaris(
    log_price.WTI, log_price.Brent, trend="c", test_type="Zt"
)
po_zt_test.summary()
```

[13]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
Phillips-Ouliaris Zt Cointegration Test
=====================================
Test Statistic                 -5.357
P-value                         0.000
Kernel                       Bartlett
Bandwidth                      10.185
-------------------------------------

Trend: Constant
Critical Values: -3.06 (10%), -3.36 (5%), -3.93 (1%)
Null Hypothesis: No Cointegration
Alternative Hypothesis: Cointegration
Distribution Order: 3
"""
```

[14]:
```
po_za_test = phillips_ouliaris(
    log_price.WTI, log_price.Brent, trend="c", test_type="Za"
)
po_za_test.summary()
```

[14]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
Phillips-Ouliaris Za Cointegration Test
=====================================
Test Statistic                -53.531
P-value                         0.000
Kernel                       Bartlett
Bandwidth                      10.185
-------------------------------------

Trend: Constant
Critical Values: -16.95 (10%), -20.34 (5%), -27.76 (1%)
Null Hypothesis: No Cointegration
Alternative Hypothesis: Cointegration
Distribution Order: 3
"""
```

The P_u and P_z statistics are variance ratios where under the null the numerator and denominator are balanced and so converge at the same rate. Under the alternative the denominator converges to zero and the statistic diverges, so that rejection of the null occurs when the test statistic is *above* a critical value.

[15]:
```
po_pu_test = phillips_ouliaris(
    log_price.WTI, log_price.Brent, trend="c", test_type="Pu"
)
po_pu_test.summary()
```

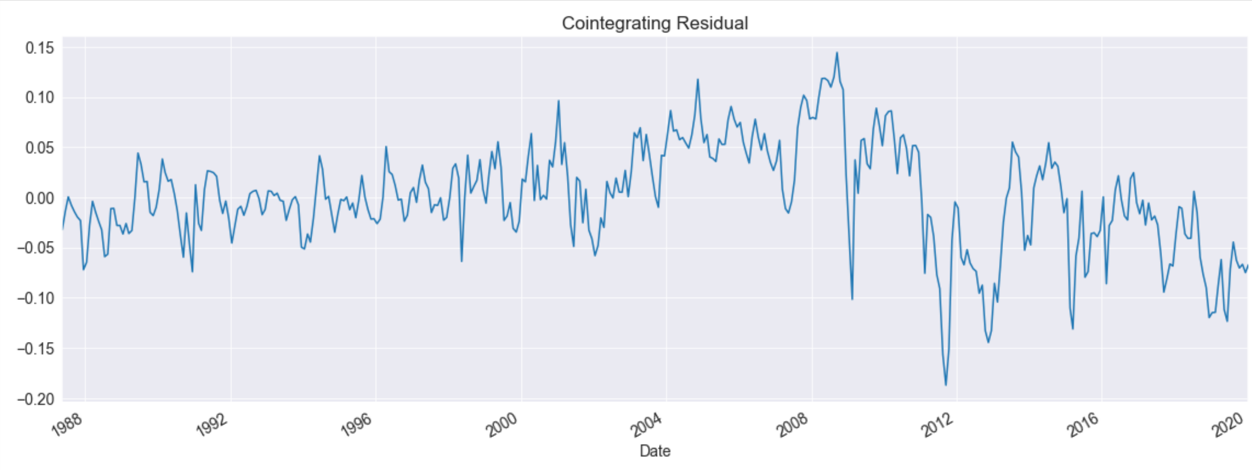
[15]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
Phillips-Ouliaris Pu Cointegration Test
=====================================
Test Statistic                102.868
P-value                         0.000
Kernel                       Bartlett
Bandwidth                      14.648
-------------------------------------

Trend: Constant
Critical Values: 27.01 (10%), 32.93 (5%), 46.01 (1%)
Null Hypothesis: No Cointegration
Alternative Hypothesis: Cointegration
Distribution Order: 2
"""
```

[16]:
```
po_pz_test = phillips_ouliaris(
    log_price.WTI, log_price.Brent, trend="c", test_type="Pz"
)
po_pz_test.summary()
```

[16]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
Phillips-Ouliaris Pz Cointegration Test
=====================================
Test Statistic                114.601
P-value                         0.000
Kernel                       Bartlett
Bandwidth                      14.648
-------------------------------------

Trend: Constant
Critical Values: 45.39 (10%), 52.41 (5%), 67.39 (1%)
Null Hypothesis: No Cointegration
Alternative Hypothesis: Cointegration
Distribution Order: 2
"""
```

The cointegrating residual is identical to the EG test since the first step is identical.

[17]:
```
fig = po_zt_test.plot()
```



5.2 Cointegration Tests

engle_granger(y, x[, trend, lags, max_lags, ...])	Test for cointegration within a set of time series.
phillips_ouliaris(y, x[, trend, test_type, ...])	Test for cointegration within a set of time series.

5.2.1 arch.unitroot.cointegration.engle_granger

arch.unitroot.cointegration.**engle_granger**(y: *ndarray* | *Series*, x: *ndarray* | *DataFrame*, trend: *'n'* | *'c'* | *'ct'* | *'ctt'* = 'c', *, lags: *int* | *None* = **None**, max_lags: *int* | *None* = **None**, method: *'aic'* | *'bic'* | *'t-stat'* = 'bic') → *EngleGrangerTestResults*

Test for cointegration within a set of time series.

Parameters

y: ndarray | Series
The left-hand-side variable in the cointegrating regression.

x: ndarray | DataFrame
The right-hand-side variables in the cointegrating regression.

trend: 'n' | 'c' | 'ct' | 'ctt' = 'c'

Trend to include in the cointegrating regression. Trends are:

- ”n”: No deterministic terms
- ”c”: Constant
- ”ct”: Constant and linear trend
- ”ctt”: Constant, linear and quadratic trends

lags: int | None = None
The number of lagged differences to include in the Augmented Dickey-Fuller test used on the residuals of the

max_lags: int | None = None
The maximum number of lags to consider when using automatic lag-length in the Augmented Dickey-Fuller regression.

method: 'aic' | 'bic' | 't-stat' = 'bic'
The method used to select the number of lags included in the Augmented Dickey-Fuller regression.

Returns
Results of the Engle-Granger test.

Return type
EngleGrangerTestResults

See also

arch.unitroot.ADF
Augmented Dickey-Fuller testing.

arch.unitroot.PhillipsPerron
Phillips & Perron's unit root test.

arch.unitroot.cointegration.phillips_ouliaris
Phillips-Ouliaris tests of cointegration.

Notes

The model estimated is

$$Y_t = X_t\beta + D_t\gamma + \epsilon_t$$

where $Z_t = [Y_t, X_t]$ is being tested for cointegration. D_t is a set of deterministic terms that may include a constant, a time trend or a quadratic time trend.

The null hypothesis is that the series are not cointegrated.

The test is implemented as an ADF of the estimated residuals from the cross-sectional regression using a set of critical values that is determined by the number of assumed stochastic trends when the null hypothesis is true.

5.2.2 arch.unitroot.cointegration.phillips_ouliaris

```
arch.unitroot.cointegration.phillips_ouliaris(y: ndarray | Series, x: ndarray | DataFrame, trend: 'n' | 'c' | 'ct' | 'ctt' = 'c', *, test_type: 'Za' | 'Zt' | 'Pu' | 'Pz' = 'Zt', kernel: str = 'bartlett', bandwidth: int | None = None, force_int: bool = False) → PhillipsOuliarisTestResults
```

Test for cointegration within a set of time series.

Parameters

y: ndarray | Series
The left-hand-side variable in the cointegrating regression.

x: ndarray | DataFrame
The right-hand-side variables in the cointegrating regression.

trend: 'n' | 'c' | 'ct' | 'ctt' = `'c'`

Trend to include in the cointegrating regression. Trends are:

- "n": No deterministic terms
- "c": Constant
- "ct": Constant and linear trend
- "ctt": Constant, linear and quadratic trends

test_type: 'Za' | 'Zt' | 'Pu' | 'Pz' = `'Zt'`

The test statistic to compute. Supported options are:

- "Za": The Z test based on the the debiased AR(1) coefficient.
- "Zt": The Zt test based on the t-statistic from an AR(1).
- "Pu": The P variance-ratio test.
- "Pz": The Pz test of the trace of the product of an estimate of the long-run residual variance and the inner-product of the data.

See the notes for details on the test.

kernel: str = `'bartlett'`
The string name of any of any known kernel-based long-run covariance estimators. Common choices are "bartlett" for the Bartlett kernel (Newey-West), "parzen" for the Parzen kernel and "quadratic-spectral" for the Quadratic Spectral kernel.

bandwidth: int | None = `None`
The bandwidth to use. If not provided, the optimal bandwidth is estimated from the data. Setting the bandwidth to 0 and using "unadjusted" produces the classic OLS covariance estimator. Setting the bandwidth to 0 and using "robust" produces White's covariance estimator.

force_int: bool = `False`
Whether the force the estimated optimal bandwidth to be an integer.

Returns
Results of the Phillips-Ouliaris test.

Return type
PhillipsOuliarisTestResults

See also

arch.unitroot.ADF
Augmented Dickey-Fuller testing.

arch.unitroot.PhillipsPerron
Phillips & Perron's unit root test.

arch.unitroot.cointegration.engle_granger
Engle & Granger's cointegration test.

Notes

Warning

The critical value simulation is on-going and so the critical values may change slightly as more simulations are completed. These are still based on far more simulations (minimum 2,000,000) than were possible in 1990 (5000) that are reported in[1].

Supports 4 distinct tests.

Define the cross-sectional regression

$$y_t = x_t\beta + d_t\gamma + u_t$$

where d_t are any included deterministic terms. Let $\hat{u}_t = y_t - x_t\hat{\beta} + d_t\hat{\gamma}$.

The Z and Zt statistics are defined as

$$\hat{Z}_\alpha = T \times z$$
$$\hat{Z}_t = \frac{\hat{\sigma}_u}{\hat{\omega}^2} \times \sqrt{T}z$$
$$z = (\hat{\alpha} - 1) - \hat{\omega}_1^2/\hat{\sigma}_u^2$$

where $\hat{\sigma}_u^2 = T^{-1}\sum_{t=2}^{T}\hat{u}_t^2$, $\hat{\omega}_1^2$ is an estimate of the one-sided strict autocovariance, and $\hat{\omega}^2$ is an estimate of the long-run variance of the process.

The $\hat{P}_u$ variance-ratio statistic is defined as

$$\hat{P}_u = \frac{\hat{\omega}_{11\cdot 2}}{\tilde{\sigma}_u^2}$$

where $\tilde{\sigma}_u^2 = T^{-1}\sum_{t=1}^{T}\hat{u}_t^2$ and

$$\hat{\omega}_{11\cdot 2} = \hat{\omega}_{11} - \hat{\omega}_{21}'\hat{\Omega}_{22}^{-1}\hat{\omega}_{21}$$

and

$$\hat{\Omega} = \left[\begin{array}{cc} \hat{\omega}_{11} & \hat{\omega}_{21}' \\ \hat{\omega}_{21} & \hat{\Omega}_{22} \end{array}\right]$$

is an estimate of the long-run covariance of ξ_t, the residuals from an VAR(1) on $z_t = [y_t, z_t]$ that includes and trends included in the test.

$$z_t = \Phi z_{t-1} + \xi_\tau$$

The final test statistic is defined

$$\hat{P}_z = T \times \mathrm{tr}(\hat{\Omega}M_{zz}^{-1})$$

where $M_{zz} = \sum_{t=1}^{T}\tilde{z}_t'\tilde{z}_t$, $\tilde{z}_t$ is the vector of data $z_t = [y_t, x_t]$ detrended using any trend terms included in the test, $\tilde{z}_t = z_t - d_t\hat{\kappa}$ and $\hat{\Omega}$ is defined above.

The specification of the $\hat{P}_z$ test statistic when trend is "n" differs from the expression in[1]. We recenter z_t by subtracting the first observation, so that $\tilde{z}_t = z_t - z_1$. This is needed to ensure that the initial value does not affect the distribution under the null. When the trend is anything other than "n", this set is not needed and the test statistics is identical whether the first observation is subtracted or not.

[1] Phillips, P. C., & Ouliaris, S. (1990). Asymptotic properties of residual based tests for cointegration. Econometrica: Journal of the Econometric Society, 165-193.

References

5.3 Cointegrating Vector Estimation

DynamicOLS(y, x[, trend, lags, leads, ...])	Dynamic OLS (DOLS) cointegrating vector estimation
FullyModifiedOLS(y, x[, trend, x_trend])	Fully Modified OLS cointegrating vector estimation.
CanonicalCointegratingReg(y, x[, trend, x_trend])	Canonical Cointegrating Regression cointegrating vector estimation.

5.3.1 arch.unitroot.cointegration.DynamicOLS

class arch.unitroot.cointegration.**DynamicOLS**(y: *ndarray* | *Series*, x: *ndarray* | *DataFrame*, trend: *'n'* | *'c'* | *'ct'* | *'ctt'* = 'c', lags: *int* | *None* = **None**, leads: *int* | *None* = **None**, common: *bool* = **False**, max_lag: *int* | *None* = **None**, max_lead: *int* | *None* = **None**, method: *'aic'* | *'bic'* | *'hqic'* = 'bic')

Dynamic OLS (DOLS) cointegrating vector estimation

Parameters

y: ndarray | Series
The left-hand-side variable in the cointegrating regression.

x: ndarray | DataFrame
The right-hand-side variables in the cointegrating regression.

trend: 'n' | 'c' | 'ct' | 'ctt' = 'c'

Trend to include in the cointegrating regression. Trends are:

- "n": No deterministic terms
- "c": Constant
- "ct": Constant and linear trend
- "ctt": Constant, linear and quadratic trends

lags: int | None = None
The number of lags to include in the model. If None, the optimal number of lags is chosen using method.

leads: int | None = None
The number of leads to include in the model. If None, the optimal number of leads is chosen using method.

common: bool = False
Flag indicating that lags and leads should be restricted to the same value. When common is None, lags must equal leads and max_lag must equal max_lead.

max_lag: int | None = None
The maximum lag to consider. See Notes for value used when None.

max_lead: int | None = None
The maximum lead to consider. See Notes for value used when None.

method: 'aic' | 'bic' | 'hqic' = 'bic'

The method used to select lag length when lags or leads is None.

- "aic" - Akaike Information Criterion
- "hqic" - Hannan-Quinn Information Criterion
- "bic" - Schwartz/Bayesian Information Criterion

Notes

The cointegrating vector is estimated from the regression

$$Y_t = D_t\delta + X_t\beta + \Delta X_t\gamma + \sum_{i=1}^{p} \Delta X_{t-i}\kappa_i + \sum_{j=1}^{q} \Delta X_{t+j}\lambda_j + \epsilon_t$$

where p is the lag length and q is the lead length. D_t is a vector containing the deterministic terms, if any. All specifications include the contemporaneous difference ΔX_t.

When lag lengths are not provided, the optimal lag length is chosen to minimize an Information Criterion of the form

$$\ln\left(\hat{\sigma}^2\right) + k\frac{c}{T}$$

where c is 2 for Akaike, $2 \ln \ln T$ for Hannan-Quinn and $\ln T$ for Schwartz/Bayesian.

See[1] and[2] for further details.

References

Methods

fit([cov_type, kernel, bandwidth, ...])	Estimate the Dynamic OLS regression

arch.unitroot.cointegration.DynamicOLS.fit

DynamicOLS.**fit**(cov_type: *'unadjusted' | 'homoskedastic' | 'robust' | 'kernel'* = 'unadjusted', kernel: *str* = 'bartlett', bandwidth: *int | None* = **None**, force_int: *bool* = **False**, df_adjust: *bool* = **False**) → *DynamicOLSResults*

Estimate the Dynamic OLS regression

Parameters

cov_type: 'unadjusted' | 'homoskedastic' | 'robust' | 'kernel' = 'unadjusted'
Either "unadjusted" (or is equivalent "homoskedastic") or "robust" (or its equivalent "kernel").

kernel: str = 'bartlett'
The string name of any of any known kernel-based long-run covariance estimators. Common choices are "bartlett" for the Bartlett kernel (Newey-West), "parzen" for the Parzen kernel and "quadratic-spectral" for the Quadratic Spectral kernel.

[1] Saikkonen, P. (1992). Estimation and testing of cointegrated systems by an autoregressive approximation. Econometric theory, 8(1), 1-27.
[2] Stock, J. H., & Watson, M. W. (1993). A simple estimator of cointegrating vectors in higher order integrated systems. Econometrica: Journal of the Econometric Society, 783-820.

bandwidth: int | None = None
The bandwidth to use. If not provided, the optimal bandwidth is estimated from the data. Setting the bandwidth to 0 and using "unadjusted" produces the classic OLS covariance estimator. Setting the bandwidth to 0 and using "robust" produces White's covariance estimator.

force_int: bool = False
Whether the force the estimated optimal bandwidth to be an integer.

df_adjust: bool = False
Whether the adjust the parameter covariance to account for the number of parameters estimated in the regression. If true, the parameter covariance estimator is multiplied by T/(T-k) where k is the number of regressors in the model.

Returns
The estimation results.

Return type
`DynamicOLSResults`

See also

`arch.unitroot.cointegration.engle_granger`
Cointegration testing using the Engle-Granger methodology

`statsmodels.regression.linear_model.OLS`
Ordinal Least Squares regression.

Notes

When using the unadjusted covariance, the parameter covariance is estimated as

$$T^{-1}\hat{\sigma}^2_{HAC}\hat{\Sigma}^{-1}_{ZZ}$$

where $\hat{\sigma}^2_{HAC}$ is an estimator of the long-run variance of the regression error and $\hat{\Sigma}_{ZZ} = T^{-1}Z'Z$. Z_t is a vector the includes all terms in the regression (i.e., deterministics, cross-sectional, leads and lags) When using the robust covariance, the parameter covariance is estimated as

$$T^{-1}\hat{\Sigma}^{-1}_{ZZ}\hat{S}_{HAC}\hat{\Sigma}^{-1}_{ZZ}$$

where $\hat{S}_{HAC}$ is a Heteroskedasticity-Autocorrelation Consistent estimator of the covariance of the regression scores $Z_t\epsilon_t$.

5.3.2 arch.unitroot.cointegration.FullyModifiedOLS

class arch.unitroot.cointegration.**FullyModifiedOLS**(*y: ndarray | Series, x: ndarray | DataFrame, trend: 'n' | 'c' | 'ct' | 'ctt' = 'c', x_trend: 'n' | 'c' | 'ct' | 'ctt' | None = None*)

Fully Modified OLS cointegrating vector estimation.

Parameters

y: ndarray | Series
The left-hand-side variable in the cointegrating regression.

x: ndarray | DataFrame
The right-hand-side variables in the cointegrating regression.

trend: 'n' | 'c' | 'ct' | 'ctt' = 'c'

Trend to include in the cointegrating regression. Trends are:

- "n": No deterministic terms
- "c": Constant
- "ct": Constant and linear trend
- "ctt": Constant, linear and quadratic trends

x_trend: 'n' | 'c' | 'ct' | 'ctt' | None = None
Trends that affects affect the x-data but do not appear in the cointegrating regression. x_trend must be at least as large as trend, so that if trend is "ct", x_trend must be either "ct" or "ctt".

Notes

The cointegrating vector is estimated from the regressions

$$\begin{aligned} Y_t &= D_{1t}\delta + X_t\beta + \eta_{1t} \\ X_t &= D_{1t}\Gamma_1 + D_{2t}\Gamma_2 + \epsilon_{2t} \\ \eta_{2t} &= \Delta\epsilon_{2t} \end{aligned}$$

or if estimated in differences, the last two lines are

$$\Delta X_t = \Delta D_{1t}\Gamma_1 + \Delta D_{2t}\Gamma_2 + \eta_{2t}$$

Define the vector of residuals as $\eta = (\eta_{1t}, \eta'_{2t})'$, and the long-run covariance

$$\Omega = \sum_{h=-\infty}^{\infty} E[\eta_t\eta'_{t-h}]$$

and the one-sided long-run covariance matrix

$$\Lambda_0 = \sum_{h=0}^{\infty} E[\eta_t\eta'_{t-h}]$$

The covariance matrices are partitioned into a block form

$$\Omega = \left[\begin{array}{cc} \omega_{11} & \omega_{12} \\ \omega'_{12} & \Omega_{22} \end{array}\right]$$

The cointegrating vector is then estimated using modified data

$$Y^\star_t = Y_t - \hat{\omega}_{12}\hat{\Omega}_{22}\hat{\eta}_{2t}$$

as

$$\hat{\theta} = \left[\begin{array}{c} \hat{\gamma}_1 \\ \hat{\beta} \end{array}\right] = \left(\sum_{t=2}^{T} Z_t Z'_t\right)^{-1} \left(\sum_{t=2}^{t} Z_t Y^\star_t - T\left[\begin{array}{c} 0 \\ \lambda^{\star\prime}_{12} \end{array}\right]\right)$$

where the bias term is defined

$$\lambda^\star_{12} = \hat{\lambda}_{12} - \hat{\omega}_{12}\hat{\Omega}_{22}\hat{\omega}_{21}$$

See[1] for further details.

[1] Hansen, B. E., & Phillips, P. C. (1990). Estimation and inference in models of cointegration: A simulation study. Advances in Econometrics, 8(1989), 225-248.

References

Methods

`fit`([kernel, bandwidth, force_int, diff, ...])	Estimate the cointegrating vector.

arch.unitroot.cointegration.FullyModifiedOLS.fit

FullyModifiedOLS.**fit**(kernel: *str* = 'bartlett', bandwidth: *float* | *None* = ***None***, force_int: *bool* = ***True***, diff: *bool* = ***False***, df_adjust: *bool* = ***False***) → *CointegrationAnalysisResults*

Estimate the cointegrating vector.

Parameters

diff: bool = False
Use differenced data to estimate the residuals.

kernel: str = 'bartlett'
The string name of any of any known kernel-based long-run covariance estimators. Common choices are "bartlett" for the Bartlett kernel (Newey-West), "parzen" for the Parzen kernel and "quadratic-spectral" for the Quadratic Spectral kernel.

bandwidth: float | None = None
The bandwidth to use. If not provided, the optimal bandwidth is estimated from the data. Setting the bandwidth to 0 and using "unadjusted" produces the classic OLS covariance estimator. Setting the bandwidth to 0 and using "robust" produces White's covariance estimator.

force_int: bool = True
Whether the force the estimated optimal bandwidth to be an integer.

df_adjust: bool = False
Whether the adjust the parameter covariance to account for the number of parameters estimated in the regression. If true, the parameter covariance estimator is multiplied by T/(T-k) where k is the number of regressors in the model.

Returns
The estimation results instance.

Return type
`CointegrationAnalysisResults`

5.3.3 arch.unitroot.cointegration.CanonicalCointegratingReg

class arch.unitroot.cointegration.**CanonicalCointegratingReg**(y: *ndarray* | *Series*, x: *ndarray* | *DataFrame*, trend: *'n'* | *'c'* | *'ct'* | *'ctt'* = 'c', x_trend: *'n'* | *'c'* | *'ct'* | *'ctt'* | *None* = ***None***)

Canonical Cointegrating Regression cointegrating vector estimation.

Parameters

y: ndarray | Series
The left-hand-side variable in the cointegrating regression.

x: ndarray | DataFrame
The right-hand-side variables in the cointegrating regression.

trend: 'n' | 'c' | 'ct' | 'ctt' = `'c'`

Trend to include in the cointegrating regression. Trends are:

- "n": No deterministic terms
- "c": Constant
- "ct": Constant and linear trend
- "ctt": Constant, linear and quadratic trends

x_trend: 'n' | 'c' | 'ct' | 'ctt' | None = `None`
Trends that affects affect the x-data but do not appear in the cointegrating regression. x_trend must be at least as large as trend, so that if trend is "ct", x_trend must be either "ct" or "ctt".

Notes

The cointegrating vector is estimated from the regressions

$$\begin{aligned} Y_t &= D_{1t}\delta + X_t\beta + \eta_{1t} \\ X_t &= D_{1t}\Gamma_1 + D_{2t}\Gamma_2 + \epsilon_{2t} \\ \eta_{2t} &= \Delta\epsilon_{2t} \end{aligned}$$

or if estimated in differences, the last two lines are

$$\Delta X_t = \Delta D_{1t}\Gamma_1 + \Delta D_{2t}\Gamma_2 + \eta_{2t}$$

Define the vector of residuals as $\eta = (\eta_{1t}, \eta'_{2t})'$, and the long-run covariance

$$\Omega = \sum_{h=-\infty}^{\infty} E[\eta_t\eta'_{t-h}]$$

and the one-sided long-run covariance matrix

$$\Lambda_0 = \sum_{h=0}^{\infty} E[\eta_t\eta'_{t-h}]$$

The covariance matrices are partitioned into a block form

$$\Omega = \left[\begin{array}{cc} \omega_{11} & \omega_{12} \\ \omega'_{12} & \Omega_{22} \end{array}\right]$$

The cointegrating vector is then estimated using modified data

$$\begin{aligned} X^\star_t &= X_t - \hat{\Lambda}'_2\hat{\Sigma}^{-1}\hat{\eta}_t \\ Y^\star_t &= Y_t - (\hat{\Sigma}^{-1}\hat{\Lambda}_2\hat{\beta} + \hat{\kappa})'\hat{\eta}_t \end{aligned}$$

where $\hat{\kappa} = (0, \hat{\Omega}_{22}^{-1}\hat{\Omega}'_{12})$ and the regression

$$Y^\star_t = D_{1t}\delta + X^\star_t\beta + \eta^\star_{1t}$$

See[1] for further details.

[1] Park, J. Y. (1992). Canonical cointegrating regressions. Econometrica: Journal of the Econometric Society, 119-143.

References

Methods

`fit`([kernel, bandwidth, force_int, diff, ...])	Estimate the cointegrating vector.

arch.unitroot.cointegration.CanonicalCointegratingReg.fit

`CanonicalCointegratingReg.fit`(kernel: *str* = 'bartlett', bandwidth: *float* | *None* = **None**, force_int: *bool* = **True**, diff: *bool* = **False**, df_adjust: *bool* = **False**) → *CointegrationAnalysisResults*

Estimate the cointegrating vector.

Parameters

diff: bool = False
Use differenced data to estimate the residuals.

kernel: str = 'bartlett'
The string name of any of any known kernel-based long-run covariance estimators. Common choices are "bartlett" for the Bartlett kernel (Newey-West), "parzen" for the Parzen kernel and "quadratic-spectral" for the Quadratic Spectral kernel.

bandwidth: float | None = None
The bandwidth to use. If not provided, the optimal bandwidth is estimated from the data. Setting the bandwidth to 0 and using "unadjusted" produces the classic OLS covariance estimator. Setting the bandwidth to 0 and using "robust" produces White's covariance estimator.

force_int: bool = True
Whether the force the estimated optimal bandwidth to be an integer.

df_adjust: bool = False
Whether the adjust the parameter covariance to account for the number of parameters estimated in the regression. If true, the parameter covariance estimator is multiplied by T/(T-k) where k is the number of regressors in the model.

Returns
The estimation results instance.

Return type
CointegrationAnalysisResults

5.3.4 Results Classes

`CointegrationAnalysisResults`(params, cov, ...)	
`DynamicOLSResults`(params, cov, resid, lags, ...)	Estimation results for Dynamic OLS models
`EngleGrangerTestResults`(stat, pvalue, crit_vals)	Results class for Engle-Granger cointegration tests.
`PhillipsOuliarisTestResults`(stat, pvalue, ...)	

arch.unitroot.cointegration.CointegrationAnalysisResults

class arch.unitroot.cointegration.**CointegrationAnalysisResults**(*params: pd.Series*, *cov: pd.DataFrame*, *resid: pd.Series*, *omega_112: float*, *kernel_est: lrcov.CovarianceEstimator*, *num_x: int*, *trend: UnitRootTrend*, *df_adjust: bool*, *rsquared: float*, *rsquared_adj: float*, *estimator_type: str*)

Methods

summary()	Summary of the model, containing estimated parameters and std.

arch.unitroot.cointegration.CointegrationAnalysisResults.summary

CointegrationAnalysisResults.**summary**() → Summary

Summary of the model, containing estimated parameters and std. errors

Returns
A summary instance with method that support export to text, csv or latex.

Return type
Summary

Properties

bandwidth	The bandwidth used in the parameter covariance estimation
cov	The estimated parameter covariance of the cointegrating vector
kernel	The kernel used to estimate the covariance
long_run_variance	Long-run variance estimate used in the parameter covariance estimator
params	The estimated parameters of the cointegrating vector
pvalues	P-value of the parameters in the cointegrating vector
resid	The model residuals
residual_variance	The variance of the regression residual.
rsquared	The model R^2
rsquared_adj	The degree-of-freedom adjusted R^2
std_errors	Standard errors of the parameters in the cointegrating vector
tvalues	T-statistics of the parameters in the cointegrating vector

arch.unitroot.cointegration.CointegrationAnalysisResults.bandwidth

property `CointegrationAnalysisResults.bandwidth` : float

The bandwidth used in the parameter covariance estimation

arch.unitroot.cointegration.CointegrationAnalysisResults.cov

property `CointegrationAnalysisResults.cov` : pd.DataFrame

The estimated parameter covariance of the cointegrating vector

arch.unitroot.cointegration.CointegrationAnalysisResults.kernel

property `CointegrationAnalysisResults.kernel` : str

The kernel used to estimate the covariance

arch.unitroot.cointegration.CointegrationAnalysisResults.long_run_variance

property `CointegrationAnalysisResults.long_run_variance` : float

Long-run variance estimate used in the parameter covariance estimator

arch.unitroot.cointegration.CointegrationAnalysisResults.params

property `CointegrationAnalysisResults.params` : pandas.Series

The estimated parameters of the cointegrating vector

arch.unitroot.cointegration.CointegrationAnalysisResults.pvalues

property `CointegrationAnalysisResults.pvalues` : pandas.Series

P-value of the parameters in the cointegrating vector

arch.unitroot.cointegration.CointegrationAnalysisResults.resid

property `CointegrationAnalysisResults.resid` : pandas.Series

The model residuals

arch.unitroot.cointegration.CointegrationAnalysisResults.residual_variance

property `CointegrationAnalysisResults.residual_variance` : float

The variance of the regression residual.

Returns

The estimated residual variance.

Return type

`float`

Notes

The residual variance only accounts for the short-run variance of the residual and does not account for any autocorrelation. It is defined as

$$\hat{\sigma}^2 = T^{-1} \sum_{t=p}^{T-q} \hat{\epsilon}_t^2$$

If *df_adjust* is True, then the estimator is rescaled by T/(T-m) where m is the number of regressors in the model.

arch.unitroot.cointegration.CointegrationAnalysisResults.rsquared

property `CointegrationAnalysisResults.rsquared` : float

The model R^2

arch.unitroot.cointegration.CointegrationAnalysisResults.rsquared_adj

property `CointegrationAnalysisResults.rsquared_adj` : float

The degree-of-freedom adjusted R^2

arch.unitroot.cointegration.CointegrationAnalysisResults.std_errors

property `CointegrationAnalysisResults.std_errors` : pandas.Series

Standard errors of the parameters in the cointegrating vector

arch.unitroot.cointegration.CointegrationAnalysisResults.tvalues

property `CointegrationAnalysisResults.tvalues` : pandas.Series

T-statistics of the parameters in the cointegrating vector

arch.unitroot.cointegration.DynamicOLSResults

class `arch.unitroot.cointegration.DynamicOLSResults`(params: *pd.Series*, cov: *pd.DataFrame*, resid: *pd.Series*, lags: *int*, leads: *int*, cov_type: *str*, kernel_est: *lrcov.CovarianceEstimator*, *num_x: int*, *trend: UnitRootTrend*, reg_results: *RegressionResults*, df_adjust: *bool*)

Estimation results for Dynamic OLS models

Parameters

params: pd.Series
The estimated model parameters.

cov: pd.DataFrame
The estimated parameter covariance.

resid: pd.Series
The model residuals.

lags: int
The number of lags included in the model.

leads: int
The number of leads included in the model.

cov_type: str
The type of the parameter covariance estimator used.

kernel_est: lrcov.CovarianceEstimator
The covariance estimator instance used to estimate the parameter covariance.

reg_results: RegressionResults
Regression results from fitting statsmodels OLS.

df_adjust: bool
Whether to degree of freedom adjust the estimator.

Methods

summary([full])	Summary of the model, containing estimated parameters and std.

arch.unitroot.cointegration.DynamicOLSResults.summary

`DynamicOLSResults.summary(full: bool = False) → Summary`

Summary of the model, containing estimated parameters and std. errors

Parameters

full: bool = False
Flag indicating whether to include all estimated parameters (True) or only the parameters of the cointegrating vector

Returns
A summary instance with method that support export to text, csv or latex.

Return type
`Summary`

Properties

bandwidth	The bandwidth used in the parameter covariance estimation
cov	The estimated parameter covariance of the cointegrating vector
cov_type	The type of parameter covariance estimator used
full_cov	Parameter covariance of the all model parameters, incl.
full_params	The complete set of parameters, including leads and lags
kernel	The kernel used to estimate the covariance
lags	The number of lags included in the model
leads	The number of leads included in the model
long_run_variance	The long-run variance of the regression residual.
params	The estimated parameters of the cointegrating vector
pvalues	P-value of the parameters in the cointegrating vector
resid	The model residuals
residual_variance	The variance of the regression residual.
rsquared	The model R^2
rsquared_adj	The degree-of-freedom adjusted R^2
std_errors	Standard errors of the parameters in the cointegrating vector
tvalues	T-statistics of the parameters in the cointegrating vector

arch.unitroot.cointegration.DynamicOLSResults.bandwidth

property DynamicOLSResults.**bandwidth** : float

The bandwidth used in the parameter covariance estimation

arch.unitroot.cointegration.DynamicOLSResults.cov

property DynamicOLSResults.**cov** : pd.DataFrame

The estimated parameter covariance of the cointegrating vector

arch.unitroot.cointegration.DynamicOLSResults.cov_type

property DynamicOLSResults.**cov_type** : str

The type of parameter covariance estimator used

arch.unitroot.cointegration.DynamicOLSResults.full_cov

property DynamicOLSResults.**full_cov** : pd.DataFrame

Parameter covariance of the all model parameters, incl. leads and lags

arch.unitroot.cointegration.DynamicOLSResults.full_params

property DynamicOLSResults.**full_params** : pandas.Series

The complete set of parameters, including leads and lags

arch.unitroot.cointegration.DynamicOLSResults.kernel

property DynamicOLSResults.**kernel** : str

The kernel used to estimate the covariance

arch.unitroot.cointegration.DynamicOLSResults.lags

property DynamicOLSResults.**lags** : int

The number of lags included in the model

arch.unitroot.cointegration.DynamicOLSResults.leads

property DynamicOLSResults.**leads** : int

The number of leads included in the model

arch.unitroot.cointegration.DynamicOLSResults.long_run_variance

property DynamicOLSResults.**long_run_variance** : float

The long-run variance of the regression residual.

Returns

The estimated long-run variance of the residual.

Return type

`float`

Notes

The long-run variance is estimated from the model residuals using the same kernel used to estimate the parameter covariance.

If *df_adjust* is True, then the estimator is rescaled by T/(T-m) where m is the number of regressors in the model.

arch.unitroot.cointegration.DynamicOLSResults.params

property `DynamicOLSResults.params` : pandas.Series

The estimated parameters of the cointegrating vector

arch.unitroot.cointegration.DynamicOLSResults.pvalues

property `DynamicOLSResults.pvalues` : pandas.Series

P-value of the parameters in the cointegrating vector

arch.unitroot.cointegration.DynamicOLSResults.resid

property `DynamicOLSResults.resid` : pandas.Series

The model residuals

arch.unitroot.cointegration.DynamicOLSResults.residual_variance

property `DynamicOLSResults.residual_variance` : float

The variance of the regression residual.

Returns
The estimated residual variance.

Return type
`float`

Notes

The residual variance only accounts for the short-run variance of the residual and does not account for any autocorrelation. It is defined as

$$\hat{\sigma}^2 = T^{-1} \sum_{t=p}^{T-q} \hat{\epsilon}_t^2$$

If *df_adjust* is True, then the estimator is rescaled by T/(T-m) where m is the number of regressors in the model.

arch.unitroot.cointegration.DynamicOLSResults.rsquared

property `DynamicOLSResults.rsquared` : float

The model R^2

arch.unitroot.cointegration.DynamicOLSResults.rsquared_adj

property `DynamicOLSResults.rsquared_adj` : float

The degree-of-freedom adjusted R^2

arch.unitroot.cointegration.DynamicOLSResults.std_errors

property `DynamicOLSResults.std_errors` : pandas.Series

Standard errors of the parameters in the cointegrating vector

arch.unitroot.cointegration.DynamicOLSResults.tvalues

property `DynamicOLSResults.tvalues` : pandas.Series

T-statistics of the parameters in the cointegrating vector

arch.unitroot.cointegration.EngleGrangerTestResults

class `arch.unitroot.cointegration.EngleGrangerTestResults`(stat: *float*, pvalue: *float*, crit_vals: *pandas.Series*, null: *str* = `'No Cointegration'`, alternative: *str* = `'Cointegration'`, trend: *str* = `'c'`, order: *int* = *2*, adf: ADF | *None* = ***None***, xsection: *RegressionResults* | *None* = ***None***)

Results class for Engle-Granger cointegration tests.

Parameters

stat: float
The Engle-Granger test statistic.

pvalue: float
The pvalue of the Engle-Granger test statistic.

crit_vals: pandas.Series
The critical values of the Engle-Granger specific to the sample size and model dimension.

null: str = `'No Cointegration'`
The null hypothesis.

alternative: str = `'Cointegration'`
The alternative hypothesis.

trend: str = `'c'`
The model's trend description.

order: int = 2
The number of stochastic trends in the null distribution.

adf: *ADF* | None = None
The ADF instance used to perform the test and lag selection.

xsection: RegressionResults | None = None
The OLS results used in the cross-sectional regression.

Methods

plot([axes, title])	Plot the cointegration residuals.
summary()	Summary of test, containing statistic, p-value and critical values

arch.unitroot.cointegration.EngleGrangerTestResults.plot

EngleGrangerTestResults.**plot**(axes: *Axes* | *None* = ***None***, title: *str* | *None* = ***None***) → Figure

Plot the cointegration residuals.

Parameters

axes: Axes | None = None
matplotlib axes instance to hold the figure.

title: str | None = None
Title for the figure.

Returns
The matplotlib Figure instance.

Return type
`matplotlib.figure.Figure`

arch.unitroot.cointegration.EngleGrangerTestResults.summary

EngleGrangerTestResults.**summary**() → Summary

Summary of test, containing statistic, p-value and critical values

Properties

alternative_hypothesis	The alternative hypothesis
cointegrating_vector	The estimated cointegrating vector.
critical_values	Critical Values
distribution_order	The number of stochastic trends under the null hypothesis.
lags	The number of lags used in the Augmented Dickey-Fuller regression.
max_lags	The maximum number of lags used in the lag-length selection.
name	Sets or gets the name of the cointegration test
null_hypothesis	The null hypothesis
pvalue	The p-value of the test statistic.
resid	The residual from the cointegrating regression.
rho	The estimated coefficient in the Dickey-Fuller Test
stat	The test statistic.
trend	The trend used in the cointegrating regression

arch.unitroot.cointegration.EngleGrangerTestResults.alternative_hypothesis

property `EngleGrangerTestResults.alternative_hypothesis` : str

The alternative hypothesis

arch.unitroot.cointegration.EngleGrangerTestResults.cointegrating_vector

property `EngleGrangerTestResults.cointegrating_vector` : pandas.Series

The estimated cointegrating vector.

arch.unitroot.cointegration.EngleGrangerTestResults.critical_values

property `EngleGrangerTestResults.critical_values` : pandas.Series

Critical Values

Returns

Series with three keys, 1, 5 and 10 containing the critical values of the test statistic.

Return type

`pandas.Series`

arch.unitroot.cointegration.EngleGrangerTestResults.distribution_order

property `EngleGrangerTestResults.distribution_order` : int

The number of stochastic trends under the null hypothesis.

arch.unitroot.cointegration.EngleGrangerTestResults.lags

property `EngleGrangerTestResults.lags` : int

The number of lags used in the Augmented Dickey-Fuller regression.

arch.unitroot.cointegration.EngleGrangerTestResults.max_lags

property `EngleGrangerTestResults.max_lags` : int | None

The maximum number of lags used in the lag-length selection.

arch.unitroot.cointegration.EngleGrangerTestResults.name

property `EngleGrangerTestResults.name` : str

Sets or gets the name of the cointegration test

arch.unitroot.cointegration.EngleGrangerTestResults.null_hypothesis

property EngleGrangerTestResults.**null_hypothesis** : str

The null hypothesis

arch.unitroot.cointegration.EngleGrangerTestResults.pvalue

property EngleGrangerTestResults.**pvalue** : float

The p-value of the test statistic.

arch.unitroot.cointegration.EngleGrangerTestResults.resid

property EngleGrangerTestResults.**resid** : pandas.Series

The residual from the cointegrating regression.

arch.unitroot.cointegration.EngleGrangerTestResults.rho

property EngleGrangerTestResults.**rho** : float

The estimated coefficient in the Dickey-Fuller Test

Returns
The coefficient.

Return type
`float`

Notes

The value returned is $\hat{\rho} = \hat{\gamma} + 1$ from the ADF regression

$$\Delta y_t = \gamma y_{t-1} + \sum_{i=1}^{p} \delta_i \Delta y_{t-i} + \epsilon_t$$

arch.unitroot.cointegration.EngleGrangerTestResults.stat

property EngleGrangerTestResults.**stat** : float

The test statistic.

arch.unitroot.cointegration.EngleGrangerTestResults.trend

property EngleGrangerTestResults.**trend** : str

The trend used in the cointegrating regression

arch.unitroot.cointegration.PhillipsOuliarisTestResults

class arch.unitroot.cointegration.**PhillipsOuliarisTestResults**(*stat: float, pvalue: float, crit_vals: pandas.Series, null: str = 'No Cointegration', alternative: str = 'Cointegration', trend: str = 'c', order: int = 2, xsection: RegressionResults | None = None, test_type: str = 'Za', kernel_est: CovarianceEstimator | None = None, rho: float = 0.0*)

Methods

plot([axes, title])	Plot the cointegration residuals.
summary()	Summary of test, containing statistic, p-value and critical values

arch.unitroot.cointegration.PhillipsOuliarisTestResults.plot

PhillipsOuliarisTestResults.**plot**(axes: *Axes | None = None*, title: *str | None = None*) → Figure

Plot the cointegration residuals.

Parameters

axes: Axes | None = None
matplotlib axes instance to hold the figure.

title: str | None = None
Title for the figure.

Returns
The matplotlib Figure instance.

Return type
matplotlib.figure.Figure

arch.unitroot.cointegration.PhillipsOuliarisTestResults.summary

PhillipsOuliarisTestResults.**summary**() → Summary

Summary of test, containing statistic, p-value and critical values

Properties

`alternative_hypothesis`	The alternative hypothesis
`bandwidth`	Bandwidth used by the long-run covariance estimator
`cointegrating_vector`	The estimated cointegrating vector.
`critical_values`	Critical Values
`distribution_order`	The number of stochastic trends under the null hypothesis.
`kernel`	Name of the long-run covariance estimator
`name`	Sets or gets the name of the cointegration test
`null_hypothesis`	The null hypothesis
`pvalue`	The p-value of the test statistic.
`resid`	The residual from the cointegrating regression.
`stat`	The test statistic.
`trend`	The trend used in the cointegrating regression

arch.unitroot.cointegration.PhillipsOuliarisTestResults.alternative_hypothesis

property `PhillipsOuliarisTestResults.`**`alternative_hypothesis`** : str

The alternative hypothesis

arch.unitroot.cointegration.PhillipsOuliarisTestResults.bandwidth

property `PhillipsOuliarisTestResults.`**`bandwidth`** : float

Bandwidth used by the long-run covariance estimator

arch.unitroot.cointegration.PhillipsOuliarisTestResults.cointegrating_vector

property `PhillipsOuliarisTestResults.`**`cointegrating_vector`** : pandas.Series

The estimated cointegrating vector.

arch.unitroot.cointegration.PhillipsOuliarisTestResults.critical_values

property `PhillipsOuliarisTestResults.`**`critical_values`** : pandas.Series

Critical Values

Returns

Series with three keys, 1, 5 and 10 containing the critical values of the test statistic.

Return type

`pandas.Series`

arch.unitroot.cointegration.PhillipsOuliarisTestResults.distribution_order

property PhillipsOuliarisTestResults.**distribution_order** : int

The number of stochastic trends under the null hypothesis.

arch.unitroot.cointegration.PhillipsOuliarisTestResults.kernel

property PhillipsOuliarisTestResults.**kernel** : str

Name of the long-run covariance estimator

arch.unitroot.cointegration.PhillipsOuliarisTestResults.name

property PhillipsOuliarisTestResults.**name** : str

Sets or gets the name of the cointegration test

arch.unitroot.cointegration.PhillipsOuliarisTestResults.null_hypothesis

property PhillipsOuliarisTestResults.**null_hypothesis** : str

The null hypothesis

arch.unitroot.cointegration.PhillipsOuliarisTestResults.pvalue

property PhillipsOuliarisTestResults.**pvalue** : float

The p-value of the test statistic.

arch.unitroot.cointegration.PhillipsOuliarisTestResults.resid

property PhillipsOuliarisTestResults.**resid** : pandas.Series

The residual from the cointegrating regression.

arch.unitroot.cointegration.PhillipsOuliarisTestResults.stat

property PhillipsOuliarisTestResults.**stat** : float

The test statistic.

arch.unitroot.cointegration.PhillipsOuliarisTestResults.trend

property PhillipsOuliarisTestResults.**trend** : str

The trend used in the cointegrating regression

CHAPTER

SIX

LONG-RUN COVARIANCE ESTIMATION

6.1 Long-run Covariance Estimators

`Andrews`(x[, bandwidth, df_adjust, center, ...])	Alternative name of the QuadraticSpectral covariance estimator.
`Bartlett`(x[, bandwidth, df_adjust, center, ...])	Bartlett's (Newey-West) kernel covariance estimation.
`Gallant`(x[, bandwidth, df_adjust, center, ...])	Alternative name for Parzen covariance estimator.
`NeweyWest`(x[, bandwidth, df_adjust, center, ...])	Alternative name for Bartlett covariance estimator.
`Parzen`(x[, bandwidth, df_adjust, center, ...])	Parzen's kernel covariance estimation.
`ParzenCauchy`(x[, bandwidth, df_adjust, ...])	Parzen's Cauchy kernel covariance estimation.
`ParzenGeometric`(x[, bandwidth, df_adjust, ...])	Parzen's Geometric kernel covariance estimation.
`ParzenRiesz`(x[, bandwidth, df_adjust, ...])	Parzen-Reisz kernel covariance estimation.
`QuadraticSpectral`(x[, bandwidth, df_adjust, ...])	Quadratic-Spectral (Andrews') kernel covariance estimation.
`TukeyHamming`(x[, bandwidth, df_adjust, ...])	Tukey-Hamming kernel covariance estimation.
`TukeyHanning`(x[, bandwidth, df_adjust, ...])	Tukey-Hanning kernel covariance estimation.
`TukeyParzen`(x[, bandwidth, df_adjust, ...])	Tukey-Parzen kernel covariance estimation.

6.1.1 arch.covariance.kernel.Andrews

class arch.covariance.kernel.**Andrews**(*x: ndarray | DataFrame | Series*, *bandwidth: float | None = None*, *df_adjust: int = 0*, *center: bool = True*, *weights: ndarray | DataFrame | Series | None = None*, *force_int: bool = False*)

Alternative name of the QuadraticSpectral covariance estimator.

See also

`QuadraticSpectral`

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.Andrews.bandwidth

property Andrews.**bandwidth** : float

The bandwidth used by the covariance estimator.

Returns
The user-provided or estimated optimal bandwidth.

Return type
`float`

arch.covariance.kernel.Andrews.bandwidth_scale

property Andrews.**bandwidth_scale**

The power used in optimal bandwidth calculation.

Returns
The power value used in the optimal bandwidth calculation.

Return type
`float`

arch.covariance.kernel.Andrews.centered

property Andrews.**centered** : bool

Flag indicating whether the data are centered (demeaned).

Returns
A flag indicating whether the estimator is centered.

Return type
`bool`

arch.covariance.kernel.Andrews.cov

property Andrews.cov : *CovarianceEstimate*

The estimated covariances.

Returns

Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type

`CovarianceEstimate`

See also

`CovarianceEstimate`

arch.covariance.kernel.Andrews.force_int

property Andrews.force_int : bool

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.Andrews.kernel_const

property Andrews.kernel_const

The constant used in optimal bandwidth calculation.

Returns

The constant value used in the optimal bandwidth calculation.

Return type

`float`

arch.covariance.kernel.Andrews.kernel_weights

property Andrews.kernel_weights : ndarray

Weights used in covariance calculation.

Returns

The weight vector including 1 in position 0.

Return type

`numpy.ndarray`

arch.covariance.kernel.Andrews.name

property Andrews.**name** : str

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
`str`

arch.covariance.kernel.Andrews.opt_bandwidth

property Andrews.**opt_bandwidth** : float

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
`float`

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right) T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.Andrews.rate

property Andrews.**rate**

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
`float`

6.1.2 arch.covariance.kernel.Bartlett

class arch.covariance.kernel.**Bartlett**(x: *ndarray* | *DataFrame* | *Series*, bandwidth: *float* | *None* = **None**, df_adjust: *int* = **0**, center: *bool* = **True**, weights: *ndarray* | *DataFrame* | *Series* | *None* = **None**, force_int: *bool* = **False**)

Bartlett's (Newey-West) kernel covariance estimation.

Parameters

x: ndarray | DataFrame | Series
The data to use in covariance estimation.

bandwidth: float | None = None
The kernel's bandwidth. If None, optimal bandwidth is estimated.

df_adjust: int = 0
Degrees of freedom to remove when adjusting the covariance. Uses the number of observations in x minus df_adjust when dividing inner-products.

center: bool = True
A flag indicating whether x should be demeaned before estimating the covariance.

weights: ndarray | DataFrame | Series | None = None
An array of weights used to combine when estimating optimal bandwidth. If not provided, a vector of 1s is used. Must have nvar elements.

force_int: bool = False
Force bandwidth to be an integer.

Notes

The kernel weights are computed using

$$w = \begin{cases} 1 - |z| & z \leq 1 \\ 0 & z > 1 \end{cases}$$

where $z = \frac{h}{H}, h = 0, 1, \ldots, H$ where H is the bandwidth.

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.Bartlett.bandwidth

property `Bartlett.bandwidth` : float

The bandwidth used by the covariance estimator.

Returns
The user-provided or estimated optimal bandwidth.

Return type
`float`

arch.covariance.kernel.Bartlett.bandwidth_scale

property `Bartlett.bandwidth_scale`

The power used in optimal bandwidth calculation.

Returns
The power value used in the optimal bandwidth calculation.

Return type
`float`

arch.covariance.kernel.Bartlett.centered

property `Bartlett.centered` : bool

Flag indicating whether the data are centered (demeaned).

Returns
A flag indicating whether the estimator is centered.

Return type
`bool`

arch.covariance.kernel.Bartlett.cov

property `Bartlett.cov` : *CovarianceEstimate*

The estimated covariances.

Returns

Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type
`CovarianceEstimate`

See also

`CovarianceEstimate`

arch.covariance.kernel.Bartlett.force_int

property Bartlett.**force_int** : bool

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.Bartlett.kernel_const

property Bartlett.**kernel_const**

The constant used in optimal bandwidth calculation.

Returns
The constant value used in the optimal bandwidth calculation.

Return type
`float`

arch.covariance.kernel.Bartlett.kernel_weights

property Bartlett.**kernel_weights** : ndarray

Weights used in covariance calculation.

Returns
The weight vector including 1 in position 0.

Return type
`numpy.ndarray`

arch.covariance.kernel.Bartlett.name

property Bartlett.**name** : str

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
`str`

arch.covariance.kernel.Bartlett.opt_bandwidth

property Bartlett.**opt_bandwidth** : float

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
`float`

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right) T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.Bartlett.rate

`property Bartlett.rate`

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
`float`

6.1.3 arch.covariance.kernel.Gallant

class `arch.covariance.kernel.Gallant`(*x: ndarray | DataFrame | Series*, *bandwidth: float | None = None*, *df_adjust: int = 0*, *center: bool = True*, *weights: ndarray | DataFrame | Series | None = None*, *force_int: bool = False*)

Alternative name for Parzen covariance estimator.

See also

Parzen

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.Gallant.bandwidth

property Gallant.**bandwidth** : float

The bandwidth used by the covariance estimator.

Returns
The user-provided or estimated optimal bandwidth.

Return type
float

arch.covariance.kernel.Gallant.bandwidth_scale

property Gallant.**bandwidth_scale**

The power used in optimal bandwidth calculation.

Returns
The power value used in the optimal bandwidth calculation.

Return type
float

arch.covariance.kernel.Gallant.centered

property Gallant.**centered** : bool

Flag indicating whether the data are centered (demeaned).

Returns
A flag indicating whether the estimator is centered.

Return type
bool

arch.covariance.kernel.Gallant.cov

property Gallant.**cov** : *CovarianceEstimate*

The estimated covariances.

Returns

Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type

CovarianceEstimate

See also

CovarianceEstimate

---

arch.covariance.kernel.Gallant.force_int

property Gallant.**force_int** : bool

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.Gallant.kernel_const

property Gallant.**kernel_const**

The constant used in optimal bandwidth calculation.

Returns

The constant value used in the optimal bandwidth calculation.

Return type

float

arch.covariance.kernel.Gallant.kernel_weights

property Gallant.**kernel_weights** : ndarray

Weights used in covariance calculation.

Returns

The weight vector including 1 in position 0.

Return type

numpy.ndarray

arch.covariance.kernel.Gallant.name

property Gallant.**name** : str

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
str

arch.covariance.kernel.Gallant.opt_bandwidth

property Gallant.**opt_bandwidth** : float

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
float

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right)T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.Gallant.rate

property Gallant.**rate**

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
float

6.1.4 arch.covariance.kernel.NeweyWest

class arch.covariance.kernel.**NeweyWest**(*x: ndarray | DataFrame | Series*, *bandwidth: float | None = None*, *df_adjust: int = 0*, *center: bool = True*, *weights: ndarray | DataFrame | Series | None = None*, *force_int: bool = False*)

Alternative name for Bartlett covariance estimator.

See also

Bartlett

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.NeweyWest.bandwidth

property NeweyWest.**bandwidth** : float

The bandwidth used by the covariance estimator.

Returns
The user-provided or estimated optimal bandwidth.

Return type
float

arch.covariance.kernel.NeweyWest.bandwidth_scale

property NeweyWest.**bandwidth_scale**

The power used in optimal bandwidth calculation.

Returns
The power value used in the optimal bandwidth calculation.

Return type
float

arch.covariance.kernel.NeweyWest.centered

property NeweyWest.**centered** : bool

Flag indicating whether the data are centered (demeaned).

Returns
A flag indicating whether the estimator is centered.

Return type
bool

arch.covariance.kernel.NeweyWest.cov

property NeweyWest.**cov** : *CovarianceEstimate*

The estimated covariances.

Returns

Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type
CovarianceEstimate

See also

CovarianceEstimate

arch.covariance.kernel.NeweyWest.force_int

property NeweyWest.**force_int** : bool

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.NeweyWest.kernel_const

property NeweyWest.**kernel_const**

The constant used in optimal bandwidth calculation.

Returns
The constant value used in the optimal bandwidth calculation.

Return type
float

arch.covariance.kernel.NeweyWest.kernel_weights

property NeweyWest.**kernel_weights** : ndarray

Weights used in covariance calculation.

Returns
The weight vector including 1 in position 0.

Return type
numpy.ndarray

arch.covariance.kernel.NeweyWest.name

property NeweyWest.**name** : str

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
str

arch.covariance.kernel.NeweyWest.opt_bandwidth

property NeweyWest.**opt_bandwidth** : float

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
float

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right) T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.NeweyWest.rate

property NeweyWest.**rate**

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
float

6.1.5 arch.covariance.kernel.Parzen

class arch.covariance.kernel.**Parzen**(*x: ndarray | DataFrame | Series*, *bandwidth: float | None = None*, *df_adjust: int = 0*, *center: bool = True*, *weights: ndarray | DataFrame | Series | None = None*, *force_int: bool = False*)

Parzen's kernel covariance estimation.

Parameters

x: ndarray | DataFrame | Series
The data to use in covariance estimation.

bandwidth: float | None = None
The kernel's bandwidth. If None, optimal bandwidth is estimated.

df_adjust: int = 0
Degrees of freedom to remove when adjusting the covariance. Uses the number of observations in x minus df_adjust when dividing inner-products.

center: bool = True
A flag indicating whether x should be demeaned before estimating the covariance.

weights: ndarray | DataFrame | Series | None = None
An array of weights used to combine when estimating optimal bandwidth. If not provided, a vector of 1s is used. Must have nvar elements.

force_int: bool = False
Force bandwidth to be an integer.

Notes

The kernel weights are computed using

$$w = \begin{cases} 1 - 6z^2\left(1-z\right) & z \leq \frac{1}{2} \\ 2\left(1-z\right)^3 & \frac{1}{2} < z \leq 1 \\ 0 & z > 1 \end{cases}$$

where $z = \frac{h}{H}, h = 0, 1, \ldots, H$ where H is the bandwidth.

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.Parzen.bandwidth

property Parzen.**bandwidth** : float

The bandwidth used by the covariance estimator.

Returns
The user-provided or estimated optimal bandwidth.

Return type
float

arch.covariance.kernel.Parzen.bandwidth_scale

property Parzen.**bandwidth_scale**

The power used in optimal bandwidth calculation.

Returns
The power value used in the optimal bandwidth calculation.

Return type
float

arch.covariance.kernel.Parzen.centered

property Parzen.**centered** : bool

Flag indicating whether the data are centered (demeaned).

Returns
A flag indicating whether the estimator is centered.

Return type
bool

arch.covariance.kernel.Parzen.cov

property Parzen.**cov** : *CovarianceEstimate*

The estimated covariances.

Returns

Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type

`CovarianceEstimate`

See also

`CovarianceEstimate`

---

arch.covariance.kernel.Parzen.force_int

property Parzen.**force_int** : bool

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.Parzen.kernel_const

property Parzen.**kernel_const**

The constant used in optimal bandwidth calculation.

Returns

The constant value used in the optimal bandwidth calculation.

Return type

`float`

arch.covariance.kernel.Parzen.kernel_weights

property Parzen.**kernel_weights** : ndarray

Weights used in covariance calculation.

Returns

The weight vector including 1 in position 0.

Return type

`numpy.ndarray`

arch.covariance.kernel.Parzen.name

property Parzen.**name** : str

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
str

arch.covariance.kernel.Parzen.opt_bandwidth

property Parzen.**opt_bandwidth** : float

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
float

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right) T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.Parzen.rate

property Parzen.**rate**

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
float

6.1.6 arch.covariance.kernel.ParzenCauchy

class arch.covariance.kernel.**ParzenCauchy**(x: *ndarray* | *DataFrame* | *Series*, bandwidth: *float* | *None* = ***None***, df_adjust: *int* = **0**, center: *bool* = ***True***, weights: *ndarray* | *DataFrame* | *Series* | *None* = ***None***, force_int: *bool* = ***False***)

Parzen's Cauchy kernel covariance estimation.

Parameters

x: ndarray | DataFrame | Series
The data to use in covariance estimation.

bandwidth: float | None = None
The kernel's bandwidth. If None, optimal bandwidth is estimated.

df_adjust: int = 0
Degrees of freedom to remove when adjusting the covariance. Uses the number of observations in x minus df_adjust when dividing inner-products.

center: bool = True
A flag indicating whether x should be demeaned before estimating the covariance.

weights: ndarray | DataFrame | Series | None = None
An array of weights used to combine when estimating optimal bandwidth. If not provided, a vector of 1s is used. Must have nvar elements.

force_int: bool = False
Force bandwidth to be an integer.

Notes

The kernel weights are computed using

$$w = \begin{cases} \frac{1}{1+z^2} & z \leq 1 \\ 0 & z > 1 \end{cases}$$

where $z = \frac{h}{H}, h = 0, 1, \ldots, H$ where H is the bandwidth.

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.ParzenCauchy.bandwidth

property ParzenCauchy.**bandwidth** : float

The bandwidth used by the covariance estimator.

Returns
The user-provided or estimated optimal bandwidth.

Return type
`float`

arch.covariance.kernel.ParzenCauchy.bandwidth_scale

property ParzenCauchy.**bandwidth_scale**

The power used in optimal bandwidth calculation.

Returns
The power value used in the optimal bandwidth calculation.

Return type
`float`

arch.covariance.kernel.ParzenCauchy.centered

property ParzenCauchy.**centered** : bool

Flag indicating whether the data are centered (demeaned).

Returns
A flag indicating whether the estimator is centered.

Return type
`bool`

arch.covariance.kernel.ParzenCauchy.cov

property ParzenCauchy.**cov** : *CovarianceEstimate*

The estimated covariances.

Returns

Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type
CovarianceEstimate

See also

CovarianceEstimate

arch.covariance.kernel.ParzenCauchy.force_int

`property ParzenCauchy.force_int : bool`

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.ParzenCauchy.kernel_const

`property ParzenCauchy.kernel_const`

The constant used in optimal bandwidth calculation.

Returns
The constant value used in the optimal bandwidth calculation.

Return type
`float`

arch.covariance.kernel.ParzenCauchy.kernel_weights

`property ParzenCauchy.kernel_weights : ndarray`

Weights used in covariance calculation.

Returns
The weight vector including 1 in position 0.

Return type
`numpy.ndarray`

arch.covariance.kernel.ParzenCauchy.name

`property ParzenCauchy.name : str`

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
`str`

arch.covariance.kernel.ParzenCauchy.opt_bandwidth

`property ParzenCauchy.opt_bandwidth : float`

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
`float`

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right) T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.ParzenCauchy.rate

property `ParzenCauchy.rate`

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
`float`

6.1.7 arch.covariance.kernel.ParzenGeometric

class `arch.covariance.kernel.ParzenGeometric`(x: *ndarray* | *DataFrame* | *Series*, bandwidth: *float* | *None* = ***None***, df_adjust: *int* = ***0***, center: *bool* = ***True***, weights: *ndarray* | *DataFrame* | *Series* | *None* = ***None***, force_int: *bool* = ***False***)

Parzen's Geometric kernel covariance estimation.

Parameters

x: ndarray | DataFrame | Series
The data to use in covariance estimation.

bandwidth: float | None = None
The kernel's bandwidth. If None, optimal bandwidth is estimated.

df_adjust: int = 0
Degrees of freedom to remove when adjusting the covariance. Uses the number of observations in x minus df_adjust when dividing inner-products.

center: bool = True
A flag indicating whether x should be demeaned before estimating the covariance.

weights: ndarray | DataFrame | Series | None = None
An array of weights used to combine when estimating optimal bandwidth. If not provided, a vector of 1s is used. Must have nvar elements.

force_int: bool = False
Force bandwidth to be an integer.

Notes

The kernel weights are computed using

$$w=\begin{cases}\frac{1}{1+z} & z\leq 1\\ 0 & z>1\end{cases}$$

where $z=\frac{h}{H}, h=0,1,\ldots,H$ where H is the bandwidth.

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.ParzenGeometric.bandwidth

property ParzenGeometric.**bandwidth** : float

The bandwidth used by the covariance estimator.

Returns

The user-provided or estimated optimal bandwidth.

Return type

`float`

arch.covariance.kernel.ParzenGeometric.bandwidth_scale

property ParzenGeometric.**bandwidth_scale**

The power used in optimal bandwidth calculation.

Returns

The power value used in the optimal bandwidth calculation.

Return type

`float`

arch.covariance.kernel.ParzenGeometric.centered

property `ParzenGeometric.centered` : bool

Flag indicating whether the data are centered (demeaned).

Returns
A flag indicating whether the estimator is centered.

Return type
`bool`

arch.covariance.kernel.ParzenGeometric.cov

property `ParzenGeometric.cov` : *CovarianceEstimate*

The estimated covariances.

Returns

Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type
`CovarianceEstimate`

See also

`CovarianceEstimate`

arch.covariance.kernel.ParzenGeometric.force_int

property `ParzenGeometric.force_int` : bool

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.ParzenGeometric.kernel_const

property `ParzenGeometric.kernel_const`

The constant used in optimal bandwidth calculation.

Returns
The constant value used in the optimal bandwidth calculation.

Return type
`float`

arch.covariance.kernel.ParzenGeometric.kernel_weights

property `ParzenGeometric.kernel_weights` : ndarray

Weights used in covariance calculation.

Returns
The weight vector including 1 in position 0.

Return type
`numpy.ndarray`

arch.covariance.kernel.ParzenGeometric.name

property `ParzenGeometric.name` : str

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
`str`

arch.covariance.kernel.ParzenGeometric.opt_bandwidth

property `ParzenGeometric.opt_bandwidth` : float

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
`float`

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right) T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.ParzenGeometric.rate

property `ParzenGeometric.rate`

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
`float`

6.1.8 arch.covariance.kernel.ParzenRiesz

class arch.covariance.kernel.**ParzenRiesz**(x: *ndarray* | *DataFrame* | *Series*, bandwidth: *float* | *None* = ***None***, df_adjust: *int* = ***0***, center: *bool* = ***True***, weights: *ndarray* | *DataFrame* | *Series* | *None* = ***None***, force_int: *bool* = ***False***)

Parzen-Reisz kernel covariance estimation.

Parameters

x: ndarray | DataFrame | Series
The data to use in covariance estimation.

bandwidth: float | None = None
The kernel's bandwidth. If None, optimal bandwidth is estimated.

df_adjust: int = 0
Degrees of freedom to remove when adjusting the covariance. Uses the number of observations in x minus df_adjust when dividing inner-products.

center: bool = True
A flag indicating whether x should be demeaned before estimating the covariance.

weights: ndarray | DataFrame | Series | None = None
An array of weights used to combine when estimating optimal bandwidth. If not provided, a vector of 1s is used. Must have nvar elements.

force_int: bool = False
Force bandwidth to be an integer.

Notes

The kernel weights are computed using

$$w = \begin{cases} 1 - z^2 & z \leq 1 \\ 0 & z > 1 \end{cases}$$

where $z = \frac{h}{H}, h = 0, 1, \ldots, H$ where H is the bandwidth.

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.ParzenRiesz.bandwidth

property ParzenRiesz.**bandwidth** : float

The bandwidth used by the covariance estimator.

Returns

The user-provided or estimated optimal bandwidth.

Return type

float

arch.covariance.kernel.ParzenRiesz.bandwidth_scale

property ParzenRiesz.**bandwidth_scale**

The power used in optimal bandwidth calculation.

Returns

The power value used in the optimal bandwidth calculation.

Return type

float

arch.covariance.kernel.ParzenRiesz.centered

property ParzenRiesz.**centered** : bool

Flag indicating whether the data are centered (demeaned).

Returns

A flag indicating whether the estimator is centered.

Return type

bool

arch.covariance.kernel.ParzenRiesz.cov

property ParzenRiesz.**cov** : *CovarianceEstimate*

The estimated covariances.

Returns

Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type

`CovarianceEstimate`

See also

`CovarianceEstimate`

arch.covariance.kernel.ParzenRiesz.force_int

property ParzenRiesz.**force_int** : bool

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.ParzenRiesz.kernel_const

property ParzenRiesz.**kernel_const**

The constant used in optimal bandwidth calculation.

Returns

The constant value used in the optimal bandwidth calculation.

Return type

`float`

arch.covariance.kernel.ParzenRiesz.kernel_weights

property ParzenRiesz.**kernel_weights** : ndarray

Weights used in covariance calculation.

Returns

The weight vector including 1 in position 0.

Return type

`numpy.ndarray`

arch.covariance.kernel.ParzenRiesz.name

property ParzenRiesz.**name** : str

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
`str`

arch.covariance.kernel.ParzenRiesz.opt_bandwidth

property ParzenRiesz.**opt_bandwidth** : float

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
`float`

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right) T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.ParzenRiesz.rate

property ParzenRiesz.**rate**

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
`float`

6.1.9 arch.covariance.kernel.QuadraticSpectral

class arch.covariance.kernel.**QuadraticSpectral**(x: *ndarray* | *DataFrame* | *Series*, bandwidth: *float* | *None* = **None**, df_adjust: *int* = **0**, center: *bool* = **True**, weights: *ndarray* | *DataFrame* | *Series* | *None* = **None**, force_int: *bool* = **False**)

Quadratic-Spectral (Andrews') kernel covariance estimation.

Parameters

x: ndarray | DataFrame | Series
The data to use in covariance estimation.

bandwidth: float | None = None
The kernel's bandwidth. If None, optimal bandwidth is estimated.

df_adjust: int = 0
Degrees of freedom to remove when adjusting the covariance. Uses the number of observations in x minus df_adjust when dividing inner-products.

center: bool = True
A flag indicating whether x should be demeaned before estimating the covariance.

weights: ndarray | DataFrame | Series | None = None
An array of weights used to combine when estimating optimal bandwidth. If not provided, a vector of 1s is used. Must have nvar elements.

force_int: bool = False
Force bandwidth to be an integer.

Notes

The kernel weights are computed using

$$w=\begin{cases}1 & z=0\\ \frac{3}{x^2}\left(\frac{\sin x}{x}-\cos x\right), x=\frac{6\pi z}{5} & z>0\end{cases}$$

where $z=\frac{h}{H}, h=0,1,\ldots,H$ where H is the bandwidth.

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.QuadraticSpectral.bandwidth

property `QuadraticSpectral.bandwidth` : float

The bandwidth used by the covariance estimator.

Returns
The user-provided or estimated optimal bandwidth.

Return type
`float`

arch.covariance.kernel.QuadraticSpectral.bandwidth_scale

property `QuadraticSpectral.bandwidth_scale`

The power used in optimal bandwidth calculation.

Returns
The power value used in the optimal bandwidth calculation.

Return type
`float`

arch.covariance.kernel.QuadraticSpectral.centered

property `QuadraticSpectral.centered` : bool

Flag indicating whether the data are centered (demeaned).

Returns
A flag indicating whether the estimator is centered.

Return type
`bool`

arch.covariance.kernel.QuadraticSpectral.cov

property `QuadraticSpectral.cov` : *CovarianceEstimate*

The estimated covariances.

Returns
Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type
`CovarianceEstimate`

See also

`CovarianceEstimate`

arch.covariance.kernel.QuadraticSpectral.force_int

property `QuadraticSpectral.force_int` : bool

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.QuadraticSpectral.kernel_const

property `QuadraticSpectral.kernel_const`

The constant used in optimal bandwidth calculation.

Returns
The constant value used in the optimal bandwidth calculation.

Return type
`float`

arch.covariance.kernel.QuadraticSpectral.kernel_weights

property `QuadraticSpectral.kernel_weights` : ndarray

Weights used in covariance calculation.

Returns
The weight vector including 1 in position 0.

Return type
`numpy.ndarray`

arch.covariance.kernel.QuadraticSpectral.name

property `QuadraticSpectral.name` : str

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
`str`

arch.covariance.kernel.QuadraticSpectral.opt_bandwidth

property `QuadraticSpectral.opt_bandwidth` : float

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
`float`

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right) T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.QuadraticSpectral.rate

property QuadraticSpectral.rate

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
`float`

6.1.10 arch.covariance.kernel.TukeyHamming

class arch.covariance.kernel.TukeyHamming(x: *ndarray* | *DataFrame* | *Series*, bandwidth: *float* | *None* = **None**, df_adjust: *int* = **0**, center: *bool* = **True**, weights: *ndarray* | *DataFrame* | *Series* | *None* = **None**, force_int: *bool* = ***False***)

Tukey-Hamming kernel covariance estimation.

Parameters

x: ndarray | DataFrame | Series
The data to use in covariance estimation.

bandwidth: float | None = None
The kernel's bandwidth. If None, optimal bandwidth is estimated.

df_adjust: int = 0
Degrees of freedom to remove when adjusting the covariance. Uses the number of observations in x minus df_adjust when dividing inner-products.

center: bool = True
A flag indicating whether x should be demeaned before estimating the covariance.

weights: ndarray | DataFrame | Series | None = None
An array of weights used to combine when estimating optimal bandwidth. If not provided, a vector of 1s is used. Must have nvar elements.

force_int: bool = False
Force bandwidth to be an integer.

Notes

The kernel weights are computed using

$$w=\begin{cases}0.54+0.46\cos\pi z & z\leq 1\\ 0 & z>1\end{cases}$$

where $z=\frac{h}{H}, h=0,1,\ldots,H$ where H is the bandwidth.

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.TukeyHamming.bandwidth

property `TukeyHamming.`**`bandwidth`** : float

The bandwidth used by the covariance estimator.

Returns

The user-provided or estimated optimal bandwidth.

Return type

`float`

arch.covariance.kernel.TukeyHamming.bandwidth_scale

property `TukeyHamming.`**`bandwidth_scale`**

The power used in optimal bandwidth calculation.

Returns

The power value used in the optimal bandwidth calculation.

Return type

`float`

arch.covariance.kernel.TukeyHamming.centered

property `TukeyHamming.centered` : bool

Flag indicating whether the data are centered (demeaned).

Returns
A flag indicating whether the estimator is centered.

Return type
`bool`

arch.covariance.kernel.TukeyHamming.cov

property `TukeyHamming.cov` : *CovarianceEstimate*

The estimated covariances.

Returns
Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type
`CovarianceEstimate`

See also

`CovarianceEstimate`

arch.covariance.kernel.TukeyHamming.force_int

property `TukeyHamming.force_int` : bool

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.TukeyHamming.kernel_const

property `TukeyHamming.kernel_const`

The constant used in optimal bandwidth calculation.

Returns
The constant value used in the optimal bandwidth calculation.

Return type
`float`

arch.covariance.kernel.TukeyHamming.kernel_weights

property TukeyHamming.**kernel_weights** : ndarray

Weights used in covariance calculation.

Returns
The weight vector including 1 in position 0.

Return type
`numpy.ndarray`

arch.covariance.kernel.TukeyHamming.name

property TukeyHamming.**name** : str

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
`str`

arch.covariance.kernel.TukeyHamming.opt_bandwidth

property TukeyHamming.**opt_bandwidth** : float

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
`float`

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right) T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.TukeyHamming.rate

property TukeyHamming.**rate**

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
`float`

6.1.11 arch.covariance.kernel.TukeyHanning

class arch.covariance.kernel.**TukeyHanning**(x: *ndarray | DataFrame | Series*, bandwidth: *float | None* = **None**, df_adjust: *int* = **0**, center: *bool* = **True**, weights: *ndarray | DataFrame | Series | None* = **None**, force_int: *bool* = **False**)

Tukey-Hanning kernel covariance estimation.

Parameters

x: ndarray | DataFrame | Series
The data to use in covariance estimation.

bandwidth: float | None = None
The kernel's bandwidth. If None, optimal bandwidth is estimated.

df_adjust: int = 0
Degrees of freedom to remove when adjusting the covariance. Uses the number of observations in x minus df_adjust when dividing inner-products.

center: bool = True
A flag indicating whether x should be demeaned before estimating the covariance.

weights: ndarray | DataFrame | Series | None = None
An array of weights used to combine when estimating optimal bandwidth. If not provided, a vector of 1s is used. Must have nvar elements.

force_int: bool = False
Force bandwidth to be an integer.

Notes

The kernel weights are computed using

$$w = \begin{cases} \frac{1}{2} + \frac{1}{2}\cos\pi z & z \leq 1 \\ 0 & z > 1 \end{cases}$$

where $z = \frac{h}{H}, h = 0, 1, \ldots, H$ where H is the bandwidth.

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.TukeyHanning.bandwidth

property TukeyHanning.**bandwidth** : float

The bandwidth used by the covariance estimator.

Returns
The user-provided or estimated optimal bandwidth.

Return type
`float`

arch.covariance.kernel.TukeyHanning.bandwidth_scale

property TukeyHanning.**bandwidth_scale**

The power used in optimal bandwidth calculation.

Returns
The power value used in the optimal bandwidth calculation.

Return type
`float`

arch.covariance.kernel.TukeyHanning.centered

property TukeyHanning.**centered** : bool

Flag indicating whether the data are centered (demeaned).

Returns
A flag indicating whether the estimator is centered.

Return type
`bool`

arch.covariance.kernel.TukeyHanning.cov

property TukeyHanning.cov : *CovarianceEstimate*

The estimated covariances.

Returns

Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type

`CovarianceEstimate`

See also

`CovarianceEstimate`

arch.covariance.kernel.TukeyHanning.force_int

property TukeyHanning.force_int : bool

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.TukeyHanning.kernel_const

property TukeyHanning.kernel_const

The constant used in optimal bandwidth calculation.

Returns

The constant value used in the optimal bandwidth calculation.

Return type

`float`

arch.covariance.kernel.TukeyHanning.kernel_weights

property TukeyHanning.kernel_weights : ndarray

Weights used in covariance calculation.

Returns

The weight vector including 1 in position 0.

Return type

`numpy.ndarray`

arch.covariance.kernel.TukeyHanning.name

property `TukeyHanning.name` : str

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
`str`

arch.covariance.kernel.TukeyHanning.opt_bandwidth

property `TukeyHanning.opt_bandwidth` : float

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
`float`

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right) T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.TukeyHanning.rate

property `TukeyHanning.rate`

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
`float`

6.1.12 arch.covariance.kernel.TukeyParzen

class `arch.covariance.kernel.TukeyParzen`(x: *ndarray* | *DataFrame* | *Series*, bandwidth: *float* | *None* = **None**, df_adjust: *int* = **0**, center: *bool* = **True**, weights: *ndarray* | *DataFrame* | *Series* | *None* = **None**, force_int: *bool* = **False**)

Tukey-Parzen kernel covariance estimation.

Parameters

x: ndarray | DataFrame | Series
The data to use in covariance estimation.

bandwidth: float | None = None
The kernel's bandwidth. If None, optimal bandwidth is estimated.

df_adjust: int = 0
Degrees of freedom to remove when adjusting the covariance. Uses the number of observations in x minus df_adjust when dividing inner-products.

center: bool = True
A flag indicating whether x should be demeaned before estimating the covariance.

weights: ndarray | DataFrame | Series | None = None
An array of weights used to combine when estimating optimal bandwidth. If not provided, a vector of 1s is used. Must have nvar elements.

force_int: bool = False
Force bandwidth to be an integer.

Notes

The kernel weights are computed using

$$w=\begin{cases}0.436+0.564\cos\pi z & z\leq 1\\ 0 & z>1\end{cases}$$

where $z=\frac{h}{H}, h=0,1,\ldots,H$ where H is the bandwidth.

Methods

Properties

bandwidth	The bandwidth used by the covariance estimator.
bandwidth_scale	The power used in optimal bandwidth calculation.
centered	Flag indicating whether the data are centered (demeaned).
cov	The estimated covariances.
force_int	Flag indicating whether the bandwidth is restricted to be an integer.
kernel_const	The constant used in optimal bandwidth calculation.
kernel_weights	Weights used in covariance calculation.
name	The covariance estimator's name.
opt_bandwidth	Estimate optimal bandwidth.
rate	The optimal rate used in bandwidth selection.

arch.covariance.kernel.TukeyParzen.bandwidth

property `TukeyParzen.bandwidth` : float

The bandwidth used by the covariance estimator.

Returns
The user-provided or estimated optimal bandwidth.

Return type
`float`

arch.covariance.kernel.TukeyParzen.bandwidth_scale

property `TukeyParzen.bandwidth_scale`

The power used in optimal bandwidth calculation.

Returns
The power value used in the optimal bandwidth calculation.

Return type
`float`

arch.covariance.kernel.TukeyParzen.centered

property `TukeyParzen.centered` : bool

Flag indicating whether the data are centered (demeaned).

Returns
A flag indicating whether the estimator is centered.

Return type
`bool`

arch.covariance.kernel.TukeyParzen.cov

property `TukeyParzen.cov` : *CovarianceEstimate*

The estimated covariances.

Returns
Covariance estimate instance containing 4 estimates:

- long_run
- short_run
- one_sided
- one_sided_strict

Return type
`CovarianceEstimate`

See also

`CovarianceEstimate`

arch.covariance.kernel.TukeyParzen.force_int

property TukeyParzen.**force_int** : bool

Flag indicating whether the bandwidth is restricted to be an integer.

arch.covariance.kernel.TukeyParzen.kernel_const

property TukeyParzen.**kernel_const**

The constant used in optimal bandwidth calculation.

Returns
The constant value used in the optimal bandwidth calculation.

Return type
float

arch.covariance.kernel.TukeyParzen.kernel_weights

property TukeyParzen.**kernel_weights** : ndarray

Weights used in covariance calculation.

Returns
The weight vector including 1 in position 0.

Return type
numpy.ndarray

arch.covariance.kernel.TukeyParzen.name

property TukeyParzen.**name** : str

The covariance estimator's name.

Returns
The covariance estimator's name.

Return type
str

arch.covariance.kernel.TukeyParzen.opt_bandwidth

property TukeyParzen.**opt_bandwidth** : float

Estimate optimal bandwidth.

Returns
The estimated optimal bandwidth.

Return type
float

Notes

Computed as

$$\hat{b}_T = c_k \left[\hat{\alpha}\left(q\right) T\right]^{\frac{1}{2q+1}}$$

where c_k is a kernel-dependent constant, T is the sample size, q determines the optimal bandwidth rate for the kernel.

arch.covariance.kernel.TukeyParzen.rate

property TukeyParzen.**rate**

The optimal rate used in bandwidth selection.

Controls the number of lags used in the variance estimate that determines the estimate of the optimal bandwidth.

Returns
The rate used in bandwidth selection.

Return type
`float`

6.2 Results

CovarianceEstimate(short_run, one_sided_strict)	Covariance estimate using a long-run covariance estimator

6.2.1 arch.covariance.kernel.CovarianceEstimate

class arch.covariance.kernel.**CovarianceEstimate**(short_run: *ndarray*, one_sided_strict: *ndarray*, columns: *Index | list[str] | None* = ***None***, long_run: *ndarray | None* = ***None***, *one_sided: ndarray | None* = ***None***)

Covariance estimate using a long-run covariance estimator

Parameters

short_run: ndarray
The short-run covariance estimate.

one_sided_strict: ndarray
The one-sided strict covariance estimate.

columns: Index | list[str] | None = None
Column labels to use if covariance estimates are returned as DataFrames.

long_run: ndarray | None = None
The long-run covariance estimate. If not provided, computed from short_run and one_sided_strict.

one_sided_strict: ndarray
The one-sided-strict covariance estimate. If not provided, computed from short_run and one_sided_strict.

Notes

If Γ_0 is the short-run covariance and Λ_1 is the one-sided strict covariance, then the long-run covariance is defined

$$\Omega = \Gamma_0 + \Lambda_1 + \Lambda_1'$$

and the one-sided covariance is

$$\Lambda_0 = \Gamma_0 + \Lambda_1.$$

Methods

Properties

long_run	The long-run covariance estimate.
one_sided	The one-sided covariance estimate.
one_sided_strict	The one-sided strict covariance estimate.
short_run	The short-run covariance estimate.

arch.covariance.kernel.CovarianceEstimate.long_run

property `CovarianceEstimate.`**`long_run`** : Float64Array | DataFrame

The long-run covariance estimate.

arch.covariance.kernel.CovarianceEstimate.one_sided

property `CovarianceEstimate.`**`one_sided`** : Float64Array | DataFrame

The one-sided covariance estimate.

arch.covariance.kernel.CovarianceEstimate.one_sided_strict

property `CovarianceEstimate.`**`one_sided_strict`** : Float64Array | DataFrame

The one-sided strict covariance estimate.

arch.covariance.kernel.CovarianceEstimate.short_run

property `CovarianceEstimate.`**`short_run`** : Float64Array | DataFrame

The short-run covariance estimate.

CHAPTER

SEVEN

API REFERENCE

This page lists contains a list of the essential end-user API functions and classes.

7.1 Volatility Modeling

7.1.1 High-level

arch_model(y[, x, mean, lags, vol, p, o, q, ...])	Initialization of common ARCH model specifications

7.1.2 Mean Specification

ConstantMean([y, hold_back, volatility, ...])	Constant mean model estimation and simulation.
ZeroMean([y, hold_back, volatility, ...])	Model with zero conditional mean estimation and simulation
HARX([y, x, lags, constant, use_rotated, ...])	Heterogeneous Autoregression (HAR), with optional exogenous regressors, model estimation and simulation
ARX([y, x, lags, constant, hold_back, ...])	Autoregressive model with optional exogenous regressors estimation and simulation
LS([y, x, constant, hold_back, volatility, ...])	Least squares model estimation and simulation

7.1.3 Volatility Process Specification

GARCH([p, o, q, power])	GARCH and related model estimation
EGARCH([p, o, q])	EGARCH model estimation
HARCH([lags])	Heterogeneous ARCH process
FIGARCH([p, q, power, truncation])	FIGARCH model
MIDASHyperbolic([m, asym])	MIDAS Hyperbolic ARCH process
EWMAVariance([lam])	Exponentially Weighted Moving-Average (RiskMetrics) Variance process
RiskMetrics2006([tau0, tau1, kmax, rho])	RiskMetrics 2006 Variance process
ConstantVariance()	Constant volatility process
FixedVariance(variance[, unit_scale])	Fixed volatility process

7.1.4 Shock Distributions

Normal([random_state, seed])	Standard normal distribution for use with ARCH models
StudentsT([random_state, seed])	Standardized Student's distribution for use with ARCH models
SkewStudent([random_state, seed])	Standardized Skewed Student's distribution for use with ARCH models
GeneralizedError([random_state, seed])	Generalized Error distribution for use with ARCH models

7.2 Unit Root Testing

ADF(y[, lags, trend, max_lags, method, ...])	Augmented Dickey-Fuller unit root test
DFGLS(y[, lags, trend, max_lags, method, ...])	Elliott, Rothenberg and Stock's ([1]_) GLS detrended Dickey-Fuller
PhillipsPerron(y[, lags, trend, test_type])	Phillips-Perron unit root test
ZivotAndrews(y[, lags, trend, trim, ...])	Zivot-Andrews structural-break unit-root test
VarianceRatio(y[, lags, trend, debiased, ...])	Variance Ratio test of a random walk.
KPSS(y[, lags, trend])	Kwiatkowski, Phillips, Schmidt and Shin (KPSS) stationarity test

7.3 Cointegration Testing

engle_granger(y, x[, trend, lags, max_lags, ...])	Test for cointegration within a set of time series.
phillips_ouliaris(y, x[, trend, test_type, ...])	Test for cointegration within a set of time series.

7.4 Cointegrating Relationship Estimation

CanonicalCointegratingReg(y, x[, trend, x_trend])	Canonical Cointegrating Regression cointegrating vector estimation.
DynamicOLS(y, x[, trend, lags, leads, ...])	Dynamic OLS (DOLS) cointegrating vector estimation
FullyModifiedOLS(y, x[, trend, x_trend])	Fully Modified OLS cointegrating vector estimation.

7.5 Bootstraps

IIDBootstrap(*args[, random_state, seed])	Bootstrap using uniform resampling
IndependentSamplesBootstrap(*args[, ...])	Bootstrap where each input is independently resampled
StationaryBootstrap(block_size, *args[, ...])	Politis and Romano (1994) bootstrap with expon distributed block sizes
CircularBlockBootstrap(block_size, *args[, ...])	Bootstrap using blocks of the same length with end-to-start wrap around
MovingBlockBootstrap(block_size, *args[, ...])	Bootstrap using blocks of the same length without wrap around

7.5.1 Block-length Selection

optimal_block_length(x)	Estimate optimal window length for time-series bootstraps

7.6 Testing with Multiple-Comparison

SPA(benchmark, models[, block_size, reps, ...])	Test of Superior Predictive Ability (SPA) of White and Hansen.
MCS(losses, size[, reps, block_size, ...])	Model Confidence Set (MCS) of Hansen, Lunde and Nason.
StepM(benchmark, models[, size, block_size, ...])	StepM multiple comparison procedure of Romano and Wolf.

7.7 Long-run Covariance (HAC) Estimation

Bartlett(x[, bandwidth, df_adjust, center, ...])	Bartlett's (Newey-West) kernel covariance estimation.
Parzen(x[, bandwidth, df_adjust, center, ...])	Parzen's kernel covariance estimation.
ParzenCauchy(x[, bandwidth, df_adjust, ...])	Parzen's Cauchy kernel covariance estimation.
ParzenGeometric(x[, bandwidth, df_adjust, ...])	Parzen's Geometric kernel covariance estimation.
ParzenRiesz(x[, bandwidth, df_adjust, ...])	Parzen-Reisz kernel covariance estimation.
QuadraticSpectral(x[, bandwidth, df_adjust, ...])	Quadratic-Spectral (Andrews') kernel covariance estimation.
TukeyHamming(x[, bandwidth, df_adjust, ...])	Tukey-Hamming kernel covariance estimation.
TukeyHanning(x[, bandwidth, df_adjust, ...])	Tukey-Hanning kernel covariance estimation.
TukeyParzen(x[, bandwidth, df_adjust, ...])	Tukey-Parzen kernel covariance estimation.

CHAPTER

EIGHT

CHANGE LOGS

8.1 Version 6

8.1.1 Release 6.3

- Performance enhancement for long-run covariance estimators when numba is installed (GH687)
- Python 3.12 support
- Compatability with NumPy 2
- Fixes for future changes in pandas

8.1.2 Release 6.2

- Fixed a bug that affected forecasting from `FIGARCH` models (GH606).
- Added a performance warning when testing for unit roots in large series using a lag-length search with no-max-lag specified.
- Fixed a bug that affected forecasting from `arch.univariate.volatility.FIGARCH` models (GH606).
- Changed the default value of `reindex` to `False` so that forecasts will not match the input by default. Set `reindex` to `True` if this is required.
- Made `from __future__ import reindex` a no-op.
- Updated notebooks to reflect best practices

8.1.3 Release 6.1

- Pushed back the adoption of Cython 3 until a later date
- Fixed a bug that occurred when:
  - Using a AR, HAR or other model with lagged dependent variables; and
  - `rescale=True` or with data that was automatically rescaled.

8.1.4 Release 6.0

- Minimum supported Python is 3.9
- Bumped minimum NumPy, SciPy, pandas, statsmodels and Cython
- Removed dependence on property-cached
- Added compatability with Cython 3

8.2 Past Releases

8.2.1 Version 5

Changes Since 5.5

- Removed dependence on property-cached
- Bumped minimum NumPy, SciPy, pandas, statsmodels and Cython
- Added compatability with Cython 3

Release 5.5

- NumPy 1.25 fixes
- Initial pandas copy-on-write support
- Switched doc theme to sphinx-immaterial
- Small fixes for typing issues

Release 5.4

- Compatability release with pandas 2.0
- Add testing and wheel support for Python 3.11

Release 5.3

- Fixed a bug in `arch_model()` where `power` was not passed to the *FIGARCH* constructor (GH572).
- Fixed a bug that affected downstream projects due to an overly specific assert (GH569).

Release 5.2

- Fixed a bug in in `std_resid()` that would raise an exception when the data used to construct the model with a NumPy array (GH565).
- Fixed a bug in `forecast()` and related `forecast` methods when producing multi-step forecasts usign simulation with exogenous variables (GH551).

Release 5.1

Unit Root

- Improved automatic lag length selection in *DFGLS* by using OLS rather than GLS detrended data when selecting the lag length. This problem was studied by Perron, P., & Qu, Z. (2007).

Release 5.0

Unit Root

- All unit root tests are now immutable, and so properties such as `trend` cannot be set after the test is created.

Bootstrap

- Added `seed` keyword argument to all bootstraps (e.g., *IIDBootstrap* and *StationaryBootstrap*) that allows a NumPy `numpy.random.Generator` to be used. The `seed` keyword argument also accepts legacy `numpy.random.RandomState` instances and integers. If an integer is passed, the random number generator is constructed by calling `numpy.random.default_rng()` The `seed` keyword argument replaces the `random_state` keyword argument.
- The *random_state()* property has also been deprecated in favor of *generator()*.
- The *get_state()* and *set_state()* methods have been replaced by the *state()* property.

Volatility Modeling

- Added `seed` keyword argument to all distributions (e.g., *Normal* and *StudentsT*) that allows a NumPy `numpy.random.Generator` to be used. The `seed` keyword argument also accepts legacy `numpy.random.RandomState` instances and integers. If an integer is passed, the random number generator is constructed by calling `numpy.random.default_rng()` The `seed` keyword argument replaces the `random_state` keyword argument.
- The `random_state()` property has also been deprecated in favor of `generator()`.
- Added *ARCHInMean* mean process supporting (G)ARCH-in-mean models.
- Extended *VolatilityProcess* with *volatility_updater()* that contains a *VolatilityUpdater* to allow *ARCHInMean* to be created from different volatility processes.

Setup

- Added support for using an environmental variable to disable C-extension compilation.
  - Linux and OSX: `export ARCH_NO_BINARY=1`
  - PowerShell: `$env:ARCH_NO_BINARY=1`
  - cmd: `set ARCH_NO_BINARY=1`

8.2.2 Version 4

Release 4.19

- Added the keyword argument `reindex` to `forecast()` that allows the returned forecasts to have minimal size when `reindex=False`. The default is `reindex=True` which preserved the current behavior. This will change in a future release. Using `reindex=True` often requires substantially more memory than when `reindex=False`. This is especially true when using simulation or bootstrap-based forecasting.
- The default value `reindex` can be changed by importing

```
from arch.__future__ import reindexing
```

- Fixed handling of exogenous regressors in `forecast()`. It is now possible to pass values for $E_t[X_{t+h}]$ using the `x` argument.

Release 4.18

- Improved `fit()` performance of ARCH models.
- Fixed a bug where `` `typing_extensions `` was subtly introduced as a run-time dependency.

Release 4.17

- Fixed a bug that produced incorrect conditional volatility from EWMA models (GH458).

Release 4.16

- Added `APARCH` volatilty process (GH443).
- Added support for Python 3.9 in pyproject.toml (GH438).
- Fixed a bug in model degree-of-freedom calculation (GH437).
- Improved HARX initialization (GH417).

Release 4.15

- This is a minor release with doc fixes and other small updates. The only notable feature is `regression()` which returns regression results from the model estimated as part of the test (GH395).

Release 4.14

- Added Kernel-based long-run variance estimation in `arch.covariance.kernel`. Examples include the `Bartlett` and the `Parzen` kernels. All estimators suppose automatic bandwidth selection.
- Improved exceptions in `ADF`, `KPSS`, `PhillipsPerron`, `VarianceRatio`, and `ZivotAndrews` when test specification is infeasible to the time series being too short or the required regression model having reduced rank (GH364).
- Fixed a bug when using "bca" confidence intervals with `extra_kwargs` (GH366).
- Added Phillips-Ouliaris (`phillips_ouliaris()`) cointegration tests (GH360).

- Added three methods to estimate cointegrating vectors: *CanonicalCointegratingReg*, *DynamicOLS*, and *FullyModifiedOLS* (GH356, GH359).
- Added the Engle-Granger (*engle_granger()*) cointegration test (GH354).
- Issue warnings when unit root tests are mutated. Will raise after 5.0 is released.
- Fixed a bug in *arch.univariate.SkewStudent* which did not use the user-provided *RandomState* when one was provided. This prevented reproducing simulated values (GH353).

Release 4.13

- Restored the vendored copy of `property_cached` for conda package building.

Release 4.12

- Added typing support to all classes, functions and methods (GH338, GH341, GH342, GH343, GH345, GH346).
- Fixed an issue that caused tests to fail on SciPy 1.4+ (GH339).
- Dropped support for Python 3.5 inline with NEP 29 (GH334).
- Added methods to compute moment and lower partial moments for standardized residuals. See, for example, *moment()* and *partial_moment()* (GH329).
- Fixed a bug that produced an OverflowError when a time series has no variance (GH331).

Release 4.11

- Added *std_resid()* (GH326).
- Error if inputs are not ndarrays, DataFrames or Series (GH315).
- Added a check that the covariance is non-zero when using "studentized" confidence intervals. If the function bootstrapped produces statistics with 0 variance, it is not possible to studentized (GH322).

Release 4.10

- Fixed a bug in arch_lm_test that assumed that the model data is contained in a pandas Series. (GH313).
- Fixed a bug that can affect use in certain environments that reload modules (GH317).

Release 4.9

- Removed support for Python 2.7.
- Added *auto_bandwidth()* to compute optimized bandwidth for a number of common kernel covariance estimators (GH303). This code was written by Michael Rabba.
- Added a parameter *rescale* to `arch_model()` that allows the estimator to rescale data if it may help parameter estimation. If *rescale=True*, then the data will be rescaled by a power of 10 (e.g., 10, 100, or 1000) to produce a series with a residual variance between 1 and 1000. The model is then estimated on the rescaled data. The scale is reported *scale()*. If *rescale=None*, a warning is produced if the data appear to be poorly scaled, but no change of scale is applied. If *rescale=False*, no scale change is applied and no warning is issued.
- Fixed a bug when using the BCA bootstrap method where the leave-one-out jackknife used the wrong centering variable (GH288).

- Added `optimization_result()` to simplify checking for convergence of the numerical optimizer (GH292).
- Added *random_state* argument to `forecast()` to allow a `RandomState` object to be passed in when forecasting when *method='bootstrap'*. This allows the repeatable forecast to be produced (GH290).
- Fixed a bug in `VarianceRatio` that used the wrong variance in nonrobust inference with overlapping samples (GH286).

Release 4.8.1

- Fixed a bug which prevented extension modules from being correctly imported.

Release 4.8

- Added Zivot-Andrews unit root test `ZivotAndrews`. This code was originally written by Jim Varanelli.
- Added data dependent lag length selection to the KPSS test, `KPSS`. This code was originally written by Jim Varanelli.
- Added `IndependentSamplesBootstrap` to perform bootstrap inference on statistics from independent samples that may have uneven length (GH260).
- Added `arch_lm_test()` to perform ARCH-LM tests on model residuals or standardized residuals (GH261).
- Fixed a bug in `ADF` when applying to very short time series (GH262).
- Added ability to set the `random_state` when initializing a bootstrap (GH259).

Release 4.7

- Added support for Fractionally Integrated GARCH (FIGARCH) in `FIGARCH`.
- Enable user to specify a specific value of the *backcast* in place of the automatically generated value.
- Fixed a big where parameter-less models where incorrectly reported as having constant variance (GH248).

Release 4.6

- Added support for MIDAS volatility processes using Hyperbolic weighting in `MidasHyperbolic` (GH233).

Release 4.5

- Added a parameter to forecast that allows a user-provided callable random generator to be used in place of the model random generator (GH225).
- Added a low memory automatic lag selection method that can be used with very large time-series.
- Improved performance of automatic lag selection in ADF and related tests.

Release 4.4

- Added named parameters to Dickey-Fuller regressions.
- Removed use of the module-level NumPy RandomState. All random number generators use separate RandomState instances.
- Fixed a bug that prevented 1-step forecasts with exogenous regressors.
- Added the Generalized Error Distribution for univariate ARCH models.
- Fixed a bug in MCS when using the max method that prevented all included models from being listed.

Release 4.3

- Added `FixedVariance` volatility process which allows pre-specified variances to be used with a mean model. This has been added to allow so-called zig-zag estimation where a mean model is estimated with a fixed variance, and then a variance model is estimated on the residuals using a `ZeroMean` variance process.

Release 4.2

- Fixed a bug that prevented `fix` from being used with a new model (GH156).
- Added `first_obs` and `last_obs` parameters to `fix` to mimic `fit`.
- Added ability to jointly estimate smoothing parameter in EWMA variance when fitting the model.
- Added ability to pass optimization options to ARCH model estimation (GH195).

8.2.3 Version 3

- Added forecast code for mean forecasting
- Added volatility hedgehog plot
- Added `fix` to arch models which allows for user specified parameters instead of estimated parameters.
- Added Hansen's Skew T distribution to distribution (Stanislav Khrapov)
- Updated IPython notebooks to latest IPython version
- Bug and typo fixes to IPython notebooks
- Changed MCS to give a pvalue of 1.0 to best model. Previously was NaN
- Removed `hold_back` and `last_obs` from model initialization and to `fit` method to simplify estimating a model over alternative samples (e.g., rolling window estimation)
- Redefined `hold_back` to only accept integers so that is simply defined the number of observations held back. This number is now held out of the sample irrespective of the value of `first_obs`.

8.2.4 Version 2

Version 2.2

- Added multiple comparison procedures
- Typographical and other small changes

Version 2.1

- Add unit root tests: * Augmented Dickey-Fuller * Dickey-Fuller GLS * Phillips-Perron * KPSS * Variance Ratio
- Removed deprecated locations for ARCH modeling functions

8.2.5 Version 1

Version 1.1

- Refactored to move the univariate routines to *arch.univariate* and added deprecation warnings in the old locations
- Enable *numba* jit compilation in the python recursions
- Added a bootstrap framework, which will be used in future versions. The bootstrap framework is general purpose and can be used via high-level functions such as *conf_int* or *cov*, or as a low level iterator using *bootstrap*

CHAPTER

NINE

CITATION

This package should be cited using Zenodo. For example, for the 4.13 release,

CHAPTER

TEN

INDEX

- genindex
- modindex

BIBLIOGRAPHY

[Chernick] Chernick, M. R. (2011). *Bootstrap methods: A guide for practitioners and researchers* (Vol. 619). John Wiley & Sons.

[Davidson] Davison, A. C. (1997). *Bootstrap methods and their application* (Vol. 1). Cambridge university press.

[EfronTibshirani] Efron, B., & Tibshirani, R. J. (1994). *An introduction to the bootstrap* (Vol. 57). CRC press.

[PolitisRomanoWolf] Politis, D. N., & Romano, J. P. M. Wolf, 1999. *Subsampling*.

[CarpenterBithell] Carpenter, J., & Bithell, J. (2000). "Bootstrap confidence intervals: when, which, what? A practical guide for medical statisticians." *Statistics in medicine*, 19(9), 1141-1164.

[DavidsonMacKinnon] Davidson, R., & MacKinnon, J. G. (2006). "Bootstrap methods in econometrics." *Palgrave Handbook of Econometrics*, 1, 812-38.

[DiCiccioEfron] DiCiccio, T. J., & Efron, B. (1996). "Bootstrap confidence intervals." *Statistical Science*, 189-212.

[Efron] Efron, B. (1987). "Better bootstrap confidence intervals." *Journal of the American statistical Association*, 82(397), 171-185.

[Hansen] Hansen, P. R. (2005). A test for superior predictive ability. *Journal of Business & Economic Statistics*, 23(4).

[HansenLundeNason] Hansen, P. R., Lunde, A., & Nason, J. M. (2011). The model confidence set. *Econometrica*, 79(2), 453-497.

[RomanoWolf] Romano, J. P., & Wolf, M. (2005). Stepwise multiple testing as formalized data snooping. *Econometrica*, 73(4), 1237-1282.

[White] White, H. (2000). A reality check for data snooping. *Econometrica*, 68(5), 1097-1126.

PYTHON MODULE INDEX

a

arch.bootstrap, 249
arch.bootstrap.multiple_comparison, 330
arch.covariance.kernel, 405
arch.unitroot, 350
arch.unitroot.cointegration, 380
arch.utility.testing, 248

INDEX

A

ADF (*class in arch.unitroot*), 350
aic (*arch.univariate.base.ARCHModelFixedResult property*), 246
aic (*arch.univariate.base.ARCHModelResult property*), 238
alternative_hypothesis (*arch.unitroot.ADF property*), 352
alternative_hypothesis (*arch.unitroot.cointegration.EngleGrangerTestResults property*), 400
alternative_hypothesis (*arch.unitroot.cointegration.PhillipsOuliarisTestResults property*), 403
alternative_hypothesis (*arch.unitroot.DFGLS property*), 356
alternative_hypothesis (*arch.unitroot.KPSS property*), 370
alternative_hypothesis (*arch.unitroot.PhillipsPerron property*), 360
alternative_hypothesis (*arch.unitroot.VarianceRatio property*), 366
alternative_hypothesis (*arch.unitroot.ZivotAndrews property*), 363
Andrews (*class in arch.covariance.kernel*), 405
APARCH (*class in arch.univariate*), 163
apply() (*arch.bootstrap.CircularBlockBootstrap method*), 301
apply() (*arch.bootstrap.IIDBootstrap method*), 269
apply() (*arch.bootstrap.IndependentSamplesBootstrap method*), 279
apply() (*arch.bootstrap.MovingBlockBootstrap method*), 311
apply() (*arch.bootstrap.StationaryBootstrap method*), 290
ARCH (*class in arch.univariate*), 156
arch.bootstrap
module, 249
arch.bootstrap.multiple_comparison
module, 330
arch.covariance.kernel
module, 405
arch.unitroot
module, 350
arch.unitroot.cointegration
module, 380
arch.utility.testing
module, 248
arch_lm_test() (*arch.univariate.base.ARCHModelFixedResult method*), 242
arch_lm_test() (*arch.univariate.base.ARCHModelResult method*), 233
arch_model() (*in module arch.univariate*), 5
ARCHInMean (*class in arch.univariate*), 97
ARCHModel (*class in arch.univariate.base*), 106
ARCHModelFixedResult (*class in arch.univariate.base*), 241
ARCHModelForecast (*class in arch.univariate.base*), 23
ARCHModelForecastSimulation (*class in arch.univariate.base*), 25
ARCHModelResult (*class in arch.univariate.base*), 232
ARX (*class in arch.univariate*), 69
auto_bandwidth() (*in module arch.unitroot*), 371

B

backcast() (*arch.univariate.APARCH method*), 164
backcast() (*arch.univariate.ARCH method*), 157
backcast() (*arch.univariate.ConstantVariance method*), 114
backcast() (*arch.univariate.EGARCH method*), 136
backcast() (*arch.univariate.EWMAVariance method*), 172
backcast() (*arch.univariate.FIGARCH method*), 129
backcast() (*arch.univariate.FixedVariance method*), 186
backcast() (*arch.univariate.GARCH method*), 122
backcast() (*arch.univariate.HARCH method*), 143
backcast() (*arch.univariate.MIDASHyperbolic method*), 150
backcast() (*arch.univariate.RiskMetrics2006 method*), 179
backcast() (*arch.univariate.volatility.VolatilityProcess method*), 192

backcast_transform() (*arch.univariate.APARCH method*), 165
backcast_transform() (*arch.univariate.ARCH method*), 157
backcast_transform() (*arch.univariate.ConstantVariance method*), 114
backcast_transform() (*arch.univariate.EGARCH method*), 137
backcast_transform() (*arch.univariate.EWMAVariance method*), 172
backcast_transform() (*arch.univariate.FIGARCH method*), 129
backcast_transform() (*arch.univariate.FixedVariance method*), 186
backcast_transform() (*arch.univariate.GARCH method*), 122
backcast_transform() (*arch.univariate.HARCH method*), 144
backcast_transform() (*arch.univariate.MIDASHyperbolic method*), 151
backcast_transform() (*arch.univariate.RiskMetrics2006 method*), 179
backcast_transform() (*arch.univariate.volatility.VolatilityProcess method*), 193
bandwidth (*arch.covariance.kernel.Andrews property*), 406
bandwidth (*arch.covariance.kernel.Bartlett property*), 410
bandwidth (*arch.covariance.kernel.Gallant property*), 413
bandwidth (*arch.covariance.kernel.NeweyWest property*), 416
bandwidth (*arch.covariance.kernel.Parzen property*), 420
bandwidth (*arch.covariance.kernel.ParzenCauchy property*), 424
bandwidth (*arch.covariance.kernel.ParzenGeometric property*), 427
bandwidth (*arch.covariance.kernel.ParzenRiesz property*), 431
bandwidth (*arch.covariance.kernel.QuadraticSpectral property*), 435
bandwidth (*arch.covariance.kernel.TukeyHamming property*), 438
bandwidth (*arch.covariance.kernel.TukeyHanning property*), 442
bandwidth (*arch.covariance.kernel.TukeyParzen property*), 446
bandwidth (*arch.unitroot.cointegration.CointegrationAnalysisResults property*), 392
bandwidth (*arch.unitroot.cointegration.DynamicOLSResults property*), 395
bandwidth (*arch.unitroot.cointegration.PhillipsOuliarisTestResults property*), 403
bandwidth_scale (*arch.covariance.kernel.Andrews property*), 406
bandwidth_scale (*arch.covariance.kernel.Bartlett property*), 410
bandwidth_scale (*arch.covariance.kernel.Gallant property*), 413
bandwidth_scale (*arch.covariance.kernel.NeweyWest property*), 416
bandwidth_scale (*arch.covariance.kernel.Parzen property*), 420
bandwidth_scale (*arch.covariance.kernel.ParzenCauchy property*), 424
bandwidth_scale (*arch.covariance.kernel.ParzenGeometric property*), 427
bandwidth_scale (*arch.covariance.kernel.ParzenRiesz property*), 431
bandwidth_scale (*arch.covariance.kernel.QuadraticSpectral property*), 435
bandwidth_scale (*arch.covariance.kernel.TukeyHamming property*), 438
bandwidth_scale (*arch.covariance.kernel.TukeyHanning property*), 442
bandwidth_scale (*arch.covariance.kernel.TukeyParzen property*), 446
Bartlett (*class in arch.covariance.kernel*), 408
better_models() (*arch.bootstrap.SPA method*), 331
bic (*arch.univariate.base.ARCHModelFixedResult property*), 246
bic (*arch.univariate.base.ARCHModelResult property*), 238
bootstrap() (*arch.bootstrap.CircularBlockBootstrap method*), 301
bootstrap() (*arch.bootstrap.IIDBootstrap method*), 269
bootstrap() (*arch.bootstrap.IndependentSamplesBootstrap method*), 280
bootstrap() (*arch.bootstrap.MovingBlockBootstrap method*), 312
bootstrap() (*arch.bootstrap.StationaryBootstrap method*), 291
bounds() (*arch.univariate.APARCH method*), 165
bounds() (*arch.univariate.ARCH method*), 158
bounds() (*arch.univariate.ARCHInMean method*), 98
bounds() (*arch.univariate.ARX method*), 70
bounds() (*arch.univariate.base.ARCHModel method*), 107
bounds() (*arch.univariate.ConstantMean method*), 61
bounds() (*arch.univariate.ConstantVariance method*),

115
bounds() (arch.univariate.distribution.Distribution method), 227
bounds() (arch.univariate.EGARCH method), 137
bounds() (arch.univariate.EWMAVariance method), 172
bounds() (arch.univariate.FIGARCH method), 130
bounds() (arch.univariate.FixedVariance method), 186
bounds() (arch.univariate.GARCH method), 122
bounds() (arch.univariate.GeneralizedError method), 222
bounds() (arch.univariate.HARCH method), 144
bounds() (arch.univariate.HARX method), 80
bounds() (arch.univariate.LS method), 89
bounds() (arch.univariate.MIDASHyperbolic method), 151
bounds() (arch.univariate.Normal method), 206
bounds() (arch.univariate.RiskMetrics2006 method), 179
bounds() (arch.univariate.SkewStudent method), 217
bounds() (arch.univariate.StudentsT method), 211
bounds() (arch.univariate.volatility.VolatilityProcess method), 193
bounds() (arch.univariate.ZeroMean method), 53

C

CanonicalCointegratingReg (class in arch.unitroot.cointegration), 388
cdf() (arch.univariate.distribution.Distribution method), 228
cdf() (arch.univariate.GeneralizedError method), 222
cdf() (arch.univariate.Normal method), 207
cdf() (arch.univariate.SkewStudent method), 217
cdf() (arch.univariate.StudentsT method), 212
centered (arch.covariance.kernel.Andrews property), 406
centered (arch.covariance.kernel.Bartlett property), 410
centered (arch.covariance.kernel.Gallant property), 413
centered (arch.covariance.kernel.NeweyWest property), 417
centered (arch.covariance.kernel.Parzen property), 420
centered (arch.covariance.kernel.ParzenCauchy property), 424
centered (arch.covariance.kernel.ParzenGeometric property), 428
centered (arch.covariance.kernel.ParzenRiesz property), 431
centered (arch.covariance.kernel.QuadraticSpectral property), 435
centered (arch.covariance.kernel.TukeyHamming property), 439
centered (arch.covariance.kernel.TukeyHanning property), 442
centered (arch.covariance.kernel.TukeyParzen property), 446
CircularBlockBootstrap (class in arch.bootstrap), 299
clone() (arch.bootstrap.CircularBlockBootstrap method), 302
clone() (arch.bootstrap.IIDBootstrap method), 270
clone() (arch.bootstrap.IndependentSamplesBootstrap method), 281
clone() (arch.bootstrap.MovingBlockBootstrap method), 313
clone() (arch.bootstrap.StationaryBootstrap method), 292
cointegrating_vector (arch.unitroot.cointegration.EngleGrangerTestResults property), 400
cointegrating_vector (arch.unitroot.cointegration.PhillipsOuliarisTestResults property), 403
CointegrationAnalysisResults (class in arch.unitroot.cointegration), 391
common_asym (arch.univariate.APARCH property), 169
compute() (arch.bootstrap.MCS method), 337
compute() (arch.bootstrap.SPA method), 332
compute() (arch.bootstrap.StepM method), 335
compute_param_cov() (arch.univariate.ARCHInMean method), 98
compute_param_cov() (arch.univariate.ARX method), 71
compute_param_cov() (arch.univariate.base.ARCHModel method), 107
compute_param_cov() (arch.univariate.ConstantMean method), 62
compute_param_cov() (arch.univariate.HARX method), 80
compute_param_cov() (arch.univariate.LS method), 89
compute_param_cov() (arch.univariate.ZeroMean method), 53
compute_variance() (arch.univariate.APARCH method), 165
compute_variance() (arch.univariate.ARCH method), 158
compute_variance() (arch.univariate.ConstantVariance method), 115
compute_variance() (arch.univariate.EGARCH method), 137
compute_variance() (arch.univariate.EWMAVariance method), 173
compute_variance() (arch.univariate.FIGARCH method), 130
compute_variance() (arch.univariate.FixedVariance method), 187
compute_variance() (arch.univariate.GARCH

method), 123
compute_variance() (arch.univariate.HARCH method), 144
compute_variance() (arch.univariate.MIDASHyperbolic method), 151
compute_variance() (arch.univariate.RiskMetrics2006 method), 180
compute_variance() (arch.univariate.volatility.VolatilityProcess method), 193
conditional_volatility (arch.univariate.base.ARCHModelFixedResult property), 247
conditional_volatility (arch.univariate.base.ARCHModelResult property), 238
conf_int() (arch.bootstrap.CircularBlockBootstrap method), 303
conf_int() (arch.bootstrap.IIDBootstrap method), 271
conf_int() (arch.bootstrap.IndependentSamplesBootstrap method), 281
conf_int() (arch.bootstrap.MovingBlockBootstrap method), 313
conf_int() (arch.bootstrap.StationaryBootstrap method), 292
conf_int() (arch.univariate.base.ARCHModelResult method), 233
ConstantMean (class in arch.univariate), 60
ConstantVariance (class in arch.univariate), 114
constraints() (arch.univariate.APARCH method), 166
constraints() (arch.univariate.ARCH method), 158
constraints() (arch.univariate.ARCHInMean method), 99
constraints() (arch.univariate.ARX method), 71
constraints() (arch.univariate.base.ARCHModel method), 108
constraints() (arch.univariate.ConstantMean method), 62
constraints() (arch.univariate.ConstantVariance method), 115
constraints() (arch.univariate.distribution.Distribution method), 228
constraints() (arch.univariate.EGARCH method), 138
constraints() (arch.univariate.EWMAVariance method), 173
constraints() (arch.univariate.FIGARCH method), 130
constraints() (arch.univariate.FixedVariance method), 187
constraints() (arch.univariate.GARCH method), 123
constraints() (arch.univariate.GeneralizedError method), 223
constraints() (arch.univariate.HARCH method), 145
constraints() (arch.univariate.HARX method), 81
constraints() (arch.univariate.LS method), 90
constraints() (arch.univariate.MIDASHyperbolic method), 152
constraints() (arch.univariate.Normal method), 207
constraints() (arch.univariate.RiskMetrics2006 method), 180
constraints() (arch.univariate.SkewStudent method), 217
constraints() (arch.univariate.StudentsT method), 212
constraints() (arch.univariate.volatility.VolatilityProcess method), 194
constraints() (arch.univariate.ZeroMean method), 54
convergence_flag (arch.univariate.base.ARCHModelResult property), 239
cov (arch.covariance.kernel.Andrews property), 407
cov (arch.covariance.kernel.Bartlett property), 410
cov (arch.covariance.kernel.Gallant property), 414
cov (arch.covariance.kernel.NeweyWest property), 417
cov (arch.covariance.kernel.Parzen property), 421
cov (arch.covariance.kernel.ParzenCauchy property), 424
cov (arch.covariance.kernel.ParzenGeometric property), 428
cov (arch.covariance.kernel.ParzenRiesz property), 432
cov (arch.covariance.kernel.QuadraticSpectral property), 435
cov (arch.covariance.kernel.TukeyHamming property), 439
cov (arch.covariance.kernel.TukeyHanning property), 443
cov (arch.covariance.kernel.TukeyParzen property), 446
cov (arch.unitroot.cointegration.CointegrationAnalysisResults property), 392
cov (arch.unitroot.cointegration.DynamicOLSResults property), 395
cov() (arch.bootstrap.CircularBlockBootstrap method), 304
cov() (arch.bootstrap.IIDBootstrap method), 272
cov() (arch.bootstrap.IndependentSamplesBootstrap method), 283
cov() (arch.bootstrap.MovingBlockBootstrap method), 315
cov() (arch.bootstrap.StationaryBootstrap method), 294
cov_type (arch.unitroot.cointegration.DynamicOLSResults property), 395
CovarianceEstimate (class in arch.covariance.kernel), 448
critical_values (arch.unitroot.ADF property), 352
critical_values (arch.unitroot.cointegration.EngleGrangerTestResults property), 400
critical_values (arch.unitroot.cointegration.PhillipsOuliarisTestResults property), 403
critical_values (arch.unitroot.DFGLS property), 356
critical_values (arch.unitroot.KPSS property), 370

critical_values (*arch.unitroot.PhillipsPerron property*), 360
critical_values (*arch.unitroot.VarianceRatio property*), 366
critical_values (*arch.unitroot.ZivotAndrews property*), 363
critical_values (*arch.utility.testing.WaldTestStatistic property*), 248
critical_values() (*arch.bootstrap.SPA method*), 332

D

data (*arch.bootstrap.CircularBlockBootstrap attribute*), 299
data (*arch.bootstrap.IIDBootstrap attribute*), 267
data (*arch.bootstrap.IndependentSamplesBootstrap attribute*), 277
data (*arch.bootstrap.MovingBlockBootstrap attribute*), 309
data (*arch.bootstrap.StationaryBootstrap attribute*), 288
debiased (*arch.unitroot.VarianceRatio property*), 366
delta (*arch.univariate.APARCH property*), 169
DFGLS (*class in arch.unitroot*), 354
distribution (*arch.univariate.ARCHInMean property*), 105
distribution (*arch.univariate.ARX property*), 77
distribution (*arch.univariate.base.ARCHModel property*), 113
distribution (*arch.univariate.ConstantMean property*), 68
distribution (*arch.univariate.HARX property*), 87
distribution (*arch.univariate.LS property*), 96
distribution (*arch.univariate.ZeroMean property*), 59
Distribution (*class in arch.univariate.distribution*), 227
distribution_order (*arch.unitroot.cointegration.EngleGrangerTestResults property*), 400
distribution_order (*arch.unitroot.cointegration.PhillipsOuliarisTestResults property*), 404
DynamicOLS (*class in arch.unitroot.cointegration*), 384
DynamicOLSResults (*class in arch.unitroot.cointegration*), 393

E

EGARCH (*class in arch.univariate*), 135
engle_granger() (*in module arch.unitroot.cointegration*), 380
EngleGrangerTestResults (*class in arch.unitroot.cointegration*), 398
EWMAVariance (*class in arch.univariate*), 171
excluded (*arch.bootstrap.MCS property*), 337

F

FIGARCH (*class in arch.univariate*), 127
fit() (*arch.unitroot.cointegration.CanonicalCointegratingReg method*), 390
fit() (*arch.unitroot.cointegration.DynamicOLS method*), 385
fit() (*arch.unitroot.cointegration.FullyModifiedOLS method*), 388
fit() (*arch.univariate.ARCHInMean method*), 99
fit() (*arch.univariate.ARX method*), 71
fit() (*arch.univariate.base.ARCHModel method*), 108
fit() (*arch.univariate.ConstantMean method*), 62
fit() (*arch.univariate.HARX method*), 81
fit() (*arch.univariate.LS method*), 90
fit() (*arch.univariate.ZeroMean method*), 54
fit_start (*arch.univariate.base.ARCHModelResult property*), 239
fit_stop (*arch.univariate.base.ARCHModelResult property*), 239
fix() (*arch.univariate.ARCHInMean method*), 100
fix() (*arch.univariate.ARX method*), 72
fix() (*arch.univariate.base.ARCHModel method*), 109
fix() (*arch.univariate.ConstantMean method*), 63
fix() (*arch.univariate.HARX method*), 82
fix() (*arch.univariate.LS method*), 91
fix() (*arch.univariate.ZeroMean method*), 55
FixedVariance (*class in arch.univariate*), 185
force_int (*arch.covariance.kernel.Andrews property*), 407
force_int (*arch.covariance.kernel.Bartlett property*), 411
force_int (*arch.covariance.kernel.Gallant property*), 414
force_int (*arch.covariance.kernel.NeweyWest property*), 417
force_int (*arch.covariance.kernel.Parzen property*), 421
force_int (*arch.covariance.kernel.ParzenCauchy property*), 425
force_int (*arch.covariance.kernel.ParzenGeometric property*), 428
force_int (*arch.covariance.kernel.ParzenRiesz property*), 432
force_int (*arch.covariance.kernel.QuadraticSpectral property*), 436
force_int (*arch.covariance.kernel.TukeyHamming property*), 439
force_int (*arch.covariance.kernel.TukeyHanning property*), 443
force_int (*arch.covariance.kernel.TukeyParzen property*), 447
forecast() (*arch.univariate.APARCH method*), 166
forecast() (*arch.univariate.ARCH method*), 159
forecast() (*arch.univariate.ARCHInMean method*), 101
forecast() (*arch.univariate.ARX method*), 73

forecast() (*arch.univariate.base.ARCHModel method*), 109
forecast() (*arch.univariate.base.ARCHModelFixedResult method*), 242
forecast() (*arch.univariate.base.ARCHModelResult method*), 234
forecast() (*arch.univariate.ConstantMean method*), 64
forecast() (*arch.univariate.ConstantVariance method*), 116
forecast() (*arch.univariate.EGARCH method*), 138
forecast() (*arch.univariate.EWMAVariance method*), 173
forecast() (*arch.univariate.FIGARCH method*), 131
forecast() (*arch.univariate.FixedVariance method*), 187
forecast() (*arch.univariate.GARCH method*), 123
forecast() (*arch.univariate.HARCH method*), 145
forecast() (*arch.univariate.HARX method*), 83
forecast() (*arch.univariate.LS method*), 91
forecast() (*arch.univariate.MIDASHyperbolic method*), 152
forecast() (*arch.univariate.RiskMetrics2006 method*), 180
forecast() (*arch.univariate.volatility.VolatilityProcess method*), 194
forecast() (*arch.univariate.ZeroMean method*), 55
form (*arch.univariate.ARCHInMean property*), 105
full_cov (*arch.unitroot.cointegration.DynamicOLSResults property*), 396
full_params (*arch.unitroot.cointegration.DynamicOLSResults property*), 396
FullyModifiedOLS (*class in arch.unitroot.cointegration*), 386

G

Gallant (*class in arch.covariance.kernel*), 412
GARCH (*class in arch.univariate*), 120
GeneralizedError (*class in arch.univariate*), 221
generator (*arch.bootstrap.CircularBlockBootstrap property*), 308
generator (*arch.bootstrap.IIDBootstrap property*), 276
generator (*arch.bootstrap.IndependentSamplesBootstrap property*), 287
generator (*arch.bootstrap.MovingBlockBootstrap property*), 319
generator (*arch.bootstrap.StationaryBootstrap property*), 298
generator (*arch.univariate.distribution.Distribution property*), 231
generator (*arch.univariate.GeneralizedError property*), 226
generator (*arch.univariate.Normal property*), 210
generator (*arch.univariate.SkewStudent property*), 221
generator (*arch.univariate.StudentsT property*), 215
get_state() (*arch.bootstrap.CircularBlockBootstrap method*), 306
get_state() (*arch.bootstrap.IIDBootstrap method*), 274
get_state() (*arch.bootstrap.IndependentSamplesBootstrap method*), 284
get_state() (*arch.bootstrap.MovingBlockBootstrap method*), 316
get_state() (*arch.bootstrap.StationaryBootstrap method*), 295

H

HARCH (*class in arch.univariate*), 142
HARX (*class in arch.univariate*), 78
hedgehog_plot() (*arch.univariate.base.ARCHModelFixedResult method*), 244
hedgehog_plot() (*arch.univariate.base.ARCHModelResult method*), 235

I

IIDBootstrap (*class in arch.bootstrap*), 267
included (*arch.bootstrap.MCS property*), 337
IndependentSamplesBootstrap (*class in arch.bootstrap*), 277
index (*arch.bootstrap.CircularBlockBootstrap property*), 308
index (*arch.bootstrap.IIDBootstrap property*), 276
index (*arch.bootstrap.IndependentSamplesBootstrap property*), 287
index (*arch.bootstrap.MovingBlockBootstrap property*), 319
index (*arch.bootstrap.StationaryBootstrap property*), 298
index (*arch.univariate.base.ARCHModelForecastSimulation property*), 25
initialize_update() (*arch.univariate.recursions_python.VolatilityUpdater method*), 199

K

kernel (*arch.unitroot.cointegration.CointegrationAnalysisResults property*), 392
kernel (*arch.unitroot.cointegration.DynamicOLSResults property*), 396
kernel (*arch.unitroot.cointegration.PhillipsOuliarisTestResults property*), 404
kernel_const (*arch.covariance.kernel.Andrews property*), 407
kernel_const (*arch.covariance.kernel.Bartlett property*), 411
kernel_const (*arch.covariance.kernel.Gallant property*), 414
kernel_const (*arch.covariance.kernel.NeweyWest property*), 417

kernel_const (*arch.covariance.kernel.Parzen property*), 421
kernel_const (*arch.covariance.kernel.ParzenCauchy property*), 425
kernel_const (*arch.covariance.kernel.ParzenGeometric property*), 428
kernel_const (*arch.covariance.kernel.ParzenRiesz property*), 432
kernel_const (*arch.covariance.kernel.QuadraticSpectral property*), 436
kernel_const (*arch.covariance.kernel.TukeyHamming property*), 439
kernel_const (*arch.covariance.kernel.TukeyHanning property*), 443
kernel_const (*arch.covariance.kernel.TukeyParzen property*), 447
kernel_weights (*arch.covariance.kernel.Andrews property*), 407
kernel_weights (*arch.covariance.kernel.Bartlett property*), 411
kernel_weights (*arch.covariance.kernel.Gallant property*), 414
kernel_weights (*arch.covariance.kernel.NeweyWest property*), 418
kernel_weights (*arch.covariance.kernel.Parzen property*), 421
kernel_weights (*arch.covariance.kernel.ParzenCauchy property*), 425
kernel_weights (*arch.covariance.kernel.ParzenGeometric property*), 429
kernel_weights (*arch.covariance.kernel.ParzenRiesz property*), 432
kernel_weights (*arch.covariance.kernel.QuadraticSpectral property*), 436
kernel_weights (*arch.covariance.kernel.TukeyHamming property*), 440
kernel_weights (*arch.covariance.kernel.TukeyHanning property*), 443
kernel_weights (*arch.covariance.kernel.TukeyParzen property*), 447
KPSS (*class in arch.unitroot*), 368
kw_data (*arch.bootstrap.CircularBlockBootstrap attribute*), 299
kw_data (*arch.bootstrap.IIDBootstrap attribute*), 267
kw_data (*arch.bootstrap.IndependentSamplesBootstrap attribute*), 277
kw_data (*arch.bootstrap.MovingBlockBootstrap attribute*), 309
kw_data (*arch.bootstrap.StationaryBootstrap attribute*), 289

L

lags (*arch.unitroot.ADF property*), 352
lags (*arch.unitroot.cointegration.DynamicOLSResults property*), 396
lags (*arch.unitroot.cointegration.EngleGrangerTestResults property*), 400
lags (*arch.unitroot.DFGLS property*), 356
lags (*arch.unitroot.KPSS property*), 370
lags (*arch.unitroot.PhillipsPerron property*), 360
lags (*arch.unitroot.VarianceRatio property*), 367
lags (*arch.unitroot.ZivotAndrews property*), 363
leads (*arch.unitroot.cointegration.DynamicOLSResults property*), 396
loglikelihood (*arch.univariate.base.ARCHModelFixedResult property*), 247
loglikelihood (*arch.univariate.base.ARCHModelResult property*), 239
loglikelihood() (*arch.univariate.distribution.Distribution method*), 228
loglikelihood() (*arch.univariate.GeneralizedError method*), 223
loglikelihood() (*arch.univariate.Normal method*), 207
loglikelihood() (*arch.univariate.SkewStudent method*), 218
loglikelihood() (*arch.univariate.StudentsT method*), 212
long_run (*arch.covariance.kernel.CovarianceEstimate property*), 449
long_run_variance (*arch.unitroot.cointegration.CointegrationAnalysisRe property*), 392
long_run_variance (*arch.unitroot.cointegration.DynamicOLSResults property*), 396
LS (*class in arch.univariate*), 88

M

max_lags (*arch.unitroot.ADF property*), 352
max_lags (*arch.unitroot.cointegration.EngleGrangerTestResults property*), 400
max_lags (*arch.unitroot.DFGLS property*), 356
MCS (*class in arch.bootstrap*), 336
mean (*arch.univariate.base.ARCHModelForecast property*), 24
MIDASHyperbolic (*class in arch.univariate*), 149
model (*arch.univariate.base.ARCHModelFixedResult property*), 247
model (*arch.univariate.base.ARCHModelResult property*), 239
module
arch.bootstrap, 249
arch.bootstrap.multiple_comparison, 330
arch.covariance.kernel, 405
arch.unitroot, 350
arch.unitroot.cointegration, 380
arch.utility.testing, 248

moment() (arch.univariate.distribution.Distribution method), 229
moment() (arch.univariate.GeneralizedError method), 224
moment() (arch.univariate.Normal method), 208
moment() (arch.univariate.SkewStudent method), 218
moment() (arch.univariate.StudentsT method), 213
MovingBlockBootstrap (class in arch.bootstrap), 309

N

name (arch.covariance.kernel.Andrews property), 408
name (arch.covariance.kernel.Bartlett property), 411
name (arch.covariance.kernel.Gallant property), 415
name (arch.covariance.kernel.NeweyWest property), 418
name (arch.covariance.kernel.Parzen property), 422
name (arch.covariance.kernel.ParzenCauchy property), 425
name (arch.covariance.kernel.ParzenGeometric property), 429
name (arch.covariance.kernel.ParzenRiesz property), 433
name (arch.covariance.kernel.QuadraticSpectral property), 436
name (arch.covariance.kernel.TukeyHamming property), 440
name (arch.covariance.kernel.TukeyHanning property), 444
name (arch.covariance.kernel.TukeyParzen property), 447
name (arch.unitroot.cointegration.EngleGrangerTestResults property), 400
name (arch.unitroot.cointegration.PhillipsOuliarisTestResults property), 404
name (arch.univariate.APARCH property), 170
name (arch.univariate.ARCH property), 162
name (arch.univariate.ARCHInMean property), 105
name (arch.univariate.ARX property), 77
name (arch.univariate.base.ARCHModel property), 113
name (arch.univariate.ConstantMean property), 68
name (arch.univariate.ConstantVariance property), 119
name (arch.univariate.distribution.Distribution property), 231
name (arch.univariate.EGARCH property), 141
name (arch.univariate.EWMAVariance property), 177
name (arch.univariate.FIGARCH property), 134
name (arch.univariate.FixedVariance property), 191
name (arch.univariate.GARCH property), 127
name (arch.univariate.GeneralizedError property), 226
name (arch.univariate.HARCH property), 148
name (arch.univariate.HARX property), 87
name (arch.univariate.LS property), 96
name (arch.univariate.MIDASHyperbolic property), 155
name (arch.univariate.Normal property), 210
name (arch.univariate.RiskMetrics2006 property), 184
name (arch.univariate.SkewStudent property), 221
name (arch.univariate.StudentsT property), 215
name (arch.univariate.volatility.VolatilityProcess property), 197
name (arch.univariate.ZeroMean property), 60
NeweyWest (class in arch.covariance.kernel), 415
nobs (arch.unitroot.ADF property), 353
nobs (arch.unitroot.DFGLS property), 357
nobs (arch.unitroot.KPSS property), 370
nobs (arch.unitroot.PhillipsPerron property), 360
nobs (arch.unitroot.VarianceRatio property), 367
nobs (arch.unitroot.ZivotAndrews property), 363
nobs (arch.univariate.base.ARCHModelFixedResult property), 247
nobs (arch.univariate.base.ARCHModelResult property), 239
Normal (class in arch.univariate), 206
null (arch.utility.testing.WaldTestStatistic property), 248
null_hypothesis (arch.unitroot.ADF property), 353
null_hypothesis (arch.unitroot.cointegration.EngleGrangerTestResults property), 401
null_hypothesis (arch.unitroot.cointegration.PhillipsOuliarisTestResults property), 404
null_hypothesis (arch.unitroot.DFGLS property), 357
null_hypothesis (arch.unitroot.KPSS property), 370
null_hypothesis (arch.unitroot.PhillipsPerron property), 360
null_hypothesis (arch.unitroot.VarianceRatio property), 367
null_hypothesis (arch.unitroot.ZivotAndrews property), 364
num_params (arch.univariate.APARCH property), 170
num_params (arch.univariate.ARCH property), 162
num_params (arch.univariate.ARCHInMean property), 105
num_params (arch.univariate.ARX property), 77
num_params (arch.univariate.base.ARCHModel property), 113
num_params (arch.univariate.base.ARCHModelFixedResult property), 247
num_params (arch.univariate.base.ARCHModelResult property), 239
num_params (arch.univariate.ConstantMean property), 68
num_params (arch.univariate.ConstantVariance property), 119
num_params (arch.univariate.EGARCH property), 141
num_params (arch.univariate.EWMAVariance property), 177
num_params (arch.univariate.FIGARCH property), 134
num_params (arch.univariate.FixedVariance property), 191
num_params (arch.univariate.GARCH property), 127
num_params (arch.univariate.HARCH property), 148
num_params (arch.univariate.HARX property), 87
num_params (arch.univariate.LS property), 96

num_params (*arch.univariate.MIDASHyperbolic property*), 155
num_params (*arch.univariate.RiskMetrics2006 property*), 184
num_params (*arch.univariate.volatility.VolatilityProcess property*), 197
num_params (*arch.univariate.ZeroMean property*), 60

O

one_sided (*arch.covariance.kernel.CovarianceEstimate property*), 449
one_sided_strict (*arch.covariance.kernel.CovarianceEstimate property*), 449
opt_bandwidth (*arch.covariance.kernel.Andrews property*), 408
opt_bandwidth (*arch.covariance.kernel.Bartlett property*), 411
opt_bandwidth (*arch.covariance.kernel.Gallant property*), 415
opt_bandwidth (*arch.covariance.kernel.NeweyWest property*), 418
opt_bandwidth (*arch.covariance.kernel.Parzen property*), 422
opt_bandwidth (*arch.covariance.kernel.ParzenCauchy property*), 425
opt_bandwidth (*arch.covariance.kernel.ParzenGeometric property*), 429
opt_bandwidth (*arch.covariance.kernel.ParzenRiesz property*), 433
opt_bandwidth (*arch.covariance.kernel.QuadraticSpectral property*), 436
opt_bandwidth (*arch.covariance.kernel.TukeyHamming property*), 440
opt_bandwidth (*arch.covariance.kernel.TukeyHanning property*), 444
opt_bandwidth (*arch.covariance.kernel.TukeyParzen property*), 447
optimal_block_length() (*in module arch.bootstrap*), 320
optimization_result (*arch.univariate.base.ARCHModelResult property*), 240
overlap (*arch.unitroot.VarianceRatio property*), 367

P

param_cov (*arch.univariate.base.ARCHModelResult property*), 240
parameter_names() (*arch.univariate.APARCH method*), 167
parameter_names() (*arch.univariate.ARCH method*), 160
parameter_names() (*arch.univariate.ARCHInMean method*), 103
parameter_names() (*arch.univariate.ARX method*), 75
parameter_names() (*arch.univariate.base.ARCHModel method*), 111
parameter_names() (*arch.univariate.ConstantMean method*), 66
parameter_names() (*arch.univariate.ConstantVariance method*), 117
parameter_names() (*arch.univariate.distribution.Distribution method*), 229
parameter_names() (*arch.univariate.EGARCH method*), 139
parameter_names() (*arch.univariate.EWMAVariance method*), 174
parameter_names() (*arch.univariate.FIGARCH method*), 132
parameter_names() (*arch.univariate.FixedVariance method*), 188
parameter_names() (*arch.univariate.GARCH method*), 124
parameter_names() (*arch.univariate.GeneralizedError method*), 224
parameter_names() (*arch.univariate.HARCH method*), 146
parameter_names() (*arch.univariate.HARX method*), 85
parameter_names() (*arch.univariate.LS method*), 93
parameter_names() (*arch.univariate.MIDASHyperbolic method*), 153
parameter_names() (*arch.univariate.Normal method*), 208
parameter_names() (*arch.univariate.RiskMetrics2006 method*), 182
parameter_names() (*arch.univariate.SkewStudent method*), 219
parameter_names() (*arch.univariate.StudentsT method*), 213
parameter_names() (*arch.univariate.volatility.VolatilityProcess method*), 195
parameter_names() (*arch.univariate.ZeroMean method*), 57
params (*arch.unitroot.cointegration.CointegrationAnalysisResults property*), 392
params (*arch.unitroot.cointegration.DynamicOLSResults property*), 397
params (*arch.univariate.base.ARCHModelFixedResult property*), 247
params (*arch.univariate.base.ARCHModelResult property*), 240
partial_moment() (*arch.univariate.distribution.Distribution method*), 229
partial_moment() (*arch.univariate.GeneralizedError method*), 224
partial_moment() (*arch.univariate.Normal method*), 208
partial_moment() (*arch.univariate.SkewStudent*

method), 219
partial_moment() (arch.univariate.StudentsT method), 213
Parzen (class in arch.covariance.kernel), 419
ParzenCauchy (class in arch.covariance.kernel), 422
ParzenGeometric (class in arch.covariance.kernel), 426
ParzenRiesz (class in arch.covariance.kernel), 430
phillips_ouliaris() (in module arch.unitroot.cointegration), 381
PhillipsOuliarisTestResults (class in arch.unitroot.cointegration), 402
PhillipsPerron (class in arch.unitroot), 358
plot() (arch.unitroot.cointegration.EngleGrangerTestResults method), 399
plot() (arch.unitroot.cointegration.PhillipsOuliarisTestResults method), 402
plot() (arch.univariate.base.ARCHModelFixedResult method), 245
plot() (arch.univariate.base.ARCHModelResult method), 236
pos_data (arch.bootstrap.CircularBlockBootstrap attribute), 299
pos_data (arch.bootstrap.IIDBootstrap attribute), 267
pos_data (arch.bootstrap.IndependentSamplesBootstrap attribute), 277
pos_data (arch.bootstrap.MovingBlockBootstrap attribute), 309
pos_data (arch.bootstrap.StationaryBootstrap attribute), 288
ppf() (arch.univariate.distribution.Distribution method), 230
ppf() (arch.univariate.GeneralizedError method), 225
ppf() (arch.univariate.Normal method), 209
ppf() (arch.univariate.SkewStudent method), 220
ppf() (arch.univariate.StudentsT method), 214
pval (arch.utility.testing.WaldTestStatistic property), 248
pvalue (arch.unitroot.ADF property), 353
pvalue (arch.unitroot.cointegration.EngleGrangerTestResults property), 401
pvalue (arch.unitroot.cointegration.PhillipsOuliarisTestResults property), 404
pvalue (arch.unitroot.DFGLS property), 357
pvalue (arch.unitroot.KPSS property), 370
pvalue (arch.unitroot.PhillipsPerron property), 360
pvalue (arch.unitroot.VarianceRatio property), 367
pvalue (arch.unitroot.ZivotAndrews property), 364
pvalues (arch.bootstrap.MCS property), 338
pvalues (arch.bootstrap.SPA property), 333
pvalues (arch.unitroot.cointegration.CointegrationAnalysisResults property), 392
pvalues (arch.unitroot.cointegration.DynamicOLSResults property), 397
pvalues (arch.univariate.base.ARCHModelResult property), 240

Q

QuadraticSpectral (class in arch.covariance.kernel), 433

R

random_state (arch.bootstrap.CircularBlockBootstrap property), 308
random_state (arch.bootstrap.IIDBootstrap property), 276
random_state (arch.bootstrap.IndependentSamplesBootstrap property), 287
random_state (arch.bootstrap.MovingBlockBootstrap property), 319
random_state (arch.bootstrap.StationaryBootstrap property), 298
random_state (arch.univariate.distribution.Distribution property), 231
random_state (arch.univariate.GeneralizedError property), 226
random_state (arch.univariate.Normal property), 210
random_state (arch.univariate.SkewStudent property), 221
random_state (arch.univariate.StudentsT property), 216
rate (arch.covariance.kernel.Andrews property), 408
rate (arch.covariance.kernel.Bartlett property), 412
rate (arch.covariance.kernel.Gallant property), 415
rate (arch.covariance.kernel.NeweyWest property), 418
rate (arch.covariance.kernel.Parzen property), 422
rate (arch.covariance.kernel.ParzenCauchy property), 426
rate (arch.covariance.kernel.ParzenGeometric property), 429
rate (arch.covariance.kernel.ParzenRiesz property), 433
rate (arch.covariance.kernel.QuadraticSpectral property), 437
rate (arch.covariance.kernel.TukeyHamming property), 440
rate (arch.covariance.kernel.TukeyHanning property), 444
rate (arch.covariance.kernel.TukeyParzen property), 448
regression (arch.unitroot.ADF property), 353
regression (arch.unitroot.DFGLS property), 357
regression (arch.unitroot.PhillipsPerron property), 360
reset() (arch.bootstrap.CircularBlockBootstrap method), 306
reset() (arch.bootstrap.IIDBootstrap method), 274
reset() (arch.bootstrap.IndependentSamplesBootstrap method), 284
reset() (arch.bootstrap.MCS method), 337
reset() (arch.bootstrap.MovingBlockBootstrap method), 316
reset() (arch.bootstrap.SPA method), 332
reset() (arch.bootstrap.StationaryBootstrap method), 295

reset() (*arch.bootstrap.StepM method*), 335
resid (*arch.unitroot.cointegration.CointegrationAnalysisResults property*), 392
resid (*arch.unitroot.cointegration.DynamicOLSResults property*), 397
resid (*arch.unitroot.cointegration.EngleGrangerTestResults property*), 401
resid (*arch.unitroot.cointegration.PhillipsOuliarisTestResults property*), 404
resid (*arch.univariate.base.ARCHModelFixedResult property*), 247
resid (*arch.univariate.base.ARCHModelResult property*), 240
resids() (*arch.univariate.ARCHInMean method*), 103
resids() (*arch.univariate.ARX method*), 75
resids() (*arch.univariate.base.ARCHModel method*), 112
resids() (*arch.univariate.ConstantMean method*), 66
resids() (*arch.univariate.HARX method*), 85
resids() (*arch.univariate.LS method*), 94
resids() (*arch.univariate.ZeroMean method*), 58
residual_variance (*arch.unitroot.cointegration.CointegrationAnalysisResults property*), 392
residual_variance (*arch.unitroot.cointegration.DynamicOLSResults property*), 397
residual_variance (*arch.univariate.base.ARCHModelForecast property*), 24
residual_variances (*arch.univariate.base.ARCHModelForecastSimulation property*), 26
residuals (*arch.univariate.base.ARCHModelForecastSimulation property*), 26
rho (*arch.unitroot.cointegration.EngleGrangerTestResults property*), 401
RiskMetrics2006 (*class in arch.univariate*), 178
robust (*arch.unitroot.VarianceRatio property*), 367
rsquared (*arch.unitroot.cointegration.CointegrationAnalysisResults property*), 393
rsquared (*arch.unitroot.cointegration.DynamicOLSResults property*), 397
rsquared (*arch.univariate.base.ARCHModelResult property*), 240
rsquared_adj (*arch.unitroot.cointegration.CointegrationAnalysisResults property*), 393
rsquared_adj (*arch.unitroot.cointegration.DynamicOLSResults property*), 398
rsquared_adj (*arch.univariate.base.ARCHModelResult property*), 240

S

scale (*arch.univariate.base.ARCHModelResult property*), 241
seed() (*arch.bootstrap.CircularBlockBootstrap method*), 306
seed() (*arch.bootstrap.IIDBootstrap method*), 274
seed() (*arch.bootstrap.IndependentSamplesBootstrap method*), 285
seed() (*arch.bootstrap.MCS method*), 337
seed() (*arch.bootstrap.MovingBlockBootstrap method*), 317
seed() (*arch.bootstrap.SPA method*), 332
seed() (*arch.bootstrap.StationaryBootstrap method*), 296
seed() (*arch.bootstrap.StepM method*), 335
set_state() (*arch.bootstrap.CircularBlockBootstrap method*), 306
set_state() (*arch.bootstrap.IIDBootstrap method*), 274
set_state() (*arch.bootstrap.IndependentSamplesBootstrap method*), 285
set_state() (*arch.bootstrap.MovingBlockBootstrap method*), 317
set_state() (*arch.bootstrap.StationaryBootstrap method*), 296
short_run (*arch.covariance.kernel.CovarianceEstimate property*), 449
simulate() (*arch.univariate.APARCH method*), 167
simulate() (*arch.univariate.ARCH method*), 160
simulate() (*arch.univariate.ARCHInMean method*), 103
simulate() (*arch.univariate.ARX method*), 76
simulate() (*arch.univariate.base.ARCHModel method*), 112
simulate() (*arch.univariate.ConstantMean method*), 67
simulate() (*arch.univariate.ConstantVariance method*), 117
simulate() (*arch.univariate.distribution.Distribution method*), 230
simulate() (*arch.univariate.EGARCH method*), 139
simulate() (*arch.univariate.EWMAVariance method*), 175
simulate() (*arch.univariate.FIGARCH method*), 132
simulate() (*arch.univariate.FixedVariance method*), 189
simulate() (*arch.univariate.GARCH method*), 125
simulate() (*arch.univariate.GeneralizedError method*), 225
simulate() (*arch.univariate.HARCH method*), 146
simulate() (*arch.univariate.HARX method*), 85
simulate() (*arch.univariate.LS method*), 94
simulate() (*arch.univariate.MIDASHyperbolic method*), 153
simulate() (*arch.univariate.Normal method*), 209
simulate() (*arch.univariate.RiskMetrics2006 method*), 182
simulate() (*arch.univariate.SkewStudent method*), 220
simulate() (*arch.univariate.StudentsT method*), 214
simulate() (*arch.univariate.volatility.VolatilityProcess method*), 195

simulate() (*arch.univariate.ZeroMean method*), 58
simulations (*arch.univariate.base.ARCHModelForecast property*), 24
SkewStudent (*class in arch.univariate*), 216
SPA (*class in arch.bootstrap*), 330
start (*arch.univariate.APARCH property*), 170
start (*arch.univariate.ARCH property*), 163
start (*arch.univariate.ConstantVariance property*), 120
start (*arch.univariate.EGARCH property*), 142
start (*arch.univariate.EWMAVariance property*), 177
start (*arch.univariate.FIGARCH property*), 135
start (*arch.univariate.FixedVariance property*), 191
start (*arch.univariate.GARCH property*), 127
start (*arch.univariate.HARCH property*), 149
start (*arch.univariate.MIDASHyperbolic property*), 156
start (*arch.univariate.RiskMetrics2006 property*), 184
start (*arch.univariate.volatility.VolatilityProcess property*), 198
starting_values() (*arch.univariate.APARCH method*), 168
starting_values() (*arch.univariate.ARCH method*), 161
starting_values() (*arch.univariate.ARCHInMean method*), 104
starting_values() (*arch.univariate.ARX method*), 77
starting_values() (*arch.univariate.base.ARCHModel method*), 112
starting_values() (*arch.univariate.ConstantMean method*), 68
starting_values() (*arch.univariate.ConstantVariance method*), 118
starting_values() (*arch.univariate.distribution.Distribution method*), 231
starting_values() (*arch.univariate.EGARCH method*), 140
starting_values() (*arch.univariate.EWMAVariance method*), 175
starting_values() (*arch.univariate.FIGARCH method*), 133
starting_values() (*arch.univariate.FixedVariance method*), 189
starting_values() (*arch.univariate.GARCH method*), 125
starting_values() (*arch.univariate.GeneralizedError method*), 226
starting_values() (*arch.univariate.HARCH method*), 147
starting_values() (*arch.univariate.HARX method*), 86
starting_values() (*arch.univariate.LS method*), 95
starting_values() (*arch.univariate.MIDASHyperbolic method*), 154
starting_values() (*arch.univariate.Normal method*), 210
starting_values() (*arch.univariate.RiskMetrics2006 method*), 182
starting_values() (*arch.univariate.SkewStudent method*), 220
starting_values() (*arch.univariate.StudentsT method*), 215
starting_values() (*arch.univariate.volatility.VolatilityProcess method*), 196
starting_values() (*arch.univariate.ZeroMean method*), 59
stat (*arch.unitroot.ADF property*), 353
stat (*arch.unitroot.cointegration.EngleGrangerTestResults property*), 401
stat (*arch.unitroot.cointegration.PhillipsOuliarisTestResults property*), 404
stat (*arch.unitroot.DFGLS property*), 357
stat (*arch.unitroot.KPSS property*), 370
stat (*arch.unitroot.PhillipsPerron property*), 361
stat (*arch.unitroot.VarianceRatio property*), 367
stat (*arch.unitroot.ZivotAndrews property*), 364
stat (*arch.utility.testing.WaldTestStatistic property*), 248
state (*arch.bootstrap.CircularBlockBootstrap property*), 309
state (*arch.bootstrap.IIDBootstrap property*), 277
state (*arch.bootstrap.IndependentSamplesBootstrap property*), 287
state (*arch.bootstrap.MovingBlockBootstrap property*), 319
state (*arch.bootstrap.StationaryBootstrap property*), 298
StationaryBootstrap (*class in arch.bootstrap*), 288
std_err (*arch.univariate.base.ARCHModelResult property*), 241
std_errors (*arch.unitroot.cointegration.CointegrationAnalysisResults property*), 393
std_errors (*arch.unitroot.cointegration.DynamicOLSResults property*), 398
std_resid (*arch.univariate.base.ARCHModelFixedResult property*), 248
std_resid (*arch.univariate.base.ARCHModelResult property*), 241
StepM (*class in arch.bootstrap*), 333
stop (*arch.univariate.APARCH property*), 170
stop (*arch.univariate.ARCH property*), 163
stop (*arch.univariate.ConstantVariance property*), 120
stop (*arch.univariate.EGARCH property*), 142
stop (*arch.univariate.EWMAVariance property*), 177
stop (*arch.univariate.FIGARCH property*), 135
stop (*arch.univariate.FixedVariance property*), 191
stop (*arch.univariate.GARCH property*), 127
stop (*arch.univariate.HARCH property*), 149
stop (*arch.univariate.MIDASHyperbolic property*), 156
stop (*arch.univariate.RiskMetrics2006 property*), 184

stop (*arch.univariate.volatility.VolatilityProcess property*), 198
StudentsT (*class in arch.univariate*), 211
subset() (*arch.bootstrap.SPA method*), 333
summary() (*arch.unitroot.ADF method*), 351
summary() (*arch.unitroot.cointegration.CointegrationAnalysisResults method*), 391
summary() (*arch.unitroot.cointegration.DynamicOLSResults method*), 394
summary() (*arch.unitroot.cointegration.EngleGrangerTestResults method*), 399
summary() (*arch.unitroot.cointegration.PhillipsOuliarisTestResults method*), 402
summary() (*arch.unitroot.DFGLS method*), 355
summary() (*arch.unitroot.KPSS method*), 369
summary() (*arch.unitroot.PhillipsPerron method*), 359
summary() (*arch.unitroot.VarianceRatio method*), 365
summary() (*arch.unitroot.ZivotAndrews method*), 362
summary() (*arch.univariate.base.ARCHModelFixedResult method*), 246
summary() (*arch.univariate.base.ARCHModelResult method*), 237
superior_models (*arch.bootstrap.StepM property*), 335

T

test_type (*arch.unitroot.PhillipsPerron property*), 361
trend (*arch.unitroot.ADF property*), 353
trend (*arch.unitroot.cointegration.EngleGrangerTestResults property*), 401
trend (*arch.unitroot.cointegration.PhillipsOuliarisTestResults property*), 404
trend (*arch.unitroot.DFGLS property*), 357
trend (*arch.unitroot.KPSS property*), 371
trend (*arch.unitroot.PhillipsPerron property*), 361
trend (*arch.unitroot.VarianceRatio property*), 367
trend (*arch.unitroot.ZivotAndrews property*), 364
truncation (*arch.univariate.FIGARCH property*), 135
TukeyHamming (*class in arch.covariance.kernel*), 437
TukeyHanning (*class in arch.covariance.kernel*), 441
TukeyParzen (*class in arch.covariance.kernel*), 444
tvalues (*arch.unitroot.cointegration.CointegrationAnalysisResults property*), 393
tvalues (*arch.unitroot.cointegration.DynamicOLSResults property*), 398
tvalues (*arch.univariate.base.ARCHModelResult property*), 241

U

update() (*arch.univariate.APARCH method*), 168
update() (*arch.univariate.ARCH method*), 161
update() (*arch.univariate.ConstantVariance method*), 118
update() (*arch.univariate.EGARCH method*), 140
update() (*arch.univariate.EWMAVariance method*), 175
update() (*arch.univariate.FIGARCH method*), 133
update() (*arch.univariate.FixedVariance method*), 189
update() (*arch.univariate.GARCH method*), 125
update() (*arch.univariate.HARCH method*), 147
update() (*arch.univariate.MIDASHyperbolic method*), 154
update() (*arch.univariate.recursions_python.VolatilityUpdater method*), 199
update() (*arch.univariate.RiskMetrics2006 method*), 183
update() (*arch.univariate.volatility.VolatilityProcess method*), 196
update_indices() (*arch.bootstrap.CircularBlockBootstrap method*), 306
update_indices() (*arch.bootstrap.IIDBootstrap method*), 274
update_indices() (*arch.bootstrap.IndependentSamplesBootstrap method*), 285
update_indices() (*arch.bootstrap.MovingBlockBootstrap method*), 317
update_indices() (*arch.bootstrap.StationaryBootstrap method*), 296
updateable (*arch.univariate.APARCH property*), 170
updateable (*arch.univariate.ARCH property*), 163
updateable (*arch.univariate.ConstantVariance property*), 120
updateable (*arch.univariate.EGARCH property*), 142
updateable (*arch.univariate.EWMAVariance property*), 177
updateable (*arch.univariate.FIGARCH property*), 135
updateable (*arch.univariate.FixedVariance property*), 191
updateable (*arch.univariate.GARCH property*), 127
updateable (*arch.univariate.HARCH property*), 149
updateable (*arch.univariate.MIDASHyperbolic property*), 156
updateable (*arch.univariate.RiskMetrics2006 property*), 184
updateable (*arch.univariate.volatility.VolatilityProcess property*), 198

V

valid_trends (*arch.unitroot.ADF property*), 353
valid_trends (*arch.unitroot.DFGLS property*), 357
valid_trends (*arch.unitroot.KPSS property*), 371
valid_trends (*arch.unitroot.PhillipsPerron property*), 361
valid_trends (*arch.unitroot.VarianceRatio property*), 368
valid_trends (*arch.unitroot.ZivotAndrews property*), 364
values (*arch.univariate.base.ARCHModelForecastSimulation property*), 26

var() (*arch.bootstrap.CircularBlockBootstrap method*), 307
var() (*arch.bootstrap.IIDBootstrap method*), 275
var() (*arch.bootstrap.IndependentSamplesBootstrap method*), 285
var() (*arch.bootstrap.MovingBlockBootstrap method*), 317
var() (*arch.bootstrap.StationaryBootstrap method*), 296
variance (*arch.univariate.base.ARCHModelForecast property*), 25
variance_bounds() (*arch.univariate.APARCH method*), 169
variance_bounds() (*arch.univariate.ARCH method*), 162
variance_bounds() (*arch.univariate.ConstantVariance method*), 119
variance_bounds() (*arch.univariate.EGARCH method*), 141
variance_bounds() (*arch.univariate.EWMAVariance method*), 176
variance_bounds() (*arch.univariate.FIGARCH method*), 134
variance_bounds() (*arch.univariate.FixedVariance method*), 190
variance_bounds() (*arch.univariate.GARCH method*), 126
variance_bounds() (*arch.univariate.HARCH method*), 148
variance_bounds() (*arch.univariate.MIDASHyperbolic method*), 155
variance_bounds() (*arch.univariate.RiskMetrics2006 method*), 183
variance_bounds() (*arch.univariate.volatility.VolatilityProcess method*), 197
VarianceRatio (*class in arch.unitroot*), 364
variances (*arch.univariate.base.ARCHModelForecastSimulation property*), 26
volatility (*arch.univariate.ARCHInMean property*), 105
volatility (*arch.univariate.ARX property*), 78
volatility (*arch.univariate.base.ARCHModel property*), 113
volatility (*arch.univariate.ConstantMean property*), 68
volatility (*arch.univariate.HARX property*), 87
volatility (*arch.univariate.LS property*), 96
volatility (*arch.univariate.ZeroMean property*), 60
volatility_updater (*arch.univariate.APARCH property*), 170
volatility_updater (*arch.univariate.ARCH property*), 163
volatility_updater (*arch.univariate.ConstantVariance property*), 120
volatility_updater (*arch.univariate.EGARCH property*), 142
volatility_updater (*arch.univariate.EWMAVariance property*), 177
volatility_updater (*arch.univariate.FIGARCH property*), 135
volatility_updater (*arch.univariate.FixedVariance property*), 191
volatility_updater (*arch.univariate.GARCH property*), 127
volatility_updater (*arch.univariate.HARCH property*), 149
volatility_updater (*arch.univariate.MIDASHyperbolic property*), 156
volatility_updater (*arch.univariate.RiskMetrics2006 property*), 184
volatility_updater (*arch.univariate.volatility.VolatilityProcess property*), 198
VolatilityProcess (*class in arch.univariate.volatility*), 192
VolatilityUpdater (*class in arch.univariate.recursions_python*), 198
vr (*arch.unitroot.VarianceRatio property*), 368

W

WaldTestStatistic (*class in arch.utility.testing*), 248

X

x (*arch.univariate.ARCHInMean property*), 106
x (*arch.univariate.ARX property*), 78
x (*arch.univariate.ConstantMean property*), 69
x (*arch.univariate.HARX property*), 87
x (*arch.univariate.LS property*), 96
x (*arch.univariate.ZeroMean property*), 60

Y

y (*arch.unitroot.ADF property*), 353
y (*arch.unitroot.DFGLS property*), 357
y (*arch.unitroot.KPSS property*), 371
y (*arch.unitroot.PhillipsPerron property*), 361
y (*arch.unitroot.VarianceRatio property*), 368
y (*arch.unitroot.ZivotAndrews property*), 364
y (*arch.univariate.ARCHInMean property*), 106
y (*arch.univariate.ARX property*), 78
y (*arch.univariate.base.ARCHModel property*), 113
y (*arch.univariate.ConstantMean property*), 69
y (*arch.univariate.HARX property*), 88
y (*arch.univariate.LS property*), 96
y (*arch.univariate.ZeroMean property*), 60

Z

ZeroMean (*class in arch.univariate*), 52
ZivotAndrews (*class in arch.unitroot*), 361