# Contents

1 Univariate Volatility Models  .......................................................... 3
  1.1 Introduction to ARCH Models ...................................................... 3
  1.2 ARCH Modeling .................................................................. 13
  1.3 Forecasting .......................................................................... 26
  1.4 Volatility Forecasting ............................................................... 30
  1.5 Value-at-Risk Forecasting ......................................................... 38
  1.6 Volatility Scenarios ................................................................. 41
  1.7 Mean Models ....................................................................... 48
  1.8 Volatility Processes ................................................................. 82
  1.9 Using the Fixed Variance process .............................................. 129
  1.10 Distributions ....................................................................... 136
  1.11 Utilities ............................................................................. 148
  1.12 Theoretical Background .......................................................... 149

2 Bootstrapping .............................................................................. 151
  2.1 Bootstrap Examples ................................................................. 151
  2.2 Confidence Intervals ................................................................. 156
  2.3 Covariance Estimation ............................................................... 161
  2.4 Low-level Interfaces ................................................................. 162
  2.5 Semiparametric Bootstraps ....................................................... 164
  2.6 Parametric Bootstraps ............................................................... 165
  2.7 Independent, Identical Distributed Data (i.i.d.) ........................... 166
  2.8 Independent Samples ............................................................... 174
  2.9 Time-series Bootstraps .............................................................. 183
  2.10 References ........................................................................ 206

3 Multiple Comparison Procedures ................................................. 209
  3.1 Multiple Comparisons .............................................................. 209
  3.2 Module Reference .................................................................. 217
  3.3 References ......................................................................... 224

4 Unit Root Testing ........................................................................ 225
  4.1 Introduction ......................................................................... 225
  4.2 Unit Root Testing ................................................................. 226
  4.3 The Unit Root Tests ............................................................... 236

5 Change Logs .......................................................................... 259
The ARCH toolbox contains routines for:

- Univariate volatility models;
- Bootstrapping;
- Multiple comparison procedures; and
- Unit root tests.

Future plans are to continue to expand this toolbox to include additional routines relevant for the analysis of financial data.
arch.univariate provides both high-level (arch.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

\[
\begin{align*}
    r_t &= \mu + \epsilon_t \\
    \epsilon_t &= \sigma_t \epsilon_t \\
    \sigma_t^2 &= \omega + \alpha \sigma_{t-1}^2 + \beta \sigma_{t-1}^2
\end{align*}
\]

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()`

```python
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

```python
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

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

with the following fit 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:    5,   Func. Count:   38,   Neg. LLF:  5143.86337547
Iteration:    6,   Func. Count:   45,   Neg. LLF:  5143.02099169
Iteration:    8,   Func. Count:   60,   Neg. LLF:  5142.07189070
Iteration:    9,   Func. Count:   67,   Neg. LLF:  5141.416653
Iteration:   10,   Func. Count:   73,   Neg. LLF:  5141.39212288
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
```

```python
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

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]
```

(continues on next page)
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.

\[
\text{arch} \cdot \text{arch_model} \left( y, x=None, \text{mean}='Constant', \text{lags}=0, \text{vol}='Garch', p=1, o=0, q=1, \text{power}=2.0, \text{dist}='Normal', \text{hold_back}=\text{None, rescale}=\text{None} \right)
\]

Convenience function to simplify initialization of ARCH models

**Parameters**

- \( y \) \((\text{ndarray, Series, None})\) – The dependent variable
- \( x \) \((\text{np.array, DataFrame}, \text{optional})\) – Exogenous regressors. Ignored if model does not permit exogenous regressors.
- \text{mean} \((\text{str, optional})\) – Name of the mean model. Currently supported options are: ‘Constant’, ‘Zero’, ‘ARX’ and ‘HARX’
- \text{lags} \((\text{int or list (int), optional})\) – Either a scalar integer value indicating lag length or a list of integers specifying lag locations.
- \text{vol} \((\text{str, optional})\) – Name of the volatility model. Currently supported options are: ‘GARCH’ (default), ‘ARCH’, ‘EGARCH’, ‘FIARCH’ and ‘HARCH’
- \text{p} \((\text{int, optional})\) – Lag order of the symmetric innovation
- \text{o} \((\text{int, optional})\) – Lag order of the asymmetric innovation
- \text{q} \((\text{int, optional})\) – Lag order of lagged volatility or equivalent
- \text{power} \((\text{float, optional})\) – Power to use with GARCH and related models
- \text{dist} \((\text{int, optional})\) – 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”
- \text{hold_back} \((\text{int})\) – 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.

**Returns**

- \text{model} – Configured ARCH model

**Return type** \text{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.

1.1.2 Model Results

All model return the same object, a results class (ARCHModelResult)

```python
class arch.univariate.base.ARCHModelResult:
    def __init__(self, params, param_cov, r2, resid, volatility,
                 cov_type, dep_var, names, loglikelihood,
                 is_pandas, optim_output, fit_start, fit_stop,
                 model):
```

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 (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

`summary()`
Produce a summary of the results

`plot()`
Produce a plot of the volatility and standardized residuals

`conf_int()`
Confidence intervals

`loglikelihood`
Value of the log-likelihood
  
  Type  `float`

`params`
Estimated parameters
  
  Type  `Series`

`param_cov`
Estimated variance-covariance of the parameters
  
  Type  `DataFrame`

`resid`
`nobs` element array containing model residuals
  
  Type  `{ndarray, Series}`

`model`
Model instance used to produce the fit
  
  Type  `ARCHModel`

`arch_lm_test` (*lags*=`None`, *standardized*=`False`)
ARCH LM test for conditional heteroskedasticity

  Parameters
  
  • **lags** (*int*, *optional*) – Number of lags to include in the model. If not specified,

  • **standardized** (*bool*, *optional*) – 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`

`conf_int` (*alpha*=`0.05`)

  Parameters  `alpha` (*float*, *optional*) – 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  `ndarray`

1.1. Introduction to ARCH Models
**forecast** (params=None, horizon=1, start=None, align='origin', method='analytic', simulations=1000, rng=None, random_state=None)

Construct forecasts from estimated model

**Parameters**

- **params** *(ndarray, optional)* – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.

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

- **start** *(int, datetime, Timestamp, str, optional)* – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in ’1945-01-01’.

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

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

- **rng** *(callable, optional)* – 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** *(RandomState, optional)* – NumPy RandomState instance to use when method is ‘bootstrap’

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

**Return type** `ARCHModelForecast`

**Notes**

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

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

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

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

Plot forecasts from estimated model
Parameters

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

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

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

- **start** *(int, datetime or str, optional)* – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in ‘1945-01-01’. If not provided, the start is set to the earliest forecastable date.

- **type** *(['volatility', 'mean'])* – Quantity to plot, the forecast volatility or the forecast mean

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

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

Returns **fig** – Handle to the figure

Return type **figure**

Examples

```python
code
>>> 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(type='mean')
```

**plot** *(annualize=None, scale=None)*

Plot standardized residuals and conditional volatility

Parameters

- **annualize** *(str, optional)* – String containing frequency of data that indicates plot should contain annualized volatility. Supported values are ‘D’ (daily), ‘W’ (weekly) and ‘M’ (monthly), which scale variance by 252, 52, and 12, respectively.

- **scale** *(float, optional)* – Value to use when scaling returns to annualize. If scale is provided, annualize is ignored and the value in scale is used.

Returns **fig** – Handle to the figure

Return type **figure**

Examples
```python
>>> 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)
```

**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

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.

```python
class arch.univariate.base.ARCHModelFixedResult(params, resid, volatility, dep_var, names, loglikelihood, is_pandas, model)
```

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** *(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

**summary()**

Produce a summary of the results

**plot()**

Produce a plot of the volatility and standardized residuals

**forecast()**

Construct forecasts from a model

**loglikelihood**

Value of the log-likelihood

  - **Type** float
**params**
Estimated parameters

**Type** Series

**resid**
Element array containing model residuals

**Type** {ndarray, Series}

**model**
Model instance used to produce the fit

**Type** ARCHModel

**arch_lm_test** (*lags=None, standardized=False*)
ARCH LM test for conditional heteroskedasticity

**Parameters**
- **lags** (*int, optional*) – Number of lags to include in the model. If not specified,
- **standardized** (*bool, optional*) – 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

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

**Parameters**
- **params** (*ndarray, optional*) – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.
- **horizon** (*int, optional*) – Number of steps to forecast
- **start** (*{int, datetime, Timestamp, str}, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in ‘1945-01-01’.
- **align** (*str, optional*) – 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** ({'analytic', 'simulation', 'bootstrap'}, optional) – Method to use when producing the forecast. The default is analytic. The method only affects the variance forecast generation. Not all volatility models support all methods. In particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the ‘analytic’ method for horizons > 1.
- **simulations** (*int, optional*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.
• **rng** (*callable, optional*) – 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** (*RandomState, optional*) – NumPy RandomState instance to use when method is ‘bootstrap’

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

**Return type**  
`ARCHModelForecast`

**Notes**

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

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

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

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

Plot forecasts from estimated model

**Parameters**

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

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

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

• **start** (*int, datetime or str, optional*) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in ‘1945-01-01’. If not provided, the start is set to the earliest forecastable date.

• **type** (*{'volatility', 'mean'}*) – Quantity to plot, the forecast volatility or the forecast mean

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

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

**Returns**  
fig – Handle to the figure

**Return type**  
figure

---

**Chapter 1. Univariate Volatility Models**
Examples

```python
>>> 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(type='mean')
```

```python
plot(annualize=None, scale=None)
```

Plot standardized residuals and conditional volatility

Parameters

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

Returns

- `fig` – Handle to the figure

Return type

- `figure`

Examples

```python
>>> 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

```python
>>> fig = res.plot(annualize='D')
```

Override the usual scale of 252 to use 360 for an asset that trades most days of the year

```python
>>> fig = res.plot(scale=360)
```

```python
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`

1.2 ARCH Modeling

This setup code is required to run in an IPython notebook
These examples will all make use of financial data from Yahoo! Finance. This data set can be loaded from `arch.data.sp500`.

```python
[3]: 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()
figure = returns.plot()
```
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.

\[
\begin{align*}
    r_t &= \mu + \epsilon_t \\
    \sigma_t^2 &= \omega + \alpha \epsilon_{t-1}^2 + \beta \sigma_{t-1}^2 \\
    \epsilon_t &= \sigma_t e_t, \quad e_t \sim N(0,1)
\end{align*}
\]

The model is estimated by calling `fit`. The optional inputs `iter` controls the frequency of output from 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.

```python
from arch import arch_model
am = arch_model(returns)
res = am.fit(update_freq=5)
print(res.summary())
```

```
Iteration: 10, Func. Count: 72, Neg. LLF: 6936.718529994181
Optimization terminated successfully. (Exit mode 0)
Current function value: 6936.718476989043
Iterations: 11
Function evaluations: 79
Gradient evaluations: 11
```

```
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: -6936.72
Distribution: Normal AIC: 13881.4
Method: Maximum Likelihood BIC: 13907.5
No. Observations: 5030
Date: Wed, Aug 28 2019 Df Residuals: 5026
Time: 12:21:54 Df Model: 4
-----------------------------------------------------------------------------------------------
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]
```

(continues on next page)
Covariance estimator: robust

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

![Graph showing standardized residuals and annualized conditional volatility](image)

**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

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

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.

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

<table>
<thead>
<tr>
<th>Constant Mean - GJR-GARCH Model Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dep. Variable: Adj Close</td>
</tr>
<tr>
<td>Mean Model: Constant Mean</td>
</tr>
<tr>
<td>Vol Model: GJR-GARCH</td>
</tr>
<tr>
<td>Distribution: Normal</td>
</tr>
</tbody>
</table>

(continues on next page)
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

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

More general models with other powers ($\kappa$) have volatility dynamics given by

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

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

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

![Code block](image)
Financial returns are often heavy tailed, and a Student’s T distribution is a simple method to capture this feature. The call to \texttt{arch} changes the distribution from a Normal to a Student’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.

\begin{verbatim}
[8]: am = arch_model(returns, p=1, o=1, q=1, power=1.0, dist='StudentsT')
res = am.fit(update_freq=5)
print(res.summary())
\end{verbatim}

\textbf{Student’s T Errors}

\textit{Constant Mean - TARCH/ZARCH Model Results}

\begin{verbatim}
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: Wed, Aug 28 2019 Df Residuals: 5024
Time: 12:21:55 Df Model: 6
 mean Model
============================================================================
 coef std err t P>|t| 95.0% Conf. Int.
---------------------------------------------------
mu 0.0323 2.299e-03 14.037 9.214e-45 [2.776e-02,3.677e-02]

\end{verbatim}
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.

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

1.2. ARCH Modeling 19
Results generated with user-specified parameters. Std. errors not available when the model is not estimated.

```python
[10]: import pandas as pd
df = pd.concat([res.conditional_volatility, fixed_res.conditional_volatility],
                1)
df.columns = ['Estimated', 'Fixed']
subplot = df.plot()
```

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.

```python
[11]: import arch.data.core_cpi
core_cpi = arch.data.core_cpi.load()
ann_inflation = 100 * core_cpi.CPIFESL.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.
```python
[12]: from arch.univariate import ARX
    ar = ARX(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: 11.2764
Distribution:                   Normal  AIC:               -12.5529
Method:                                Maximum Likelihood  BIC:          10.3364
No. Observations:                          719
Date:        Wed, Aug 28 2019  Df Residuals:              714
Time:                   12:21:56  Df Model:                  5
```

Volatility Processes

Volatility processes can be added to 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.

```python
[13]: 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
Distribution:                  Normal  AIC:              -253.044
Method:                                Maximum Likelihood  BIC:         -207.265
No. Observations:                          719
Date:        Wed, Aug 28 2019  Df Residuals:              709
Time:                   12:21:56  Df Model:                  10
```

Covariance estimator: White’s Heteroskedasticity Consistent Estimator
Volatility Model

| coef    | std err  | t     | P>|t| | 95.0% Conf. Int. |
|---------|----------|-------|------|------------------|
| omega   | 7.6869e-03 | 1.602e-03 | 4.799 | 1.591e-06 | [4.548e-03,1.083e-02] |
| alpha[1]| 0.1345   | 4.003e-02 | 3.359 | 7.826e-04 | [5.600e-02,0.213] |
| alpha[2]| 0.2280   | 6.284e-02 | 3.628 | 2.860e-04 | [0.105,0.351] |
| alpha[3]| 0.1838   | 6.801e-02 | 2.702 | 6.894e-03 | [5.047e-02,0.317] |
| alpha[4]| 0.2538   | 7.826e-02 | 3.242 | 1.185e-03 | [0.100,0.407] |
| alpha[5]| 0.1954   | 7.092e-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.

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

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

### AR - ARCH Model Results

<table>
<thead>
<tr>
<th>Dep. Variable:</th>
<th>CPIFESL</th>
<th>R-squared:</th>
<th>0.991</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Model:</td>
<td>AR</td>
<td>Adj. R-squared:</td>
<td>0.991</td>
</tr>
<tr>
<td>Vol Model:</td>
<td>ARCH</td>
<td>Log-Likelihood:</td>
<td>142.863</td>
</tr>
<tr>
<td>Distribution:</td>
<td>Standardized Student's t</td>
<td>AIC:</td>
<td>-263.727</td>
</tr>
<tr>
<td>Method:</td>
<td>Maximum Likelihood</td>
<td>BIC:</td>
<td>-213.370</td>
</tr>
<tr>
<td>No. Observations:</td>
<td>719</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date:</td>
<td>Wed, Aug 28 2019</td>
<td>Df Residuals:</td>
<td>708</td>
</tr>
<tr>
<td>Time:</td>
<td>12:21:57</td>
<td>Df Model:</td>
<td>11</td>
</tr>
</tbody>
</table>

#### Mean Model

| coef  | std err | t     | P>|t|   | 95.0% Conf. Int. |
|-------|---------|-------|-------|------------------|
| Const | 0.0312  | 1.861e-02 | 1.678 | 9.342e-02 [-5.254e-03, 6.769e-02] |
| CPIFESL[3] | -0.0730 | 3.873e-02 | -1.885 | 5.945e-02 [-0.149, 2.910e-03] |
| CPIFESL[12] | -0.0236 | 1.316e-02 | -1.791 | 7.330e-02 [-4.935e-02, 2.224e-03] |

#### Volatility Model

| coef     | std err | t     | P>|t|   | 95.0% Conf. Int. |
|----------|---------|-------|-------|------------------|
| omega    | 8.7359e-03 | 2.063e-03 | 4.235 | 2.283e-05 [4.693e-03, 1.278e-02] |
| alpha[1] | 0.1715  | 5.064e-02 | 3.386 | 7.086e-04 [7.222e-02, 0.271] |
| alpha[2] | 0.2202  | 6.394e-02 | 3.444 | 5.742e-02 [9.486e-02, 0.345] |
| alpha[3] | 0.1547  | 6.327e-02 | 2.445 | 1.447e-02 [3.071e-02, 0.279] |
| alpha[4] | 0.2117  | 7.287e-02 | 2.905 | 3.675e-02 [6.885e-02, 0.355] |
| alpha[5] | 0.1959  | 7.852e-02 | 2.494 | 1.262e-02 [4.197e-02, 0.350] |

#### Distribution

| coef     | std err | t     | P>|t|   | 95.0% Conf. Int. |
|----------|---------|-------|-------|------------------|

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.

```python
from collections import OrderedDict
import arch.data.wti

import arch.data.wti

crude = arch.data.wti.load()

# crude_ret = 100 * crude.DCOILWTICO.dropna().pct_change().dropna()
```

(continues on next page)
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))))
print(lls)
params = pd.DataFrame(
    OrderedDict((('normal', res_normal.params), ('t', res_t.params),
                  ('skewt', res_skewt.params))))
print(params)

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

<table>
<thead>
<tr>
<th></th>
<th>normal</th>
<th>t</th>
<th>skewt</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha[1]</td>
<td>0.085627</td>
<td>0.064980</td>
<td>0.064889</td>
</tr>
<tr>
<td>beta[1]</td>
<td>0.909098</td>
<td>0.927950</td>
<td>0.928215</td>
</tr>
<tr>
<td>lambda</td>
<td>NaN</td>
<td>NaN</td>
<td>-0.036986</td>
</tr>
<tr>
<td>mu</td>
<td>0.046682</td>
<td>0.056438</td>
<td>0.040928</td>
</tr>
<tr>
<td>nu</td>
<td>NaN</td>
<td>6.178652</td>
<td>6.186604</td>
</tr>
<tr>
<td>omega</td>
<td>0.055806</td>
<td>0.048516</td>
<td>0.047683</td>
</tr>
</tbody>
</table>

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.

[17]: 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], 1)
df.columns = ['Std Resids', 'Unit Variance Resids']
subplot = df.plot(kind='kde', xlim=(-4, 4))
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 models 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-based 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

\[
\begin{align*}
\tau_t &= \mu + \epsilon_t \\
\epsilon_t &= \sigma_t e_t \\
\sigma_t^2 &= \omega + \alpha \epsilon_{t-1}^2 + \beta \sigma_{t-1}^2 \\
e_t &\sim N(0,1)
\end{align*}
\]
In code this model can be constructed using data from the S&P 500 using

```python
from arch import arch_model
import pandas_datareader.data as web
import datetime as dt
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.

```python
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

$$
\sigma^2_{t+1} = \omega + \alpha \epsilon^2_{t} + \beta \sigma^2_{t-1} \tag{1.8}
$$

$$
\sigma^2_{t+h} = \omega + \alpha E_t[\epsilon^2_{t+h-1}] + \beta E_t[\sigma^2_{t+h-1}] \quad h \geq 2 \tag{1.9}
$$

$$
= \omega + (\alpha + \beta) E_t[\sigma^2_{t+h-1}] \quad h \geq 2 \tag{1.10}
$$

```python
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, $\epsilon_{t+h}$. 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

$$
\sigma^2_{t+h,b} = \omega + \alpha \epsilon^2_{t+h-1,b} + \beta \sigma^2_{t+h-1,b} \tag{1.11}
$$

$$
\epsilon_{t+h,b} = \epsilon_{t+h,b} \sqrt{\sigma^2_{t+h,b}} \tag{1.12}
$$

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

The final variance forecasts are then computed using the $B$ simulations

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

1.3. Forecasting
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

\[ \hat{e}_t = \frac{r_t - \hat{\mu}}{\hat{\sigma}_t} \]  

(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.

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

which returns
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.

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

which produces

<table>
<thead>
<tr>
<th>Date</th>
<th>h.1</th>
<th>h.2</th>
<th>h.3</th>
<th>h.4</th>
<th>h.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-12-28</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2009-12-29</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2009-12-30</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2009-12-31</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2010-01-04</td>
<td>0.739303</td>
<td>0.741100</td>
<td>0.744529</td>
<td>0.746940</td>
<td>0.752688</td>
</tr>
<tr>
<td>2010-01-05</td>
<td>0.695349</td>
<td>0.702488</td>
<td>0.706812</td>
<td>0.713342</td>
<td>0.721629</td>
</tr>
<tr>
<td>2010-01-06</td>
<td>0.649343</td>
<td>0.654048</td>
<td>0.664055</td>
<td>0.672742</td>
<td>0.681263</td>
</tr>
</tbody>
</table>

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

```python
class arch.univariate.base.ARCHModelForecast(index, mean, variance, residual_variance, simulated_paths=None, simulated_variances=None, simulated_residual_variances=None, simulated_residuals=None, align='origin')
```

Container for forecasts from an ARCH Model

**Parameters**

- `index(list, ndarray)` -
- `mean(ndarray)` -
- `variance(ndarray)` -
- `residual_variance(ndarray)` -
- `simulated_paths(ndarray, optional)` -
- `simulated_variances(ndarray, optional)` -
- `simulated_residual_variances(ndarray, optional)` -
- `simulated_residuals(ndarray, optional)` -
- `align('origin', 'target')` -

1.3. Forecasting
mean
  Forecast values for the conditional mean of the process
  Type DataFrame

variance
  Forecast values for the conditional variance of the process
  Type DataFrame

residual_variance
  Forecast values for the conditional variance of the residuals
  Type DataFrame

class arch.univariate.base.ARCHModelForecastSimulation(values, residuals, variances, residual_variances)
  Container for a simulation or bootstrap-based forecasts from an ARCH Model

  Parameters
  • values –
  • residuals –
  • variances –
  • residual_variances –

values
  Simulated values of the process
  Type DataFrame

residuals
  Simulated residuals used to produce the values
  Type DataFrame

variances
  Simulated variances of the values
  Type DataFrame

residual_variances
  Simulated variance of the residuals
  Type DataFrame

1.4 Volatility Forecasting

This setup code is required to run in an IPython notebook

[1]:
   import warnings
   warnings.simplefilter('ignore')

   %matplotlib inline
   import seaborn
   seaborn.set_style('darkgrid')
1.4.1 Data

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

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.

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 DataFrames 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.

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

<table>
<thead>
<tr>
<th>Date</th>
<th>h.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-12-27</td>
<td>NaN</td>
</tr>
<tr>
<td>2018-12-28</td>
<td>NaN</td>
</tr>
<tr>
<td>2018-12-31</td>
<td>0.056353</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date</th>
<th>h.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-12-27</td>
<td>NaN</td>
</tr>
<tr>
<td>2018-12-28</td>
<td>NaN</td>
</tr>
<tr>
<td>2018-12-31</td>
<td>3.59647</td>
</tr>
</tbody>
</table>

Values that are not computed are nan-filled.

### 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.

```
res = am.fit(last_obs='2011-1-1', update_freq=5)
forecasts = res.forecast(horizon=5)
print(forecasts.variance.dropna().head())
```

<table>
<thead>
<tr>
<th>Date</th>
<th>h.1</th>
<th>h.2</th>
<th>h.3</th>
<th>h.4</th>
<th>h.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-12-27</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2018-12-28</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2018-12-31</td>
<td>3.59647</td>
<td>3.568502</td>
<td>3.540887</td>
<td>3.513621</td>
<td>3.4867</td>
</tr>
</tbody>
</table>

(continues on next page)
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`.

```python
[9]: 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[i + end_loc - 1]
        forecasts[fcast.name] = fcast
    print()
    print(pd.DataFrame(forecasts).T)
```

<table>
<thead>
<tr>
<th>Date</th>
<th>h.1</th>
<th>h.2</th>
<th>h.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-12-31</td>
<td>0.615314</td>
<td>0.621743</td>
<td>0.628133</td>
</tr>
<tr>
<td>2010-01-04</td>
<td>0.751747</td>
<td>0.757343</td>
<td>0.762905</td>
</tr>
<tr>
<td>2010-01-05</td>
<td>0.710453</td>
<td>0.716315</td>
<td>0.722142</td>
</tr>
<tr>
<td>2010-01-06</td>
<td>0.666244</td>
<td>0.672346</td>
<td>0.678411</td>
</tr>
<tr>
<td>2010-01-07</td>
<td>0.634424</td>
<td>0.640706</td>
<td>0.646949</td>
</tr>
<tr>
<td>2010-01-08</td>
<td>0.600109</td>
<td>0.606595</td>
<td>0.613040</td>
</tr>
<tr>
<td>2010-01-11</td>
<td>0.565514</td>
<td>0.572212</td>
<td>0.578869</td>
</tr>
<tr>
<td>2010-01-12</td>
<td>0.599561</td>
<td>0.606051</td>
<td>0.612501</td>
</tr>
<tr>
<td>2010-01-13</td>
<td>0.608309</td>
<td>0.614748</td>
<td>0.621148</td>
</tr>
<tr>
<td>2010-01-14</td>
<td>0.575065</td>
<td>0.581756</td>
<td>0.588406</td>
</tr>
<tr>
<td>2010-01-15</td>
<td>0.629890</td>
<td>0.636245</td>
<td>0.642561</td>
</tr>
<tr>
<td>2010-01-19</td>
<td>0.695074</td>
<td>0.701042</td>
<td>0.706974</td>
</tr>
<tr>
<td>2010-01-20</td>
<td>0.737154</td>
<td>0.742908</td>
<td>0.748627</td>
</tr>
<tr>
<td>2010-01-21</td>
<td>0.951667</td>
<td>0.958725</td>
<td>0.963255</td>
</tr>
<tr>
<td>2010-01-22</td>
<td>1.253453</td>
<td>1.256401</td>
<td>1.259332</td>
</tr>
<tr>
<td>2010-01-25</td>
<td>1.178691</td>
<td>1.182043</td>
<td>1.185374</td>
</tr>
<tr>
<td>2010-01-26</td>
<td>1.112205</td>
<td>1.115886</td>
<td>1.119545</td>
</tr>
<tr>
<td>2010-01-27</td>
<td>1.051295</td>
<td>1.055327</td>
<td>1.059335</td>
</tr>
<tr>
<td>2010-01-28</td>
<td>1.085678</td>
<td>1.089512</td>
<td>1.093324</td>
</tr>
<tr>
<td>2010-01-29</td>
<td>1.085786</td>
<td>1.089593</td>
<td>1.093378</td>
</tr>
</tbody>
</table>
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.

```python
import numpy as np
import pandas as pd

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(last_obs=i + end_loc, disp='off')
    temp = res.forecast(horizon=3).variance
    fcast = temp.iloc[i + end_loc - 1]
    forecasts[fcast.name] = fcast
print()
print(pd.DataFrame(forecasts).T)
```

<table>
<thead>
<tr>
<th></th>
<th>h.1</th>
<th>h.2</th>
<th>h.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-12-31</td>
<td>0.615314</td>
<td>0.621743</td>
<td>0.628133</td>
</tr>
<tr>
<td>2010-01-04</td>
<td>0.751723</td>
<td>0.757321</td>
<td>0.762885</td>
</tr>
<tr>
<td>2010-01-05</td>
<td>0.709956</td>
<td>0.715791</td>
<td>0.721591</td>
</tr>
<tr>
<td>2010-01-06</td>
<td>0.666057</td>
<td>0.672146</td>
<td>0.678197</td>
</tr>
<tr>
<td>2010-01-07</td>
<td>0.634503</td>
<td>0.640776</td>
<td>0.647011</td>
</tr>
<tr>
<td>2010-01-08</td>
<td>0.600417</td>
<td>0.606893</td>
<td>0.613329</td>
</tr>
<tr>
<td>2010-01-11</td>
<td>0.565684</td>
<td>0.572369</td>
<td>0.579014</td>
</tr>
<tr>
<td>2010-01-12</td>
<td>0.599963</td>
<td>0.606438</td>
<td>0.612874</td>
</tr>
<tr>
<td>2010-01-13</td>
<td>0.608558</td>
<td>0.614982</td>
<td>0.621366</td>
</tr>
<tr>
<td>2010-01-14</td>
<td>0.575020</td>
<td>0.581639</td>
<td>0.588217</td>
</tr>
<tr>
<td>2010-01-15</td>
<td>0.629696</td>
<td>0.635989</td>
<td>0.642244</td>
</tr>
<tr>
<td>2010-01-19</td>
<td>0.694735</td>
<td>0.700656</td>
<td>0.706541</td>
</tr>
<tr>
<td>2010-01-20</td>
<td>0.736509</td>
<td>0.742193</td>
<td>0.747842</td>
</tr>
<tr>
<td>2010-01-21</td>
<td>0.952751</td>
<td>0.957245</td>
<td>0.961713</td>
</tr>
<tr>
<td>2010-01-22</td>
<td>1.251145</td>
<td>1.254050</td>
<td>1.256936</td>
</tr>
<tr>
<td>2010-01-25</td>
<td>1.176864</td>
<td>1.180162</td>
<td>1.183441</td>
</tr>
<tr>
<td>2010-01-26</td>
<td>1.110848</td>
<td>1.114497</td>
<td>1.118124</td>
</tr>
<tr>
<td>2010-01-27</td>
<td>1.050102</td>
<td>1.054077</td>
<td>1.058028</td>
</tr>
<tr>
<td>2010-01-28</td>
<td>1.084669</td>
<td>1.088454</td>
<td>1.092216</td>
</tr>
<tr>
<td>2010-01-29</td>
<td>1.085003</td>
<td>1.088783</td>
<td>1.092541</td>
</tr>
</tbody>
</table>

### 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*.

```python
# 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: 44, Neg. LLF: 6827.96641441215
Iteration: 10, Func. Count: 84, Neg. LLF: 6822.8830945206155
Optimization terminated successfully. (Exit mode 0)
Current function value: 6822.882823472137
Iterations: 13
Function evaluations: 106
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.

```
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])
```

```[12]: (735967.0, 736328.0)```
import arch

c = data['Open']
res = archеча.EngleRiskModel(c)

horizon = 5
method = 'simulation'
sims = res.forecast(horizon=horizon, method=method)
sims = res.forecast(horizon=horizon, method=method)

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, res.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()

import seaborn as sns
sns.boxplot(data=sims.variances[-1])

<matplotlib.axes._subplots.AxesSubplot at 0x7fe173e9c0d0>
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.

```python
[15]: 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')
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,

\[ \text{VaR}_{t+1|t} = -\mu_{t+1|t} - \sigma_{t+1|t} q_\alpha \]

where \( q_\alpha \) is the \( \alpha \) 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.

```python
16: 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.

```python
17: forecasts = res.forecast(start='2018-1-1')
cond_mean = forecasts.mean['2018:']
cond_var = forecasts.variance['2018:']
q = am.distribution.ppf([0.01, 0.05], res.params[-2:])
print(q)
[-2.64484937 -1.64965928]
```

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

```python
[19]: std_rets = (returns['2017'] - res.params['mu']) / res.conditional_volatility
    std_rets = std_rets.dropna()
    q = std_rets.quantile([.01, .05])
    print(q)

         0.01    -2.668272
         0.05    -1.723353
dtype: float64
```

```python
[20]: 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

```python
[1]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn
from arch.univariate import GARCH, ConstantMean, Normal
seaborn.set_style('darkgrid')

[2]: seaborn.mpl.rcParams['figure.figsize'] = (10.0, 6.0)
seaborn.mpl.rcParams['savefig.dpi'] = 90
seaborn.mpl.rcParams['font.family'] = 'sans-serif'
seaborn.mpl.rcParams['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.

```python
[3]: import arch.data.nasdaq

data = arch.data.nasdaq.load()
nasdaq = data['Adj Close']
print(nasdaq.head())

<table>
<thead>
<tr>
<th>Date</th>
<th>Adj Close</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999-01-04</td>
<td>2208.050049</td>
</tr>
<tr>
<td>1999-01-05</td>
<td>2251.270020</td>
</tr>
<tr>
<td>1999-01-06</td>
<td>2320.860107</td>
</tr>
<tr>
<td>1999-01-07</td>
<td>2326.090088</td>
</tr>
<tr>
<td>1999-01-08</td>
<td>2344.409912</td>
</tr>
</tbody>
</table>
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

```python
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 converge since the scale of the volatility intercept is much closer to the scale of the other parameters in the model.

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

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

Fitting the model is standard.
```
res = mod.fit(disp='off')
```

```plaintext
res
```

```
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: Wed, Aug 28 2019 Df Residuals: 5025
Time: 09:43:01 Df Model: 5
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
```

```
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: 0x7f35fb9a19d0
GJR-GARCH models support analytical forecasts, which is the default. The forecasts are produced for all of 2017 using the estimated model parameters.
```

```
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
```

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 $\infty$, the simulation-based forecasts will converge to the analytical values above.

1.6. Volatility Scenarios
```python
sim_forecasts = res.forecast(start='1-1-2017', method='simulation', horizon=10)
print(sim_forecasts.residual_variance.dropna().head())
```

<table>
<thead>
<tr>
<th></th>
<th>h.01</th>
<th>h.02</th>
<th>h.03</th>
<th>h.04</th>
<th>h.05</th>
<th>h.06</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017-01-03</td>
<td>0.623295</td>
<td>0.637251</td>
<td>0.647817</td>
<td>0.663746</td>
<td>0.673404</td>
<td>0.687952</td>
</tr>
<tr>
<td>2017-01-04</td>
<td>0.599455</td>
<td>0.617539</td>
<td>0.635838</td>
<td>0.649695</td>
<td>0.659733</td>
<td>0.667267</td>
</tr>
<tr>
<td>2017-01-05</td>
<td>0.567297</td>
<td>0.583415</td>
<td>0.597571</td>
<td>0.613065</td>
<td>0.621790</td>
<td>0.636180</td>
</tr>
<tr>
<td>2017-01-06</td>
<td>0.542506</td>
<td>0.555688</td>
<td>0.570280</td>
<td>0.585426</td>
<td>0.595551</td>
<td>0.608487</td>
</tr>
<tr>
<td>2017-01-09</td>
<td>0.515452</td>
<td>0.528771</td>
<td>0.542658</td>
<td>0.559684</td>
<td>0.580434</td>
<td>0.594855</td>
</tr>
</tbody>
</table>

```python
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.

```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())
```

<table>
<thead>
<tr>
<th></th>
<th>h.01</th>
<th>h.02</th>
<th>h.03</th>
<th>h.04</th>
<th>h.05</th>
<th>h.06</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017-01-03</td>
<td>0.623295</td>
<td>0.685911</td>
<td>0.745202</td>
<td>0.821112</td>
<td>0.886289</td>
<td>0.966737</td>
</tr>
<tr>
<td>2017-01-04</td>
<td>0.599455</td>
<td>0.668181</td>
<td>0.743119</td>
<td>0.811486</td>
<td>0.877539</td>
<td>0.936587</td>
</tr>
<tr>
<td>2017-01-05</td>
<td>0.567297</td>
<td>0.629195</td>
<td>0.691225</td>
<td>0.758891</td>
<td>0.816663</td>
<td>0.893986</td>
</tr>
<tr>
<td>2017-01-06</td>
<td>0.542506</td>
<td>0.596301</td>
<td>0.656603</td>
<td>0.721505</td>
<td>0.778286</td>
<td>0.849680</td>
</tr>
<tr>
<td>2017-01-09</td>
<td>0.515452</td>
<td>0.567086</td>
<td>0.622224</td>
<td>0.689831</td>
<td>0.775048</td>
<td>0.845656</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>h.07</th>
<th>h.08</th>
<th>h.09</th>
<th>h.10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017-01-03</td>
<td>0.970796</td>
<td>0.977504</td>
<td>0.982202</td>
<td>0.992547</td>
</tr>
<tr>
<td>2017-01-04</td>
<td>0.955295</td>
<td>0.965540</td>
<td>0.966432</td>
<td>0.974248</td>
</tr>
<tr>
<td>2017-01-05</td>
<td>0.905952</td>
<td>0.915208</td>
<td>0.930777</td>
<td>0.938636</td>
</tr>
</tbody>
</table>
```
(continues on next page)
### 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.

[9]:
```python
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]
```

```python
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())
```

<table>
<thead>
<tr>
<th>Date</th>
<th>h.01</th>
<th>h.02</th>
<th>h.03</th>
<th>h.04</th>
<th>h.05</th>
<th>h.06</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-01-03</td>
<td>0.623295</td>
<td>0.676081</td>
<td>0.734322</td>
<td>0.779325</td>
<td>0.828189</td>
<td>0.898202</td>
</tr>
<tr>
<td>2017-01-04</td>
<td>0.599455</td>
<td>0.645237</td>
<td>0.697133</td>
<td>0.750169</td>
<td>0.816280</td>
<td>0.888417</td>
</tr>
<tr>
<td>2017-01-05</td>
<td>0.567297</td>
<td>0.610493</td>
<td>0.665995</td>
<td>0.722954</td>
<td>0.777860</td>
<td>0.840369</td>
</tr>
<tr>
<td>2017-01-06</td>
<td>0.542506</td>
<td>0.597387</td>
<td>0.644534</td>
<td>0.691387</td>
<td>0.741206</td>
<td>0.783319</td>
</tr>
<tr>
<td>2017-01-09</td>
<td>0.515452</td>
<td>0.561312</td>
<td>0.611026</td>
<td>0.647824</td>
<td>0.700559</td>
<td>0.757398</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date</th>
<th>h.07</th>
<th>h.08</th>
<th>h.09</th>
<th>h.10</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-01-03</td>
<td>0.623295</td>
<td>0.676081</td>
<td>0.734322</td>
<td>0.779325</td>
</tr>
<tr>
<td>2017-01-04</td>
<td>0.599455</td>
<td>0.645237</td>
<td>0.697133</td>
<td>0.750169</td>
</tr>
<tr>
<td>2017-01-05</td>
<td>0.567297</td>
<td>0.610493</td>
<td>0.665995</td>
<td>0.722954</td>
</tr>
<tr>
<td>2017-01-06</td>
<td>0.542506</td>
<td>0.597387</td>
<td>0.644534</td>
<td>0.691387</td>
</tr>
<tr>
<td>2017-01-09</td>
<td>0.515452</td>
<td>0.561312</td>
<td>0.611026</td>
<td>0.647824</td>
</tr>
</tbody>
</table>
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.

```python
[10]: import pandas as pd
def = 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]
], 1)
df.columns = ['Analytic', 'Simulation', 'Scenario Sim', 'Bootstrap Scenario']
# Plot annualized vol
subplot = np.sqrt(252 * df).plot(legend=False)
legend = subplot.legend(frameon=False)
```

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.

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

(continues on next page)
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.
1.7 Mean Models

All ARCH models start by specifying a mean model.

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZeroMean([y, hold_back, volatility, ...])</td>
<td>Model with zero conditional mean estimation and simulation</td>
</tr>
<tr>
<td>ConstantMean([y, hold_back, volatility, ...])</td>
<td>Constant mean model estimation and simulation</td>
</tr>
</tbody>
</table>
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ARX(y, x, lags, constant, hold_back, ...)</code></td>
<td>Autoregressive model with optional exogenous regressors estimation and simulation</td>
</tr>
<tr>
<td><code>HARX(y, x, lags, constant, use_rotated, ...)</code></td>
<td>Heterogeneous Autoregression (HAR), with optional exogenous regressors, model estimation and simulation</td>
</tr>
<tr>
<td><code>LS(y, x, constant, hold_back, rescale)</code></td>
<td>Least squares model estimation and simulation</td>
</tr>
</tbody>
</table>

### 1.7.1 arch.univariate.ZeroMean

**Class** arch.univariate.ZeroMean

Model with zero conditional mean estimation and simulation

**Parameters**

- `y (ndarray, Series)` — nobs element vector containing the dependent variable
- `hold_back (int)` — 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, optional)` — Volatility process to use in the model
- `distribution (Distribution, optional)` — Error distribution to use in the model
- `rescale (bool, optional)` — 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**

```python
>>> 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.
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>constraints()</code></td>
<td>Construct linear constraint arrays for use in non-linear optimization</td>
</tr>
<tr>
<td><code>fit([update_freq, disp, starting_values, ...])</code></td>
<td>Fits the model given a nobs by 1 vector of sigma2 values</td>
</tr>
<tr>
<td><code>fix(params[, first_obs, last_obs])</code></td>
<td>Allows an ARCHModelFixedResult to be constructed from fixed parameters.</td>
</tr>
<tr>
<td><code>forecast(params[, horizon, start, align, ...])</code></td>
<td>Construct forecasts from estimated model</td>
</tr>
<tr>
<td><code>parameter_names()</code></td>
<td>List of parameters names</td>
</tr>
<tr>
<td><code>resids(params[, y, regressors])</code></td>
<td>Compute model residuals</td>
</tr>
<tr>
<td><code>simulate(params, nobs[, burn, ...])</code></td>
<td>Simulated data from a zero mean model</td>
</tr>
<tr>
<td><code>starting_values()</code></td>
<td>Returns starting values for the mean model, often the same as the values returned from fit</td>
</tr>
</tbody>
</table>

**arch.univariate.ZeroMean.bounds**

ZeroMean.bounds()

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, backcast=None, robust=True)

Computes parameter covariances using numerical derivatives.

- **Parameters**
  - `params (ndarray)` – Model parameters
  - `backcast (float)` – Value to use for pre-sample observations
  - `robust (bool, optional)` – Flag indicating whether to use robust standard errors (True) or classic MLE (False)

**arch.univariate.ZeroMean.constraints**

ZeroMean.constraints()

Construct linear constraint arrays for use in non-linear optimization

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

- **Notes**
  Parameters satisfy a.dot(parameters) - b >= 0
arch.univariate.ZeroMean.fit

ZeroMean.fit(update_freq=1, disp='final', starting_values=None, cov_type='robust', show_warning=True, first_obs=None, last_obs=None, tol=None, options=None, backcast=None)

Fits the model given a nobs by 1 vector of sigma2 values

Parameters

- **update_freq (int, optional)** – Frequency of iteration updates. Output is generated every update_freq iterations. Set to 0 to disable iterative output.
- **disp (str)** – Either ‘final’ to print optimization result or ‘off’ to display nothing
- **starting_values (ndarray, optional)** – Array of starting values to use. If not provided, starting values are constructed by the model components.
- **cov_type (str, optional)** – Estimation method of parameter covariance. Supported options are ‘robust’, which does not assume the Information Matrix Equality holds and ‘classic’ which does. In the ARCH literature, ‘robust’ corresponds to Bollerslev-Wooldridge covariance estimator.
- **show_warning (bool, optional)** – Flag indicating whether convergence warnings should be shown.
- **first_obs ((int, str, datetime, Timestamp))** – First observation to use when estimating model
- **last_obs ((int, str, datetime, Timestamp))** – Last observation to use when estimating model
- **tol (float, optional)** – Tolerance for termination.
- **options (dict, optional)** – Options to pass to scipy.optimize.minimize. Valid entries include ‘ftol’, ‘eps’, ‘disp’, and ‘maxiter’.
- **backcast (float, optional)** – Value to use as backcast. Should be measure $\sigma^2_0$ 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, first_obs=None, last_obs=None)

Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

- **params ((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, Timestamp}) – First observation to use when fixing model
- **last_obs**({int, str, datetime, Timestamp}) – 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, horizon=1, start=None, method='analytic', simulations=1000, rng=None, random_state=None)

Construct forecasts from estimated model

**Parameters**

- **params**((ndarray, Series), optional) – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.
- **horizon**(int, optional) – Number of steps to forecast
- **start**((int, datetime, Timestamp, str), optional) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in ‘1945-01-01’.
- **align**(str, optional) – 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**({'analytic', 'simulation', 'bootstrap'}) – Method to use when producing the forecast. The default is analytic. The method only affects the variance forecast generation. Not all volatility models support all methods. In particular, volatility models that do not evolve in squares such as EGARCH or TARCH do not support the ‘analytic’ method for horizons > 1.
- **simulations**(int) – Number of simulations to run when computing the forecast using either simulation or bootstrap.
- **rng**(callable, optional) – 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**(RandomState, optional) – NumPy RandomState instance to use when method is ‘bootstrap’

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

**Return type** ARCHModelForecast
Examples

```python
>>> 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 avail-
able. 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 of parameters names

Returns names – List of variable names for the mean model

Return type list (str)

`arch.univariate.ZeroMean.resids`

ZeroMean.resids(params, y=None, regressors=None)
Compute model residuals

Parameters

• params (ndarray) – Model parameters
• y (ndarray, optional) – Alternative values to use when computing model residuals
• regressors (ndarray, optional) – Alternative regressor values to use when computing model residuals

Returns resid – Model residuals

Return type ndarray

1.7. Mean Models  53
arch Documentation, Release 4.9.1+4.g81ceedd

arch.univariate.ZeroMean.simulate

ZeroMean.simulate(params, nobs, burn=500, initial_value=None, x=None, initial_value_vol=None)
Simulated data from a zero mean model

Parameters

• params (ndarray, DataFrame) – 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, optional) – Number of values to simulate to initialize the model and remove dependence on initial values.

• initial_value (None) – This value is not used.

• x (None) – This value is not used.

• initial_value_vol (ndarray, float, optional) – 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 DataFrame

Examples

Basic data simulation with no mean and constant volatility

```python
>>> from arch.univariate import ZeroMean
>>> zm = ZeroMean()
>>> sim_data = zm.simulate([1.0], 1000)
```

Simulating data with a non-trivial volatility process

```python
>>> 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()
Returns starting values for the mean model, often the same as the values returned from fit

Returns sv – Starting values

Return type ndarray

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distribution</td>
<td>Set or gets the error distribution</td>
</tr>
<tr>
<td>num_params</td>
<td>Returns the number of parameters</td>
</tr>
</tbody>
</table>

Continued on next page
Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>volatility</td>
<td>Set or gets the volatility process</td>
</tr>
<tr>
<td>x</td>
<td>Gets the value of the exogenous regressors in the model</td>
</tr>
<tr>
<td>y</td>
<td>Returns the dependent variable</td>
</tr>
</tbody>
</table>

**arch.univariate.ZeroMean.distribution**

ZeroMean.distribution

Set or gets the error distribution

Distributions must be a subclass of Distribution

**arch.univariate.ZeroMean.num_params**

ZeroMean.num_params

Returns the number of parameters

**arch.univariate.ZeroMean.volatility**

ZeroMean.volatility

Set or gets the volatility process

Volatility processes must be a subclass of VolatilityProcess

**arch.univariate.ZeroMean.x**

ZeroMean.x

Gets the value of the exogenous regressors in the model

**arch.univariate.ZeroMean.y**

ZeroMean.y

Returns the dependent variable

### 1.7.2 arch.univariate.ConstantMean

class arch.univariate.ConstantMean(y=None, hold_back=None, volatility=None, distribution=None, rescale=None)

Constant mean model estimation and simulation.

**Parameters**

- **y** (ndarray, Series) – nobs element vector containing the dependent variable
- **hold_back** (int) – 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, optional) – Volatility process to use in the model
- **distribution** (Distribution, optional) – Error distribution to use in the model
- **rescale** *(bool, optional)* – 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, then y is rescaled and the new scale is reported in the estimation results.

### Examples

```python
g>>> import numpy as np
>>> from arch.univariate import ConstantMean
g>>> y = np.random.randn(100)
g>>> cm = ConstantMean(y)
g>>> 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, ...])` – Fits the model given a nobs by 1 vector of sigma2 values.
- `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.compute_param_cov

ConstantMean.\texttt{compute\_param\_cov}(\texttt{params}, \texttt{backcast=None}, \texttt{robust=True})

Computes parameter covariances using numerical derivatives.

\textbf{Parameters}

- \texttt{params} (\texttt{ndarray}) – Model parameters
- \texttt{backcast} (\texttt{float}) – Value to use for pre-sample observations
- \texttt{robust} (\texttt{bool}, \texttt{optional}) – Flag indicating whether to use robust standard errors (True) or classic MLE (False)

arch.univariate.ConstantMean.constraints

ConstantMean.\texttt{constraints}()

Construct linear constraint arrays for use in non-linear optimization

\textbf{Returns}

- \texttt{a} (\texttt{ndarray}) – Number of constraints by number of parameters loading array
- \texttt{b} (\texttt{ndarray}) – Number of constraints array of lower bounds

\textbf{Notes}

Parameters satisfy \( a \cdot \text{parameters} - b \geq 0 \)

arch.univariate.ConstantMean.fit

ConstantMean.\texttt{fit}(\texttt{update\_freq=1}, \texttt{disp='final'}, \texttt{starting\_values=None}, \texttt{cov\_type='robust'}, \texttt{show\_warning=True}, \texttt{first\_obs=None}, \texttt{last\_obs=None}, \texttt{tol=None}, \texttt{options=None}, \texttt{backcast=None})

Fits the model given a nobs by 1 vector of sigma2 values

\textbf{Parameters}

- \texttt{update\_freq} (\texttt{int}, \texttt{optional}) – Frequency of iteration updates. Output is generated every \texttt{update\_freq} iterations. Set to 0 to disable iterative output.
- \texttt{disp} (\texttt{str}) – Either ‘final’ to print optimization result or ‘off’ to display nothing
- \texttt{starting\_values} (\texttt{ndarray}, \texttt{optional}) – Array of starting values to use. If not provided, starting values are constructed by the model components.
- \texttt{cov\_type} (\texttt{str}, \texttt{optional}) – Estimation method of parameter covariance. Supported options are ‘robust’, which does not assume the Information Matrix Equality holds and ‘classic’ which does. In the ARCH literature, ‘robust’ corresponds to Bollerslev-Wooldridge covariance estimator.
- \texttt{show\_warning} (\texttt{bool}, \texttt{optional}) – Flag indicating whether convergence warnings should be shown.
- \texttt{first\_obs} ((\texttt{int}, \texttt{str}, \texttt{datetime}, \texttt{Timestamp})) – First observation to use when estimating model
- \texttt{last\_obs} ((\texttt{int}, \texttt{str}, \texttt{datetime}, \texttt{Timestamp})) – Last observation to use when estimating model
• **tol** *(float, optional)* – Tolerance for termination.
• **options** *(dict, optional)* – Options to pass to `scipy.optimize.minimize`. Valid entries include ‘ftol’, ‘eps’, ‘disp’, and ‘maxiter’.
• **backcast** *(float, optional)* – Value to use as backcast. Should be measure $\sigma_0^2$ 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, first_obs=None, last_obs=None)`

Allows an ARCHModelFixedResult to be constructed from fixed parameters.

**Parameters**

- **params** *(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, Timestamp)* – First observation to use when fixing model
- **last_obs** *(int, str, datetime, Timestamp)* – 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, horizon=1, start=None, align='origin', method='analytic', simulations=1000, rng=None, random_state=None)`

Construct forecasts from estimated model

**Parameters**

- **params** *(ndarray, Series), optional* – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.
- **horizon** *(int, optional)* – Number of steps to forecast
• **start** ((int, datetime, Timestamp, str), optional) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in ‘1945-01-01’.

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

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

• **rng** (callable, optional) – 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** (RandomState, optional) – NumPy RandomState instance to use when method is ‘bootstrap’

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

**Return type** ARCHModelForecast

### Examples

```python
>>> 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 of parameters names

**Returns** names – List of variable names for the mean model

**Return type** list (str)

arch.univariate.ConstantMean.resids

ConstantMean.resids (params, y=None, regressors=None)
Compute model residuals

**Parameters**

- **params** (ndarray) – Model parameters
- **y** (ndarray, optional) – Alternative values to use when computing model residuals
- **regressors** (ndarray, optional) – Alternative regressor values to use when computing model residuals

**Returns** resids – Model residuals

**Return type** ndarray

arch.univariate.ConstantMean.simulate

ConstantMean.simulate(params, nobs, burn=500, initial_value=None, x=None, initial_value_vol=None)
Simulated data from a constant mean model

**Parameters**

- **params** (ndarray) – 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, optional) – Number of values to simulate to initialize the model and remove dependence on initial values.
- **initial_value** (None) – This value is not used.
- **x** (None) – This value is not used.
- **initial_value_vol** (ndarray, float, optional) – 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** 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()
Returns starting values for the mean model, often the same as the values returned from fit

Returns sv – Starting values

Return type ndarray

Properties

<table>
<thead>
<tr>
<th>property</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distribution</td>
<td>Set or gets the error distribution</td>
</tr>
<tr>
<td>num_params</td>
<td>Returns the number of parameters</td>
</tr>
<tr>
<td>volatility</td>
<td>Set or gets the volatility process</td>
</tr>
<tr>
<td>x</td>
<td>Gets the value of the exogenous regressors in the model</td>
</tr>
<tr>
<td>y</td>
<td>Returns the dependent variable</td>
</tr>
</tbody>
</table>

arch.univariate.ConstantMean.distribution

ConstantMean.distribution
Set or gets the error distribution
Distributions must be a subclass of Distribution

arch.univariate.ConstantMean.num_params

ConstantMean.num_params
Returns the number of parameters

arch.univariate.ConstantMean.volatility

ConstantMean.volatility
Set or gets the volatility process
Volatility processes must be a subclass of VolatilityProcess

arch.univariate.ConstantMean.x

ConstantMean.x
Gets the value of the exogenous regressors in the model
arch.univariate.ConstantMean.y

ConstantMean.y
Returns the dependent variable

1.7.3 arch.univariate.ARX

class arch.univariate.ARX(y=None, x=None, lags=None, constant=True, hold_back=None, volatility=None, distribution=None, rescale=None)

Autoregressive model with optional exogenous regressors estimation and simulation

Parameters

- y (ndarray, Series) – nobs element vector containing the dependent variable
- x (ndarray, DataFrame, optional) – nobs by k element array containing exogenous regressors
- lags (scalar, 1-d array, optional) – 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, optional) – Flag whether the model should include a constant
- hold_back (int) – 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, optional) – 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

```python
>>> 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

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

Notes

The AR-X model is described by

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

Methods
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bounds()</code></td>
<td>Construct bounds for parameters to use in non-linear optimization.</td>
</tr>
<tr>
<td><code>compute_param_cov</code></td>
<td>Computes parameter covariances using numerical derivatives.</td>
</tr>
<tr>
<td><code>constraints()</code></td>
<td>Construct linear constraint arrays for use in non-linear optimization.</td>
</tr>
<tr>
<td><code>fit([update_freq, disp, starting_values,...])</code></td>
<td>Fits the model given a nobs by 1 vector of sigma2 values.</td>
</tr>
<tr>
<td><code>fix(params[, first_obs, last_obs])</code></td>
<td>Allows an ARCHModelFixedResult to be constructed from fixed parameters.</td>
</tr>
<tr>
<td><code>forecast(params[, horizon, start, align,...])</code></td>
<td>Construct forecasts from estimated model.</td>
</tr>
<tr>
<td><code>parameter_names()</code></td>
<td>List of parameters names.</td>
</tr>
<tr>
<td><code>resids(params[, y, regressors])</code></td>
<td>Compute model residuals.</td>
</tr>
<tr>
<td><code>simulate(params, nobs[, burn,...])</code></td>
<td>Simulates data from a linear regression, AR or HAR models.</td>
</tr>
<tr>
<td><code>starting_values()</code></td>
<td>Returns starting values for the mean model, often the same as the values returned from fit</td>
</tr>
</tbody>
</table>

**arch.univariate.ARX.bounds**

ARX. `bounds()`

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, backcast=None, robust=True)`

Computes parameter covariances using numerical derivatives.

- **Parameters**
  - `params` (ndarray) – Model parameters
  - `backcast` (float) – Value to use for pre-sample observations
  - `robust` (bool, optional) – Flag indicating whether to use robust standard errors (True) or classic MLE (False)

**arch.univariate.ARX.constraints**

ARX. `constraints()`

Construct linear constraint arrays for use in non-linear optimization

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

- **Notes**

Parameters satisfy `a.dot(parameters) - b >= 0`
arch.univariate.ARX.fit

\texttt{ARX.fit(update\_freq=1, \; disp='final', \; starting\_values=None, \; cov\_type='robust',
\texttt{show\_warning=True, \; first\_obs=None, \; last\_obs=None, \; tol=None, \; options=None, \; back-
\texttt{cast=None})}

Fits the model given a nobs by 1 vector of sigma2 values

Parameters

- \texttt{update\_freq}(\texttt{int, \; optional}) – Frequency of iteration updates. Output is generated every update\_freq iterations. Set to 0 to disable iterative output.
- \texttt{disp}(\texttt{str}) – Either ‘final’ to print optimization result or ‘off’ to display nothing
- \texttt{starting\_values}(\texttt{ndarray, \; optional}) – Array of starting values to use. If not provided, starting values are constructed by the model components.
- \texttt{cov\_type}(\texttt{str, \; optional}) – Estimation method of parameter covariance. Supported options are ‘robust’, which does not assume the Information Matrix Equality holds and ‘classic’ which does. In the ARCH literature, ‘robust’ corresponds to Bollerslev-Wooldridge covariance estimator.
- \texttt{show\_warning}(\texttt{bool, \; optional}) – Flag indicating whether convergence warnings should be shown.
- \texttt{first\_obs}((\texttt{int, \; str, \; datetime, \; Timestamp}) – First observation to use when estimating model
- \texttt{last\_obs}((\texttt{int, \; str, \; datetime, \; Timestamp}) – Last observation to use when estimating model
- \texttt{tol}(\texttt{float, \; optional}) – Tolerance for termination.
- \texttt{options}(\texttt{dict, \; optional}) – Options to pass to \texttt{scipy.optimize.minimize}. Valid entries include ‘ftol’, ‘eps’, ‘disp’, and ‘maxiter’.
- \texttt{backcast}(\texttt{float, \; optional}) – Value to use as backcast. Should be measure $\sigma_0^2$ since model-specific non-linear transformations are applied to value before computing the variance recursions.

Returns \texttt{results} – Object containing model results

Return type \texttt{ARCHModelResult}

Notes

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

Parameters are optimized using SLSQP.

arch.univariate.ARX.fix

\texttt{ARX.fix(params, \; first\_obs=None, \; last\_obs=None)}

Allows an \texttt{ARCHModelFixedResult} to be constructed from fixed parameters.

Parameters

- \texttt{params}(\texttt{\{(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, Timestamp)) – First observation to use when fixing model

• **last_obs** ((int, str, datetime, Timestamp)) – 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, horizon=1, start=None, align='origin', method='analytic', simulations=1000, rng=None, random_state=None)*

Construct forecasts from estimated model

**Parameters**

• **params** ((ndarray, Series), optional) – Alternative parameters to use. If not provided, the parameters estimated when fitting the model are used. Must be identical in shape to the parameters computed by fitting the model.

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

• **start** ([int, datetime, Timestamp, str], optional) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in ‘1945-01-01’.

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

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

• **rng** (callable, optional) – 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** (RandomState, optional) – NumPy RandomState instance to use when method is ‘bootstrap’

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

Return type **ARCHModelForecast**
Examples

```python
>>> 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 of parameters names

**Returns names** – List of variable names for the mean model

**Return type** list (str)

### arch.univariate.ARX.resids

**ARX.resids (params, y=None, regressors=None)**

Compute model residuals

**Parameters**

- **params (ndarray)** – Model parameters
- **y (ndarray, optional)** – Alternative values to use when computing model residuals
- **regressors (ndarray, optional)** – Alternative regressor values to use when computing model residuals

**Returns resids** – Model residuals

**Return type** ndarray
arch.univariate.ARX.simulate

ARX.simulate(params, nob=500, initial_value=None, x=None, initial_value_vol=None)
Simulates data from a linear regression, AR or HAR models

Parameters

- **params** *(ndarray)* – 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, optional)* – Number of values to simulate to initialize the model and remove dependence on initial values.
- **initial_value** *(ndarray, float, optional)* – Either a scalar value or max(lags) array set of initial values to use when initializing the model. If omitted, 0.0 is used.
- **x** *(ndarray, DataFrame, optional)* – nob + burn by k array of exogenous variables to include in the simulation.
- **initial_value_vol** *(ndarray, float, optional)* – 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 DataFrame

Examples

```python
>>> 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, nob=1000)
```

Simulating models with exogenous regressors requires the regressors to have nob + burn data points

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

arch.univariate.ARX.starting_values

ARX.starting_values()
Returns starting values for the mean model, often the same as the values returned from fit
Returns sv – Starting values

Return type ndarray

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distribution</td>
<td>Set or gets the error distribution</td>
</tr>
<tr>
<td>num_params</td>
<td>Returns the number of parameters</td>
</tr>
<tr>
<td>volatility</td>
<td>Set or gets the volatility process</td>
</tr>
<tr>
<td>x</td>
<td>Gets the value of the exogenous regressors in the model</td>
</tr>
<tr>
<td>y</td>
<td>Returns the dependent variable</td>
</tr>
</tbody>
</table>

**arch.univariate.ARX.distribution**

ARX.distribution

Set or gets the error distribution

Distributions must be a subclass of Distribution

**arch.univariate.ARX.num_params**

ARX.num_params

Returns the number of parameters

**arch.univariate.ARX.volatility**

ARX_volatility

Set or gets the volatility process

Volatility processes must be a subclass of VolatilityProcess

**arch.univariate.ARX.x**

ARX_x

Gets the value of the exogenous regressors in the model

**arch.univariate.ARX.y**

ARX_y

Returns the dependent variable

1.7.4 arch.univariate.HARX

class arch.univariate.HARX (y=None, x=None, lags=None, constant=True, use_rotated=False, hold_back=None, volatility=None, distribution=None, rescale=None)

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

Parameters
• **y**({ndarray, Series}) – nobs element vector containing the dependent variable

• **x**({ndarray, DataFrame}, optional) – nobs by k element array containing exogenous regressors

• **lags**({scalar, ndarray}, optional) – Description of lag structure of the HAR. Scalar included all lags between 1 and the value. A 1-d array includes the HAR lags 1:lags[0], 1:lags[1], ... A 2-d array includes the HAR lags of the form lags[0,j]:lags[1,j] for all columns of lags.

• **constant** (bool, optional) – Flag whether the model should include a constant

• **use_rotated** (bool, optional) – Flag indicating to use the alternative rotated form of the HAR where HAR lags do not overlap

• **hold_back** (int) – 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, optional) – Volatility process to use in the model

• **distribution** (Distribution, optional) – Error distribution to use in the model

• **rescale** (bool, optional) – 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**

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

```python
>>> 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)
```

**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**
## Function Descriptions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bounds()</code></td>
<td>Construct bounds for parameters to use in non-linear optimization</td>
</tr>
<tr>
<td><code>compute_param_cov(params[, backcast, robust])</code></td>
<td>Computes parameter covariances using numerical derivatives.</td>
</tr>
<tr>
<td><code>constraints()</code></td>
<td>Construct linear constraint arrays for use in non-linear optimization</td>
</tr>
<tr>
<td><code>fit([update_freq, disp, starting_values,...])</code></td>
<td>Fits the model given a nobs by 1 vector of sigma2 values</td>
</tr>
<tr>
<td><code>fix(params[, first_obs, last_obs])</code></td>
<td>Allows an ARCHModelFixedResult to be constructed from fixed parameters.</td>
</tr>
<tr>
<td><code>forecast(params[, horizon, start, align,...])</code></td>
<td>Construct forecasts from estimated model</td>
</tr>
<tr>
<td><code>parameter_names()</code></td>
<td>List of parameters names</td>
</tr>
<tr>
<td><code>resids(params[, y, regressors])</code></td>
<td>Compute model residuals</td>
</tr>
<tr>
<td><code>simulate(params, nobs[, burn,...])</code></td>
<td>Simulates data from a linear regression, AR or HAR models</td>
</tr>
<tr>
<td><code>starting_values()</code></td>
<td>Returns starting values for the mean model, often the same as the values returned from fit</td>
</tr>
</tbody>
</table>

### `arch.univariate.HARX.bounds`

**HARX.bounds()**

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, backcast=None, robust=True)**

Computes parameter covariances using numerical derivatives.

**Parameters**

- **params** *(ndarray)* – Model parameters
- **backcast** *(float)* – Value to use for pre-sample observations
- **robust** *(bool, optional)* – Flag indicating whether to use robust standard errors (True) or classic MLE (False)

### `arch.univariate.HARX.constraints`

**HARX.constraints()**

Construct linear constraint arrays for use in non-linear optimization

**Returns**

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

**Notes**

Parameters satisfy a.dot(parameters) - b >= 0
arch.univariate.HARX.fit

HARX.fit(update_freq=1, disp='final', starting_values=None, cov_type='robust', show_warning=True, first_obs=None, last_obs=None, tol=None, options=None, backcast=None)

Fits the model given a nobs by 1 vector of sigma2 values

Parameters

- **update_freq (int, optional)** – Frequency of iteration updates. Output is generated every update_freq iterations. Set to 0 to disable iterative output.
- **disp (str)** – Either ‘final’ to print optimization result or ‘off’ to display nothing
- **starting_values (ndarray, optional)** – Array of starting values to use. If not provided, starting values are constructed by the model components.
- **cov_type (str, optional)** – Estimation method of parameter covariance. Supported options are ‘robust’, which does not assume the Information Matrix Equality holds and ‘classic’ which does. In the ARCH literature, ‘robust’ corresponds to Bollerslev-Wooldridge covariance estimator.
- **show_warning (bool, optional)** – Flag indicating whether convergence warnings should be shown.
- **first_obs ((int, str, datetime, Timestamp))** – First observation to use when estimating model
- **last_obs ((int, str, datetime, Timestamp))** – Last observation to use when estimating model
- **tol (float, optional)** – Tolerance for termination.
- **options (dict, optional)** – Options to pass to scipy.optimize.minimize. Valid entries include ‘ftol’, ‘eps’, ‘disp’, and ‘maxiter’.
- **backcast (float, optional)** – Value to use as backcast. Should be measure $\sigma_0^2$ 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, first_obs=None, last_obs=None)

Allows an ARCHModelFixedResult to be constructed from fixed parameters.

Parameters

- **params ((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, Timestamp}) – First observation to use when fixing model

• **last_obs**({int, str, datetime, Timestamp}) – 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, horizon=1, start=None, align='origin', method='analytic', simulations=1000, rng=None, random_state=None)

Construct forecasts from estimated model

**Parameters**

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

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

• **start**({int, datetime, Timestamp, str}, optional) – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in ‘1945-01-01’.

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

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

• **rng**(callable, optional) – 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**(RandomState, optional) – NumPy RandomState instance to use when method is ‘bootstrap’

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

**Return type** ARCHModelForecast
Examples

```python
>>> import pandas as pd
>>> from arch import arch_model

```arch

```python
>>> 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 of parameters names

Returns names – List of variable names for the mean model

Return type list (str)

**arch.univariate.HARX.resids**

HARX`resids(params, y=None, regressors=None)`

Compute model residuals

Parameters

- **params** (ndarray) – Model parameters
- **y** (ndarray, optional) – Alternative values to use when computing model residuals
- **regressors** (ndarray, optional) – Alternative regressor values to use when computing model residuals

Returns resids – Model residuals

Return type ndarray
arch.univariate.HARX.simulate

HARX.simulate (params, nobs, burn=500, initial_value=None, x=None, initial_value_vol=None)
Simulates data from a linear regression, AR or HAR models

Parameters

- **params** (ndarray) – 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, optional) – Number of values to simulate to initialize the model and remove dependence on initial values.
- **initial_value** ((ndarray, float), optional) – Either a scalar value or max(lags) array set of initial values to use when initializing the model. If omitted, 0.0 is used.
- **x** (ndarray, DataFrame), optional – nob + burn by k array of exogenous variables to include in the simulation.
- **initial_value_vol** ((ndarray, float), optional) – 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 DataFrame

Examples

```python
>>> 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 nob + burn data points

```python
>>> 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()
Returns starting values for the mean model, often the same as the values returned from fit
Returns `sv` – Starting values

Return type `ndarray`

### Properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>distribution</code></td>
<td>Set or gets the error distribution</td>
</tr>
<tr>
<td><code>num_params</code></td>
<td>Returns the number of parameters</td>
</tr>
<tr>
<td><code>volatility</code></td>
<td>Set or gets the volatility process</td>
</tr>
<tr>
<td><code>x</code></td>
<td>Gets the value of the exogenous regressors in the model</td>
</tr>
<tr>
<td><code>y</code></td>
<td>Returns the dependent variable</td>
</tr>
</tbody>
</table>

**arch.univariate.HARX.distribution**

`HARX.distribution`

Set or gets the error distribution

Distributions must be a subclass of `Distribution`

**arch.univariate.HARX.num_params**

`HARX.num_params`

Returns the number of parameters

**arch.univariate.HARX.volatility**

`HARX.volatility`

Set or gets the volatility process

Volatility processes must be a subclass of `VolatilityProcess`

**arch.univariate.HARX.x**

`HARX.x`

Gets the value of the exogenous regressors in the model

**arch.univariate.HARX.y**

`HARX.y`

Returns the dependent variable

### 1.7.5 arch.univariate.LS

```python
class arch.univariate.LS(y=None, x=None, constant=True, hold_back=None, rescale=None)
```

Least squares model estimation and simulation

**Parameters**

- `y` *(ndarray, DataFrame, optional)* – nobs element vector containing the dependent variable

1.7. Mean Models
• **y** – nobs by k element array containing exogenous regressors

• **constant** *(bool, optional)* – Flag whether the model should include a constant

• **hold_back** *(int)* – 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, optional)* – 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

```python
>>> 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bounds</strong>()</td>
<td>Construct bounds for parameters to use in non-linear optimization</td>
</tr>
<tr>
<td><strong>compute_param_cov</strong>(<em>params[, backcast, robust]</em>)</td>
<td>Computes parameter covariances using numerical derivatives.</td>
</tr>
<tr>
<td><strong>constraints</strong>()</td>
<td>Construct linear constraint arrays for use in non-linear optimization</td>
</tr>
<tr>
<td><strong>fit</strong>(<em>update_freq, disp, starting_values,...</em>)</td>
<td>Fits the model given a nobs by 1 vector of sigma2 values</td>
</tr>
<tr>
<td><strong>fix</strong>(<em>params[, first_obs, last_obs]</em>)</td>
<td>Allows an ARCHModelFixedResult to be constructed from fixed parameters.</td>
</tr>
<tr>
<td><strong>forecast</strong>(<em>params[, horizon, start, align,...]</em>)</td>
<td>Construct forecasts from estimated model</td>
</tr>
<tr>
<td><strong>parameter_names</strong>()</td>
<td>List of parameters names</td>
</tr>
<tr>
<td><strong>resids</strong>(<em>params[, y, regressors]</em>)</td>
<td>Compute model residuals</td>
</tr>
<tr>
<td><strong>simulate</strong>(<em>params, nobs[, burn, ...]</em>)</td>
<td>Simulates data from a linear regression, AR or HAR models</td>
</tr>
<tr>
<td><strong>starting_values</strong>()</td>
<td>Returns starting values for the mean model, often the same as the values returned from fit</td>
</tr>
</tbody>
</table>
arch.univariate.LS.bounds

LS.bounds()
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, backcast=None, robust=True)
Computes parameter covariances using numerical derivatives.

Parameters
• params (ndarray) – Model parameters
• backcast (float) – Value to use for pre-sample observations
• robust (bool, optional) – Flag indicating whether to use robust standard errors (True) or classic MLE (False)

arch.univariate.LS.constraints

LS.constraints()
Construct linear constraint arrays for use in non-linear optimization

Returns
• a (ndarray) – Number of constraints by number of parameters loading array
• b (ndarray) – Number of constraints array of lower bounds

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

arch.univariate.LS.fit

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

Parameters
• update_freq (int, optional) – Frequency of iteration updates. Output is generated every update_freq iterations. Set to 0 to disable iterative output.
• disp (str) – Either ‘final’ to print optimization result or ‘off’ to display nothing
• starting_values (ndarray, optional) – Array of starting values to use. If not provided, starting values are constructed by the model components.
• cov_type (str, optional) – Estimation method of parameter covariance. Supported options are ‘robust’, which does not assume the Information Matrix Equality holds and ‘classic’ which does. In the ARCH literature, ‘robust’ corresponds to Bollerslev-Wooldridge covariance estimator.
• **show_warning** *(bool, optional)* – Flag indicating whether convergence warnings should be shown.

• **first_obs** *(int, str, datetime, Timestamp)* – First observation to use when estimating model.

• **last_obs** *(int, str, datetime, Timestamp)* – Last observation to use when estimating model.

• **tol** *(float, optional)* – Tolerance for termination.

• **options** *(dict, optional)* – Options to pass to `scipy.optimize.minimize`. Valid entries include ‘ftol’, ‘eps’, ‘disp’, and ‘maxiter’.

• **backcast** *(float, optional)* – Value to use as backcast. Should be measure $\sigma_0^2$ 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, first_obs=None, last_obs=None)

Allows an ARCHModelFixedResult to be constructed from fixed parameters.

**Parameters**

• **params** *(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, Timestamp)* – First observation to use when fixing model.

• **last_obs** *(int, str, datetime, Timestamp)* – 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, horizon=1, start=None, align='origin', method='analytic', simulations=1000, rng=None, random_state=None)

Construct forecasts from estimated model.
Parameters

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

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

- **start** *(int, datetime, Timestamp, str), optional* – An integer, datetime or str indicating the first observation to produce the forecast for. Datetimes can only be used with pandas inputs that have a datetime index. Strings must be convertible to a date time, such as in ‘1945-01-01’.

- **align** *(str, optional)* – 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’ simplifies computing forecast errors since the realization and h-step forecast are aligned.

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

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

- **rng** *(callable, optional)* – 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** *(RandomState, optional)* – NumPy RandomState instance to use when method is ‘bootstrap’

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

Return type **ARCHModelForecast**

Examples

```python
>>> 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 of parameters names  

Returns names – List of variable names for the mean model  

Return type list (str)

**arch.univariate.LS.resids**

`LS.resids(params, y=None, regressors=None)`  
Compute model residuals  

Parameters  

- **params** (ndarray) – Model parameters  
- **y** (ndarray, optional) – Alternative values to use when computing model residuals  
- **regressors** (ndarray, optional) – Alternative regressor values to use when computing model residuals

Returns resids – Model residuals  

Return type ndarray

**arch.univariate.LS.simulate**

`LS.simulate(params, nobs, burn=500, initial_value=None, x=None, initial_value_vol=None)`  
Simulates data from a linear regression, AR or HAR models  

Parameters  

- **params** (ndarray) – 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, optional) – Number of values to simulate to initialize the model and remove dependence on initial values.  
- **initial_value** (ndarray, float, optional) – Either a scalar value or max(lags) array set of initial values to use when initializing the model. If omitted, 0.0 is used.  
- **x** (ndarray, DataFrame, optional) – nobs + burn by k array of exogenous variables to include in the simulation.
• **initial_value_vol**((ndarray, float), optional) – 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** DataFrame

**Examples**

```python
>>> 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

```python
>>> 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()**

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

**Returns** sv – Starting values

**Return type** ndarray

**Properties**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distribution</td>
<td>Set or gets the error distribution</td>
</tr>
<tr>
<td>num_params</td>
<td>Returns the number of parameters</td>
</tr>
<tr>
<td>volatility</td>
<td>Set or gets the volatility process</td>
</tr>
<tr>
<td>x</td>
<td>Gets the value of the exogenous regressors in the model</td>
</tr>
<tr>
<td>y</td>
<td>Returns the dependent variable</td>
</tr>
</tbody>
</table>

**arch.univariate.LS.distribution**

**LS.distribution**

Set or gets the error distribution

Distributions must be a subclass of Distribution
arch.univariate.LS.num_params

LS.num_params
Returns the number of parameters

arch.univariate.LS.volatility

LS.volatility
Set or gets the volatility process
Volatility processes must be a subclass of VolatilityProcess

arch.univariate.LS.x

LS.x
Gets the value of the exogenous regressors in the model

arch.univariate.LS.y

LS.y
Returns the dependent variable

1.7.6 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.

```python
class arch.univariate.base.ARCHModel (y=None, volatility=None, distribution=None, hold_back=None, rescale=None)
```

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

All public methods that raise Not Implemented Error should be overridden by any subclass. Private methods that raise Not Implemented Error are optional to override but recommended where applicable.

1.8 Volatility Processes

A volatility process is added to a mean model to capture time-varying volatility.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstantVariance()</td>
<td>Constant volatility process</td>
</tr>
<tr>
<td>GARCH([p, o, q, power])</td>
<td>GARCH and related model estimation</td>
</tr>
<tr>
<td>FIGARCH([p, q, truncation])</td>
<td>FIGARCH model</td>
</tr>
<tr>
<td>EGARCH([p, o, q])</td>
<td>EGARCH model estimation</td>
</tr>
<tr>
<td>HARCH([lags])</td>
<td>Heterogeneous ARCH process</td>
</tr>
<tr>
<td>MIDASHyperbolic([m, asym])</td>
<td>MIDAS Hyperbolic ARCH process</td>
</tr>
<tr>
<td>ARCH([p])</td>
<td>ARCH process</td>
</tr>
</tbody>
</table>
1.8.1 arch.univariate.ConstantVariance

class arch.univariate.ConstantVariance
Constant volatility process

Notes
Model has the same variance in all periods

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>backcast(resids)</code></td>
<td>Construct values for backcasting to start the recursion</td>
</tr>
<tr>
<td><code>backcast_transform(backcast)</code></td>
<td>Transformation to apply to user-provided backcast values</td>
</tr>
<tr>
<td><code>bounds(resids)</code></td>
<td>Returns bounds for parameters</td>
</tr>
<tr>
<td><code>compute_variance(parameters, resids, sigma2, ...)</code></td>
<td>Compute the variance for the ARCH model</td>
</tr>
<tr>
<td><code>constraints()</code></td>
<td>Construct parameter constraints arrays for parameter estimation</td>
</tr>
<tr>
<td><code>forecast(parameters, resids, backcast, ...)</code></td>
<td>Forecast volatility from the model</td>
</tr>
<tr>
<td><code>parameter_names()</code></td>
<td>Names of model parameters</td>
</tr>
<tr>
<td><code>simulate(parameters, nobs, rng[, burn, ...])</code></td>
<td>Simulate data from the model</td>
</tr>
<tr>
<td><code>starting_values(resids)</code></td>
<td>Returns starting values for the ARCH model</td>
</tr>
<tr>
<td><code>variance_bounds(resids[, power])</code></td>
<td>param resids Approximate residuals to use to compute the lower and upper bounds</td>
</tr>
</tbody>
</table>

arch.univariate.ConstantVariance.backcast

ConstantVariance.backcast(resids)
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)
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)
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, resids, sigma2, backcast, var_bounds)
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()
Construct parameter constraints arrays for parameter estimation

Returns

• A (ndarray) – Parameters loadings in constraint. Shape is number of constraints by number of parameters

• b (ndarray) – Constraint values, one for each constraint

Notes

Values returned are used in constructing linear inequality constraints of the form $A \cdot \text{parameters} - b \geq 0$

arch.univariate.ConstantVariance.forecast

ConstantVariance.forecast(parameters, resids, backcast, var_bounds, start=None, horizon=1, method='analytic', simulations=1000, rng=None, random_state=None)
Forecast volatility from the model

Parameters

• parameters ((ndarray, Series)) – Parameters required to forecast the volatility model

• resids (ndarray) – Residuals to use in the recursion
backcast (float) – Value to use when initializing the recursion

var_bounds (ndarray, 2-d) – Array containing columns of lower and upper bounds

start (None, int) – Index of the first observation to use as the starting point for the forecast. Default is len(resids).

horizon (int) – Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

method ({'analytic', 'simulation', 'bootstrap}) – Method to use when producing the forecast. The default is analytic.

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

rng (callable) – 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, optional) – 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 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 ()

Names of model parameters

Returns names – Variables names

Return type list (str)

arch.univariate.ConstantVariance.simulate

ConstantVariance.simulate (parameters, nobs, rng, burn=500, initial_value=None)

Simulate data from the model

Parameters

* parameters (ndarray, Series) – Parameters required to simulate the volatility model

* nobs (int) – Number of data points to simulate

1.8. Volatility Processes
• **rng (callable)** – Callable function that takes a single integer input and returns a vector of random numbers

• **burn (int, optional)** – Number of additional observations to generate when initializing the simulation

• **initial_value (float, ndarray, optional)** – Scalar or array of initial values to use when initializing the simulation

Returns

• **resids (ndarray)** – The simulated residuals

• **variance (ndarray)** – The simulated variance

### arch.univariate.ConstantVariance.starting_values

**ConstantVariance.starting_values(resids)**

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** ndarray

### arch.univariate.ConstantVariance.variance_bounds

**ConstantVariance.variance_bounds(resids, power=2.0)**

**Parameters**

- **resids (ndarray)** – Approximate residuals to use to compute the lower and upper bounds on the conditional variance

- **power (float, optional)** – 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** ndarray

**Properties**

<table>
<thead>
<tr>
<th>start</th>
<th>Index to use to start variance subarray selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop</td>
<td>Index to use to stop variance subarray selection</td>
</tr>
</tbody>
</table>

### arch.univariate.ConstantVariance.start

**ConstantVariance.start**

Index to use to start variance subarray selection
1.8.2 arch.univariate.GARCH

class arch.univariate.GARCH (p=1, o=0, q=1, power=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)** – Order of the symmetric innovation
- **o (int)** – Order of the asymmetric innovation
- **q (int)** – Order of the lagged (transformed) conditional variance
- **power (float, optional)** – 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.

num_params
The number of parameters in the model

Type  int

Examples

```python
>>> from arch.univariate import GARCH

Standard GARCH(1,1)

```GARCH(p=1, q=1)

Asymmetric GJR-GARCH process

```python
>>> gjr = GARCH(p=1, o=1, q=1)

Asymmetric TARCH process

```GARCH(p=1, o=1, q=1, power=1.0)

```
Notes

In this class of processes, the variance dynamics are

\[ \sigma_t^\lambda = \omega + \sum_{i=1}^{p} \alpha_i |\epsilon_{t-i}|^\lambda + \sum_{j=1}^{q} \gamma_j |\epsilon_{t-j}|^\lambda I[\epsilon_{t-j} < 0] + \sum_{k=1}^{q} \beta_k \sigma_{t-k}^\lambda \]

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

- **variance_bounds**(resids[, power])
  
  **param resids** Approximate residuals to use to compute the lower and upper bounds

---

**arch.univariate.GARCH.backcast**

GARCH. **backcast**(resids)

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)

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)
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, resids, sigma2, backcast, var_bounds)
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()
Construct parameter constraints arrays for parameter estimation

Returns
- A (ndarray) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- b (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, resids, backcast, var_bounds, start=None, horizon=1, method='analytic', simulations=1000, rng=None, random_state=None)
Forecast volatility from the model

Parameters
- parameters ((ndarray, Series)) – Parameters required to forecast the volatility model
- resids (ndarray) – Residuals to use in the recursion

1.8. Volatility Processes
• **backcast** *(float)* – Value to use when initializing the recursion.

• **var_bounds** *(ndarray, 2-d)* – Array containing columns of lower and upper bounds.

• **start** *(None, int)* – Index of the first observation to use as the starting point for the forecast. Default is len(resids).

• **horizon** *(int)* – Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

• **method** *({'analytic', 'simulation', 'bootstrap'})* – Method to use when producing the forecast. The default is analytic.

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

• **rng** *(callable)* – 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, optional)* – 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** 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()

Names of model parameters

**Returns** names – Variables names

**Return type** list (str)

**arch.univariate.GARCH.simulate**

GARCH.simulate(parameters, nobs, rng, burn=500, initial_value=None)

Simulate data from the model

**Parameters**

• **parameters** *(ndarray, Series)* – Parameters required to simulate the volatility model

• **nobs** *(int)* – Number of data points to simulate
- **rng** *(callable)* – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** *(int, optional)* – Number of additional observations to generate when initializing the simulation
- **initial_value** *(float, ndarray, optional)* – Scalar or array of initial values to use when initializing the simulation

Returns
- **resids** *(ndarray)* – The simulated residuals
- **variance** *(ndarray)* – The simulated variance

### arch.univariate.GARCH.starting_values

**GARCH.starting_values** *(resids)*

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 **ndarray**

### arch.univariate.GARCH.variance_bounds

**GARCH.variance_bounds** *(resids, power=2.0)*

Parameters
- **resids** *(ndarray)* – Approximate residuals to use to compute the lower and upper bounds on the conditional variance
- **power** *(float, optional)* – 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 **ndarray**

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>start</strong></td>
<td>Index to use to start variance subarray selection</td>
</tr>
<tr>
<td><strong>stop</strong></td>
<td>Index to use to stop variance subarray selection</td>
</tr>
</tbody>
</table>

### arch.univariate.GARCH.start

**GARCH.start**

Index to use to start variance subarray selection
arch Documentation, Release 4.9.1+4.g81ceedd

arch.univariate.GARCH.stop

GARCH.stop

Index to use to stop variance subarray selection

1.8.3 arch.univariate.FIGARCH

class arch.univariate.FIGARCH(p=1, q=1, power=2.0, truncation=1000)

FIGARCH model

Parameters

- \( p ((0, 1)) \) – Order of the symmetric innovation
- \( q ((0, 1)) \) – Order of the lagged (transformed) conditional variance
- **power** \((\text{float, optional})\) – Power to use with the innovations, \(\text{abs}(e)^{\text{** power}}\). 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** \((\text{int, optional})\) – Truncation point to use in ARCH(\(\infty\)) representation. Default is 1000.

num_params

The number of parameters in the model

Type int

Examples

```python
>>> from arch.univariate import FIGARCH

Standard FIGARCH

>>> figarch = FIGARCH()

FIARCH

>>> fiarch = FIGARCH(p=0)

FIAVARCH process

>>> fiavarch = FIGARCH(power=1.0)
```

Notes

In this class of processes, the variance dynamics are

\[
h_t = \omega + [1 - \beta L - \phi L (1 - L)^d] e_t^2 + \beta h_{t-1}
\]

where \( L \) is the lag operator and \( d \) is the fractional differencing parameter. The model is estimated using the ARCH(\(\infty\)) representation,

\[
h_t = (1 - \beta)^{-1}\omega + \sum_{i=1}^{\infty} \lambda_i e_{t-i}^2
\]
The weights are constructed using

\[
\delta_1 = d \\
\lambda_1 = d - \beta + \phi
\]

and the recursive equations

\[
\delta_j = \frac{j - 1 - d}{j} \delta_{j-1} \\
\lambda_j = \beta \lambda_{j-1} + \delta_j - \phi \delta_{j-1}.
\]

When \textit{power} is not 2, the ARCH(\infty) representation is still used where \(\epsilon_t^2\) is replaced by \(|\epsilon_t|^p\) and \(p\) is the power.

**Methods**

- \textbf{backcast(resids)}: Construct values for backcasting to start the recursion
- \textbf{backcast_transform(backcast)}: Transformation to apply to user-provided backcast values
- \textbf{bounds(resids)}: Returns bounds for parameters
- \textbf{compute_variance(parameters, resids, sigma2, ...)}: Compute the variance for the ARCH model
- \textbf{constraints()}: Construct parameter constraints arrays for parameter estimation
- \textbf{forecast(parameters, resids, backcast, ...)}: Forecast volatility from the model
- \textbf{parameter_names()}: Names of model parameters
- \textbf{simulate(parameters, nob, rng[, burn, ...])}: Simulate data from the model
- \textbf{starting_values(resids)}: Returns starting values for the ARCH model
- \textbf{variance_bounds(resids[, power])}

**arch.univariate.FIGARCH.backcast**

\textbf{FIGARCH.backcast (resids)}

Construct values for backcasting to start the recursion

- \textbf{Parameters} \textit{resids (ndarray)} – Vector of (approximate) residuals
- \textbf{Returns} \textit{backcast} – Value to use in backcasting in the volatility recursion
- \textbf{Return type} \textit{float}

**arch.univariate.FIGARCH.backcast_transform**

\textbf{FIGARCH.backcast_transform (backcast)}

Transformation to apply to user-provided backcast values

- \textbf{Parameters} \textit{backcast (float, ndarray)} – User-provided \textit{backcast} that approximates \textit{sigma2}[0].
- \textbf{Returns} \textit{backcast} – Backcast transformed to the model-appropriate scale

1.8. Volatility Processes
Return type \{float, ndarray\}

**arch.univariate.FIGARCH.bounds**

FIGARCH\_bounds (resids)

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, resids, sigma2, backcast, var_bounds)

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 ()

Construct parameter constraints arrays for parameter estimation

Returns

- A (ndarray) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- b (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, resids, backcast, var_bounds, start=None, horizon=1, method='analytic', simulations=1000, rng=None, random_state=None)

Forecast volatility from the model

Parameters

- parameters ((ndarray, Series)) – Parameters required to forecast the volatility model

Chapter 1. Univariate Volatility Models
• **resids** (*ndarray*) – Residuals to use in the recursion

• **backcast** (*float*) – Value to use when initializing the recursion

• **var_bounds** (*ndarray, 2-d*) – Array containing columns of lower and upper bounds

• **start** (*None, int*) – Index of the first observation to use as the starting point for the forecast. Default is len(resids).

• **horizon** (*int*) – Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

• **method** (*{'analytic', 'simulation', 'bootstrap'}*) – Method to use when producing the forecast. The default is analytic.

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

• **rng** (*callable*) – Callable random number generator required if method is ‘simulation’. Must take a single shape input and return random sample numbers with that shape.

• **random_state** (*RandomState, optional*) – 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**

- **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()**

- **Names of model parameters**

**Returns**

- **names** – Variables names

**Return type**

- **list (str)**

**arch.univariate.FIGARCH.simulate**

**FIGARCH.simulate(parameters, nobs, rng, burn=500, initial_value=None)**

- **Simulate data from the model**

**Parameters**

- **parameters** (*{ndarray, Series}* ) – Parameters required to simulate the volatility model

- **nobs** (*int*) – Number of data points to simulate
• **rng** (*callable*) – Callable function that takes a single integer input and returns a vector of random numbers

• **burn** (*int, optional*) – Number of additional observations to generate when initializing the simulation

• **initial_value** (*{float, ndarray}, optional*) – Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (*ndarray*) – The simulated residuals

- **variance** (*ndarray*) – The simulated variance

---

**arch.univariate.FIGARCH.starting_values**

FIGARCH.starting_values(*resids*)

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 *ndarray*

---

**arch.univariate.FIGARCH.variance_bounds**

FIGARCH.variance_bounds(*resids, power=2.0*)

Parameters

- **resids** (*ndarray*) – Approximate residuals to use to compute the lower and upper bounds on the conditional variance

- **power** (*float, optional*) – 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 *ndarray*

---

**Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>start</strong></td>
<td>Index to use to start variance subarray selection</td>
</tr>
<tr>
<td><strong>stop</strong></td>
<td>Index to use to stop variance subarray selection</td>
</tr>
<tr>
<td><strong>truncation</strong></td>
<td>Truncation lag for the ARCH-infinity approximation</td>
</tr>
</tbody>
</table>

---

**arch.univariate.FIGARCH.start**

FIGARCH.start

Index to use to start variance subarray selection
arch.univariate.FIGARCH.stop

FIGARCH.stop
Index to use to stop variance subarray selection

arch.univariate.FIGARCH.truncation

FIGARCH.truncation
Truncation lag for the ARCH-infinity approximation

1.8.4 arch.univariate. EGARCH

class arch.univariate. EGARCH (p=1, o=0, q=1)
EGARCH model estimation

Parameters
• p (int) – Order of the symmetric innovation
• o (int) – Order of the asymmetric innovation
• q (int) – Order of the lagged (transformed) conditional variance

num_params
The number of parameters in the model

Type int

Examples

```python
>>> from arch.univariate import EGARCH

Symmetric EGARCH(1,1)
```  
```python
>>> egarch = EGARCH(p=1, q=1)

Standard EGARCH process
```  
```python
>>> egarch = EGARCH(p=1, o=1, q=1)

Exponential ARCH process
```  
```python
>>> each = EGARCH(p=5)
```

Notes

In this class of processes, the variance dynamics are

$$\ln \sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i (|e_{t-i}| - \sqrt{2/\pi}) + \sum_{j=1}^{o} \gamma_j e_{t-j} + \sum_{k=1}^{q} \beta_k \ln \sigma_{t-k}^2$$

where $$e_t = \epsilon_t / \sigma_t$$.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backcast(resids)</td>
<td>Construct values for backcasting to start the recursion.</td>
</tr>
<tr>
<td>backcast_transform(backcast)</td>
<td>Transformation to apply to user-provided backcast values.</td>
</tr>
<tr>
<td>bounds(resids)</td>
<td>Returns bounds for parameters.</td>
</tr>
<tr>
<td>compute_variance(parameters, resids, sigma2, ...)</td>
<td>Compute the variance for the ARCH model.</td>
</tr>
<tr>
<td>constraints()</td>
<td>Construct parameter constraints arrays for parameter estimation.</td>
</tr>
<tr>
<td>forecast(parameters, resids, backcast, ...)</td>
<td>Forecast volatility from the model.</td>
</tr>
<tr>
<td>parameter_names()</td>
<td>Names of model parameters.</td>
</tr>
<tr>
<td>simulate(parameters, nobs, rng[, burn, ...])</td>
<td>Simulate data from the model.</td>
</tr>
<tr>
<td>starting_values(resids)</td>
<td>Returns starting values for the ARCH model.</td>
</tr>
<tr>
<td>variance_bounds(resids[, power])</td>
<td>Approximate residuals to use to compute the lower and upper bounds.</td>
</tr>
</tbody>
</table>

**arch.univariate. EGARCH.backcast**

EGARCH.backcast(resids)

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)

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)

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, resids, sigma2, backcast, var_bounds)*

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()**

Construct parameter constraints arrays for parameter estimation

**Returns**

- **A** *(ndarray)* – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** *(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, resids, backcast, var_bounds, start=None, horizon=1, method='analytic', simulations=1000, rng=None, random_state=None)*

Forecast volatility from the model

**Parameters**

- **parameters** *(ndarray, Series)* – Parameters required to forecast the volatility model
- **resids** *(ndarray)* – Residuals to use in the recursion
- **backcast** *(float)* – Value to use when initializing the recursion
- **var_bounds** *(ndarray, 2-d)* – Array containing columns of lower and upper bounds
- **start** *(None, int)* – Index of the first observation to use as the starting point for the forecast. Default is len(resids).
- **horizon** *(int)* – Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].
- **method** *({'analytic', 'simulation', 'bootstrap}')* – Method to use when producing the forecast. The default is analytic.
• **simulations** (*int*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.

• **rng** (*callable*) – 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, optional*) – 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** 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* ()

Names of model parameters

**Returns**

*names* – Variables names

**Return type** list (str)

**arch.univariate.EGARCH.simulate**

EGARCH.*simulate* (*parameters, nobs, rng, burn=500, initial_value=None*)

Simulate data from the model

**Parameters**

- **parameters** (*ndarray, Series*) – Parameters required to simulate the volatility model
- **nobs** (*int*) – Number of data points to simulate
- **rng** (*callable*) – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** (*int, optional*) – Number of additional observations to generate when initializing the simulation
- **initial_value** (*float, ndarray, optional*) – Scalar or array of initial values to use when initializing the simulation

**Returns**

- **resids** (*ndarray*) – The simulated residuals
- **variance** (*ndarray*) – The simulated variance
**arch.univariate.\texttt{EGARCH.starting\_values}**

\texttt{EGARCH.starting\_values}(\texttt{resids})

Returns starting values for the ARCH model

- **Parameters** \texttt{resids} (\texttt{ndarray}) – Array of (approximate) residuals to use when computing starting values

- **Returns** \texttt{sv} – Array of starting values

- **Return type** \texttt{ndarray}

**arch.univariate.\texttt{EGARCH.variance\_bounds}**

\texttt{EGARCH.variance\_bounds}(\texttt{resids}, \texttt{power=2.0})

- **Parameters**
  - \texttt{resids} (\texttt{ndarray}) – Approximate residuals to use to compute the lower and upper bounds on the conditional variance
  - \texttt{power} (\texttt{float}, \texttt{optional}) – Power used in the model. 2.0, the default corresponds to standard ARCH models that evolve in squares.

- **Returns** \texttt{var\_bounds} – Array containing columns of lower and upper bounds with the same number of elements as \texttt{resids}

- **Return type** \texttt{ndarray}

**Properties**

<table>
<thead>
<tr>
<th>\texttt{start}</th>
<th>Index to use to start variance subarray selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{stop}</td>
<td>Index to use to stop variance subarray selection</td>
</tr>
</tbody>
</table>

**arch.univariate.\texttt{EGARCH.start}**

\texttt{EGARCH.start}

Index to use to start variance subarray selection

**arch.univariate.\texttt{EGARCH.stop}**

\texttt{EGARCH.stop}

Index to use to stop variance subarray selection

### 1.8.5 \texttt{arch.univariate.HARCH}

**class arch.univariate.\texttt{HARCH}(lags=1)**

Heterogeneous ARCH process

- **Parameters** \texttt{lags} (\{\texttt{list, array, int}\}) – List of lags to include in the model, or if scalar, includes all lags up the value

- **num\_params**

  The number of parameters in the model

---

1.8. Volatility Processes
Type  int

Examples

```python
>>> from arch.univariate import HARCH
```

Lag-1 HARCH, which is identical to an ARCH(1)

```python
>>> harch = HARCH()
```

More useful and realistic lag lengths

```python
>>> harch = HARCH(lags=[1, 5, 22])
```

Notes

In a Heterogeneous ARCH process, variance dynamics are

\[
\sigma_t^2 = \omega + \sum_{i=1}^{m} \alpha_i \left( \epsilon_{t-1}^2 \right) \left( \sum_{j=1}^{l_i} \epsilon_{t-j}^2 \right)
\]

In the common case where lags=[1,5,22], the model is

\[
\sigma_t^2 = \omega + \alpha_1 \epsilon_{t-1}^2 + \alpha_5 \left( \sum_{j=1}^{5} \epsilon_{t-j}^2 \right) + \alpha_{22} \left( \sum_{j=1}^{22} \epsilon_{t-j}^2 \right)
\]

A HARCH process is a special case of an ARCH process where parameters in the more general ARCH process have been restricted.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backcast(resids)</td>
<td>Construct values for backcasting to start the recursion</td>
</tr>
<tr>
<td>backcast_transform(backcast)</td>
<td>Transformation to apply to user-provided backcast values</td>
</tr>
<tr>
<td>bounds(resids)</td>
<td>Returns bounds for parameters</td>
</tr>
<tr>
<td>compute_variance(parameters, resids, sigma2, ...)</td>
<td>Compute the variance for the ARCH model</td>
</tr>
<tr>
<td>constraints()</td>
<td>Construct parameter constraints arrays for parameter estimation</td>
</tr>
<tr>
<td>forecast(parameters, resids, backcast, ...)</td>
<td>Forecast volatility from the model</td>
</tr>
<tr>
<td>parameter_names()</td>
<td>Names of model parameters</td>
</tr>
<tr>
<td>simulate(parameters, nobs, rng[, burn, ...])</td>
<td>Simulate data from the model</td>
</tr>
<tr>
<td>starting_values(resids)</td>
<td>Returns starting values for the ARCH model</td>
</tr>
<tr>
<td>variance_bounds(resids[, power])</td>
<td>Approximate residuals to use to compute the lower and upper bounds</td>
</tr>
</tbody>
</table>
arch.univariate.HARCH.backcast

`HARCH.backcast(resids)`
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)`
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)`
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, resids, sigma2, backcast, var_bounds)`
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()`
Construct parameter constraints arrays for parameter estimation

- **Returns**
• **A** (*ndarray*) – Parameters loadings in constraint. Shape is number of constraints by number of parameters

• **b** (*ndarray*) – Constraint values, one for each constraint

**Notes**

Values returned are used in constructing linear inequality constraints of the form \( A \cdot \text{parameters} - b \geq 0 \)

**arch.univariate.HARCH.forecast**

\[ \text{HARCH}.\text{forecast}(\text{parameters}, \text{resids}, \text{backcast}, \text{var_bounds}, \text{start}=\text{None}, \text{horizon}=1, \text{method}=\text{'analytic'}, \text{simulations}=1000, \text{rng}=\text{None}, \text{random_state}=\text{None}) \]

Forecast volatility from the model

**Parameters**

• **parameters** (*ndarray, Series*) – Parameters required to forecast the volatility model

• **resids** (*ndarray*) – Residuals to use in the recursion

• **backcast** (*float*) – Value to use when initializing the recursion

• **var_bounds** (*ndarray, 2-d*) – Array containing columns of lower and upper bounds

• **start** (*None, int*) – Index of the first observation to use as the starting point for the forecast. Default is len(resids).

• **horizon** (*int*) – Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in \([1, \text{horizon}]\).

• **method** (*{'analytic', 'simulation', 'bootstrap'}*) – Method to use when producing the forecast. The default is analytic.

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

• **rng** (*callable*) – 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, optional*) – 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** 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()
Names of model parameters

Returns:
- **names** (Variables names)

Return type: list (str)

arch.univariate.HARCH.simulate

HARCH.simulate(parameters, nobs, rng, burn=500, initial_value=None)
Simulate data from the model

Parameters:
- **parameters** (ndarray, Series) – Parameters required to simulate the volatility model
- **nobs** (int) – Number of data points to simulate
- **rng** (callable) – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** (int, optional) – Number of additional observations to generate when initializing the simulation
- **initial_value** (float, ndarray, optional) – Scalar or array of initial values to use when initializing the simulation

Returns:
- **resids** (ndarray) – The simulated residuals
- **variance** (ndarray) – The simulated variance

arch.univariate.HARCH.starting_values

HARCH.starting_values(resids)
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: ndarray

arch.univariate.HARCH.variance_bounds

HARCH.variance_bounds(resids, power=2.0)

Parameters:
- **resids** (ndarray) – Approximate residuals to use to compute the lower and upper bounds on the conditional variance
- **power** (float, optional) – 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 resid

**Return type** ndarray

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Index to use to start variance subarray selection</td>
</tr>
<tr>
<td>stop</td>
<td>Index to use to stop variance subarray selection</td>
</tr>
</tbody>
</table>

**arch.univariate.HARCH.start**

HARCH.

**HARCH.start**

Index to use to start variance subarray selection

**arch.univariate.HARCH.stop**

HARCH.

**stop**

Index to use to stop variance subarray selection

### 1.8.6 arch.univariate.MIDASHyperbolic

class **arch.univariate.MIDASHyperbolic** (m=22, asym=False)

MIDAS Hyperbolic ARCH process

**Parameters**

- **m** (*int*) – Length of maximum lag to include in the model
- **asym** (*bool*) – Flag indicating whether to include an asymmetric term

**num_params**

The number of parameters in the model

**Type** int

### Examples

```python
>>> from arch.univariate import MIDASHyperbolic
```

22-lag MIDAS Hyperbolic process

```python
>>> harch = MIDASHyperbolic()
```

Longer 66-period lag

```python
>>> harch = MIDASHyperbolic(m=66)
```

Asymmetric MIDAS Hyperbolic process

```python
>>> harch = MIDASHyperbolic(asym=True)
```
Notes

In a MIDAS Hyperbolic process, the variance evolves according to

\[ \sigma_t^2 = \omega + \sum_{i=1}^{m} (\alpha + \gamma I[\epsilon_{t-j} < 0]) \phi_i(\theta) \epsilon_{t-i}^2 \]

where

\[ \phi_i(\theta) \propto \Gamma(i + \theta)/\Gamma(i + 1)\Gamma(\theta) \]

where \( \Gamma \) is the gamma function. \( \{\phi_i(\theta)\} \) is normalized so that \( \sum \phi_i(\theta) = 1 \)

References

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>backcast</strong>(resids)</td>
<td>Construct values for backcasting to start the recursion</td>
</tr>
<tr>
<td><strong>backcast_transform</strong>(backcast)</td>
<td>Transformation to apply to user-provided backcast values</td>
</tr>
<tr>
<td><strong>bounds</strong>(resids)</td>
<td>Returns bounds for parameters</td>
</tr>
<tr>
<td><strong>compute_variance</strong>(parameters, resids, sigma2, ...)</td>
<td>Compute the variance for the ARCH model</td>
</tr>
<tr>
<td><strong>constraints()</strong></td>
<td>Constraints</td>
</tr>
<tr>
<td><strong>forecast</strong>(parameters, resids, backcast, ...)</td>
<td>Forecast volatility from the model</td>
</tr>
<tr>
<td><strong>parameter_names()</strong></td>
<td>Names of model parameters</td>
</tr>
<tr>
<td><strong>simulate</strong>(parameters, nobs, rng[, burn, ...])</td>
<td>Simulate data from the model</td>
</tr>
<tr>
<td><strong>starting_values</strong>(resids)</td>
<td>Returns starting values for the ARCH model</td>
</tr>
<tr>
<td><strong>variance_bounds</strong>(resids[, power])</td>
<td><strong>param resids</strong> Approximate residuals to use to compute the lower and upper bounds</td>
</tr>
</tbody>
</table>

**arch.univariate.MIDASHyperbolic.backcast**

**MIDASHyperbolic.backcast**(resids)

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)

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)
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, resids, sigma2, backcast, var_bounds)
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()
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, resids, backcast, var_bounds, start=None, horizon=1, method='analytic', simulations=1000, rng=None, random_state=None)

Forecast volatility from the model

Parameters

- parameters (ndarray, Series) – Parameters required to forecast the volatility model
- resids (ndarray) – Residuals to use in the recursion
- backcast (float) – Value to use when initializing the recursion
- var_bounds (ndarray, 2-d) – Array containing columns of lower and upper bounds
- start ((None, int)) – Index of the first observation to use as the starting point for the forecast. Default is len(resids).
- horizon (int) – Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].
- method ("analytic", "simulation", "bootstrap") – Method to use when producing the forecast. The default is analytic.
- simulations (int) – Number of simulations to run when computing the forecast using either simulation or bootstrap.
- rng (callable) – 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, optional) – 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 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()

Names of model parameters

Returns names – Variables names

Return type list (str)
arch.univariate.MIDASHyperbolic.simulate

MIDASHyperbolic.simulate(parameters, nobs, rng, burn=500, initial_value=None)
Simulate data from the model

Parameters

• parameters({ndarray, Series}) – Parameters required to simulate the volatility model
• nobs(int) – Number of data points to simulate
• rng(callable) – Callable function that takes a single integer input and returns a vector of random numbers
• burn(int, optional) – Number of additional observations to generate when initializing the simulation
• initial_value({float, ndarray}, optional) – Scalar or array of initial values to use when initializing the simulation

Returns

• resid(ndarray) – The simulated residuals
• variance(ndarray) – The simulated variance

arch.univariate.MIDASHyperbolic.starting_values

MIDASHyperbolic.starting_values(resids)
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 ndarray

arch.univariate.MIDASHyperbolic.variance_bounds

MIDASHyperbolic.variance_bounds(resids, power=2.0)

Parameters

• resid(ndarray) – Approximate residuals to use to compute the lower and upper bounds on the conditional variance
• power(float, optional) – 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 resid

Return type ndarray

Properties
### arch.univariate.MIDASHyperbolic.start

MIDASHyperbolic.start
- Index to use to start variance subarray selection

### arch.univariate.MIDASHyperbolic.stop

MIDASHyperbolic.stop
- Index to use to stop variance subarray selection

### 1.8.7 arch.univariate.ARCH

**class** arch.univariate.ARCH(*p=1*)

ARCH process

**Parameters**
- `p (int)` – Order of the symmetric innovation

**num_params**
- The number of parameters in the model
  - **Type** `int`

**Examples**

ARCH(1) process

```python
>>> from arch.univariate import ARCH
```

ARCH(5) process

```python
>>> arch = ARCH(p=5)
```

**Notes**

The variance dynamics of the model estimated

\[
s_t^2 = \omega + \sum_{i=1}^{p} \alpha_i s_{t-i}^2
\]

**Methods**

- `backcast(resids)`
  - Construct values for backcasting to start the recursion
- `backcast_transform(backcast)`
  - Transformation to apply to user-provided backcast values
Table 25 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bounds(resids)</code></td>
<td>Returns bounds for parameters</td>
</tr>
<tr>
<td><code>compute_variance(parameters, resids, sigma2, ...)</code></td>
<td>Compute the variance for the ARCH model</td>
</tr>
<tr>
<td><code>constraints()</code></td>
<td>Construct parameter constraints arrays for parameter estimation</td>
</tr>
<tr>
<td><code>forecast(parameters, resids, backcast, ...)</code></td>
<td>Forecast volatility from the model</td>
</tr>
<tr>
<td><code>parameter_names()</code></td>
<td>Names of model parameters</td>
</tr>
<tr>
<td><code>simulate(parameters, nobs, rng[, burn, ...])</code></td>
<td>Simulate data from the model</td>
</tr>
<tr>
<td><code>starting_values(resids)</code></td>
<td>Returns starting values for the ARCH model</td>
</tr>
<tr>
<td><code>variance_bounds(resids[, power])</code></td>
<td>Approximate residuals to use to compute the lower and upper bounds</td>
</tr>
</tbody>
</table>

---

**arch.univariate.ARCH.backcast**

**ARCH.backcast(resids)**  
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)**  
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)**  
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, resids, sigma2, backcast, var_bounds)**  
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()`  
Construct parameter constraints arrays for parameter estimation

**Returns**  
- **A** (*ndarray*) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (*ndarray*) – Constraint values, one for each constraint

**Notes**  
Values returned are used in constructing linear inequality constraints of the form \( A \cdot \text{parameters} - b \geq 0 \)

---

**arch.univariate.ARCH.forecast**

`ARCH.forecast(parameters, resids, backcast, var_bounds, start=None, horizon=1, method='analytic', simulations=1000, rng=None, random_state=None)`  
Forecast volatility from the model

**Parameters**  
- **parameters** (*{ndarray, Series}* or *ndarray*) – Parameters required to forecast the volatility model
- **resids** (*ndarray*) – Residuals to use in the recursion
- **backcast** (*float*) – Value to use when initializing the recursion
- **var_bounds** (*ndarray, 2-d*) – Array containing columns of lower and upper bounds
- **start** (*{None, int}* or *int*) – Index of the first observation to use as the starting point for the forecast. Default is len(resids).
- **horizon** (*int*) – Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in \([1, \text{horizon}]\).
- **method** (*{‘analytic’, ‘simulation’, ‘bootstrap’}* or *str*) – Method to use when producing the forecast. The default is analytic.
- **simulations** (*int*) – Number of simulations to run when computing the forecast using either simulation or bootstrap.
- **rng** (*callable*) – 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, optional*) – 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** 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.univariate.ARCH.parameter_names* ()

Names of model parameters

**Returns** names – Variables names

**Return type** list (str)

**arch.univariate.ARCH.simulate**

*arch.univariate.ARCH.simulate* (*parameters, nobs, rng, burn=500, initial_value=None*)

Simulate data from the model

**Parameters**

  * **parameters** (ndarray, Series) – Parameters required to simulate the volatility model
  * **nobs** (int) – Number of data points to simulate
  * **rng** (callable) – Callable function that takes a single integer input and returns a vector of random numbers
  * **burn** (int, optional) – Number of additional observations to generate when initializing the simulation
  * **initial_value** (float, ndarray, optional) – Scalar or array of initial values to use when initializing the simulation

**Returns**

  * **resids** (ndarray) – The simulated residuals
  * **variance** (ndarray) – The simulated variance
arch Documentation, Release 4.9.1+4.g81ceedd

**arch.univariate.ARCH.starting_values**

**ARCH.starting_values**\((resids)\)
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**
  - *ndarray*

**arch.univariate.ARCH.variance_bounds**

**ARCH.variance_bounds**\((resids, power=2.0)\)

Parameters

- *resids* \((ndarray)\) – Approximate residuals to use to compute the lower and upper bounds on the conditional variance

- *power* \((float, optional)\) – 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**
  - *ndarray*

**Properties**

<table>
<thead>
<tr>
<th>start</th>
<th>Index to use to start variance subarray selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop</td>
<td>Index to use to stop variance subarray selection</td>
</tr>
</tbody>
</table>

**arch.univariate.ARCH.start**

**ARCH.start**
Index to use to start variance subarray selection

**arch.univariate.ARCH.stop**

**ARCH.stop**
Index to use to stop variance subarray selection

### 1.8.8 Parameterless Variance Processes

Some volatility processes use fixed parameters and so have no parameters that are estimable.

<table>
<thead>
<tr>
<th><em>EWMAVariance</em>[1 lam]</th>
<th>Exponentially Weighted Moving-Average (RiskMetrics) Variance process</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>RiskMetrics2006</em>[tau0, tau1, kmax, rho]</td>
<td>RiskMetrics 2006 Variance process</td>
</tr>
</tbody>
</table>

1.8. Volatility Processes

115
arch Documentation, Release 4.9.1+4.g81ceedd

**arch.univariate.EWMAVariance**

**class** arch.univariate.EWMAVariance(  
    lam=0.94)  
Exponentially Weighted Moving-Average (RiskMetrics) Variance process

**Parameters**  
*lam* *(float, None), optional* – Smoothing parameter. Default is 0.94. Set to None to estimate lam jointly with other model parameters

**num_params**  
The number of parameters in the model  
**Type** int

**Examples**

Daily RiskMetrics EWMA process

```python
>>> from arch.univariate import EWMAVariance
>>> rm = EWMAVariance(0.94)
```

**Notes**

The variance dynamics of the model

\[
\sigma_t^2 = \lambda \sigma_{t-1}^2 + (1 - \lambda) \varepsilon_{t-1}^2
\]

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
- **variance_bounds(resids[, power])**: Approximate residuals to use to compute the lower and upper bounds
arch.univariate.EWMAVariance.backcast

EWMAVariance.backcast(resids)
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)
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)
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, resids, sigma2, backcast, var_bounds)
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()
Construct parameter constraints arrays for parameter estimation

Returns
• **A** *(ndarray)* – Parameters loadings in constraint. Shape is number of constraints by number of parameters

• **b** *(ndarray)* – Constraint values, one for each constraint

**Notes**

Values returned are used in constructing linear inequality constraints of the form \(A \cdot \text{parameters} - b \geq 0\)

**arch.univariate.EWMAVariance.forecast**

EWMAVariance.forecast \((\text{parameters, resids, backcast, var_bounds, }\text{start=\text{None, horizon=1, method='analytic', simulations=1000, rng=\text{None, random_state=\text{None}}})}\)

Forecast volatility from the model

**Parameters**

• **parameters** *(ndarray, Series)* – Parameters required to forecast the volatility model

• **resids** *(ndarray)* – Residuals to use in the recursion

• **backcast** *(float)* – Value to use when initializing the recursion

• **var_bounds** *(ndarray, 2-d)* – Array containing columns of lower and upper bounds

• **start** *(None, int)* – Index of the first observation to use as the starting point for the forecast. Default is len(resids).

• **horizon** *(int)* – Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

• **method** *(‘analytic’, ‘simulation’, ‘bootstrap’)* – Method to use when producing the forecast. The default is analytic.

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

• **rng** *(callable)* – 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, optional)* – 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**

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()

Names of model parameters

Returns names – Variables names

Return type list (str)

**arch.univariate.EWMAVariance.simulate**

EWMAVariance.simulate(parameters, nobs, rng, burn=500, initial_value=None)

Simulate data from the model

Parameters

- **parameters** (ndarray, Series) – Parameters required to simulate the volatility model
- **nobs** (int) – Number of data points to simulate
- **rng** (callable) – Callable function that takes a single integer input and returns a vector of random numbers
- **burn** (int, optional) – Number of additional observations to generate when initializing the simulation
- **initial_value** (float, ndarray, optional) – Scalar or array of initial values to use when initializing the simulation

Returns

- **resids** (ndarray) – The simulated residuals
- **variance** (ndarray) – The simulated variance

**arch.univariate.EWMAVariance.starting_values**

EWMAVariance.starting_values(resids)

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 ndarray

1.8. Volatility Processes
**arch.univariate.EWMAVariance.variance_bounds**

```python
EWMAVariance.variance_bounds(resids, power=2.0)
```

**Parameters**

- `resids` *(ndarray)* – Approximate residuals to use to compute the lower and upper bounds on the conditional variance
- `power` *(float, optional)* – 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**

`ndarray`

**Properties**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start</code></td>
<td>Index to use to start variance subarray selection</td>
</tr>
<tr>
<td><code>stop</code></td>
<td>Index to use to stop variance subarray selection</td>
</tr>
</tbody>
</table>

**arch.univariate.EWMAVariance.start**

```python
EWMAVariance.start
```

Index to use to start variance subarray selection

**arch.univariate.EWMAVariance.stop**

```python
EWMAVariance.stop
```

Index to use to stop variance subarray selection

**arch.univariate.RiskMetrics2006**

```python
class arch.univariate.RiskMetrics2006(tau0=1560, tau1=4, kmax=14, rho=1.4142135623730951)
```

RiskMetrics 2006 Variance process

**Parameters**

- `tau0` *(int, optional)* – Length of long cycle. Default is 1560.
- `tau1` *(int, optional)* – Length of short cycle. Default is 4.
- `kmax` *(int, optional)* – Number of components. Default is 14.
- `rho` *(float, optional)* – Relative scale of adjacent cycles. Default is sqrt(2)

**num_params**

The number of parameters in the model

**Type**

`int`

**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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backcast(resids)</td>
<td>Construct values for backcasting to start the recursion</td>
</tr>
<tr>
<td>backcast_transform(backcast)</td>
<td>Transformation to apply to user-provided backcast values</td>
</tr>
<tr>
<td>bounds(resids)</td>
<td>Returns bounds for parameters</td>
</tr>
<tr>
<td>compute_variance(parameters, resids, sigma2, ...)</td>
<td>Compute the variance for the ARCH model</td>
</tr>
<tr>
<td>constraints()</td>
<td>Construct parameter constraints arrays for parameter estimation</td>
</tr>
<tr>
<td>forecast(parameters, resids, backcast, ...)</td>
<td>Forecast volatility from the model</td>
</tr>
<tr>
<td>parameter_names()</td>
<td>Names of model parameters</td>
</tr>
<tr>
<td>simulate(parameters, nobs, rng[, burn, ...])</td>
<td>Simulate data from the model</td>
</tr>
<tr>
<td>starting_values(resids)</td>
<td>Returns starting values for the ARCH model</td>
</tr>
<tr>
<td>variance_bounds(resids[, power])</td>
<td></td>
</tr>
</tbody>
</table>

param resids Approximate residuals to use to compute the lower and upper bounds

arch.univariate.RiskMetrics2006.backcast

RiskMetrics2006.backcast(resids)

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 ndarray

arch.univariate.RiskMetrics2006.backcast_transform

RiskMetrics2006.backcast_transform(backcast)

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

1.8. Volatility Processes 121
Return type: {float, ndarray}

**RiskMetrics2006.bounds**

RiskMetrics2006.bounds(resids)
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]]

**RiskMetrics2006.compute_variance**

RiskMetrics2006.compute_variance(parameters, resids, sigma2, backcast, var_bounds)
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

**RiskMetrics2006.constraints**

RiskMetrics2006.constraints()
Construct parameter constraints arrays for parameter estimation

Returns:
- **A** (ndarray) – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** (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

**RiskMetrics2006.forecast**

RiskMetrics2006.forecast(parameters, resids, backcast, var_bounds, start=None, horizon=1, method='analytic', simulations=1000, rng=None, random_state=None)
Forecast volatility from the model

Parameters
• **parameters**({ndarray, Series}) – Parameters required to forecast the volatility model

• **resids**(ndarray) – Residuals to use in the recursion

• **backcast**(float) – Value to use when initializing the recursion

• **var_bounds**(ndarray, 2-d) – Array containing columns of lower and upper bounds

• **start**({None, int}) – Index of the first observation to use as the starting point for the forecast. Default is len(resids).

• **horizon**(int) – Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].

• **method**({'analytic', 'simulation', 'bootstrap'}) – Method to use when producing the forecast. The default is analytic.

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

• **rng**(callable) – 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, optional) – 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** 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()

Names of model parameters

**Returns** names – Variables names

**Return type** list (str)

**arch.univariate.RiskMetrics2006.simulate**

RiskMetrics2006.simulate(parameters, nobs, rng, burn=500, initial_value=None)

Simulate data from the model

**Parameters**
• **parameters** *(ndarray, Series)* – Parameters required to simulate the volatility model
• **nobs** *(int)* – Number of data points to simulate
• **rng** *(callable)* – Callable function that takes a single integer input and returns a vector of random numbers
• **burn** *(int, optional)* – Number of additional observations to generate when initializing the simulation
• **initial_value** *(float, ndarray, optional)* – Scalar or array of initial values to use when initializing the simulation

**Returns**

• **resids** *(ndarray)* – The simulated residuals
• **variance** *(ndarray)* – The simulated variance

### arch.univariate.RiskMetrics2006.starting_values

**RiskMetrics2006.starting_values** *(resids)*

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**

ndarray

### arch.univariate.RiskMetrics2006.variance_bounds

**RiskMetrics2006.variance_bounds** *(resids, power=2.0)*

**Parameters**

- **resids** *(ndarray)* – Approximate residuals to use to compute the lower and upper bounds on the conditional variance
- **power** *(float, optional)* – 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**

ndarray

**Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>start</strong></td>
<td>Index to use to start variance subarray selection</td>
</tr>
<tr>
<td><strong>stop</strong></td>
<td>Index to use to stop variance subarray selection</td>
</tr>
</tbody>
</table>
1.8.9 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.

\[ \text{FixedVariance}(\text{variance[, unit_scale]}) \]

Fixed volatility process

### Parameters

- **variance** (array, Series) – Array containing the variances to use. Should have the same shape as the data used in the model.
- **unit_scale** (bool, optional) – 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
**arch.univariate.FixedVariance.backcast**

FixedVariance.backcast(resids)

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)

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)

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, resids, sigma2, backcast, var_bounds)

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()**

Construct parameter constraints arrays for parameter estimation

**Returns**

- **A** *(ndarray)* – Parameters loadings in constraint. Shape is number of constraints by number of parameters
- **b** *(ndarray)* – Constraint values, one for each constraint

**Notes**

Values returned are used in constructing linear inequality constraints of the form $A \cdot \text{parameters} - b \geq 0$

### arch.univariate.FixedVariance.forecast

**FixedVariance.forecast** *(parameters, resids, backcast, var_bounds, start=None, horizon=1, method='analytic', simulations=1000, rng=None, random_state=None)*

Forecast volatility from the model

**Parameters**

- **parameters** *(ndarray, Series)* – Parameters required to forecast the volatility model
- **resids** *(ndarray)* – Residuals to use in the recursion
- **backcast** *(float)* – Value to use when initializing the recursion
- **var_bounds** *(ndarray, 2-d)* – Array containing columns of lower and upper bounds
- **start** *(None, int)* – Index of the first observation to use as the starting point for the forecast. Default is len(resids).
- **horizon** *(int)* – Forecast horizon. Must be 1 or larger. Forecasts are produced for horizons in [1, horizon].
- **method** *(‘analytic’, ’simulation’, ’bootstrap’)* – Method to use when producing the forecast. The default is analytic.
- **simulations** *(int)* – Number of simulations to run when computing the forecast using either simulation or bootstrap.
- **rng** *(callable)* – 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, optional)* – 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.

### 1.8. Volatility Processes
Return type  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()`

Names of model parameters

Returns  names – Variables names

Return type  list (str)

`arch.univariate.FixedVariance.simulate`

`FixedVariance.simulate(parameters, nobs, rng, burn=500, initial_value=None)`

Simulate data from the model

Parameters

- `parameters`({ndarray, Series}) – Parameters required to simulate the volatility model
- `nobs` (int) – Number of data points to simulate
- `rng` (callable) – Callable function that takes a single integer input and returns a vector of random numbers
- `burn` (int, optional) – Number of additional observations to generate when initializing the simulation
- `initial_value`({float, ndarray}, optional) – Scalar or array of initial values to use when initializing the simulation

Returns

- `resids` (ndarray) – The simulated residuals
- `variance` (ndarray) – The simulated variance

`arch.univariate.FixedVariance.starting_values`

`FixedVariance.starting_values(resids)`

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  ndarray

arch.univariate.FixedVariance.variance_bounds

FixedVariance.variance_bounds(resids, power=2.0)

Parameters

• resids (ndarray) – Approximate residuals to use to compute the lower and upper bounds on the conditional variance

• power (float, optional) – 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  ndarray

Properties

<table>
<thead>
<tr>
<th>start</th>
<th>Index to use to start variance subarray selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop</td>
<td>Index to use to stop variance subarray selection</td>
</tr>
</tbody>
</table>

arch.univariate.FixedVariance.start

FixedVariance.start

Index to use to start variance subarray selection

arch.univariate.FixedVariance.stop

FixedVariance.stop

Index to use to stop variance subarray selection

1.8.10 Writing New Volatility Processes

All volatility processes must inherit from :class:VolatilityProcess and provide all public methods.

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.

1.9 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
1.9.1 Setup

Imports used in this example.

```
[3]: 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.

```
[4]: 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.

```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

<table>
<thead>
<tr>
<th>Dep. Variable:</th>
<th>VIX Index</th>
<th>R-squared:</th>
<th>0.876</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Model:</td>
<td>HAR</td>
<td>Adj. R-squared:</td>
<td>0.876</td>
</tr>
<tr>
<td>Vol Model:</td>
<td>Constant Variance</td>
<td>Log-Likelihood:</td>
<td>-2267.95</td>
</tr>
<tr>
<td>Distribution:</td>
<td>Normal</td>
<td>AIC:</td>
<td>4545.90</td>
</tr>
<tr>
<td>Method:</td>
<td>Maximum Likelihood</td>
<td>BIC:</td>
<td>4571.50</td>
</tr>
<tr>
<td>No. Observations:</td>
<td>1237</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date:</td>
<td>Wed, Aug 28 2019</td>
<td>Df Residuals:</td>
<td>1232</td>
</tr>
<tr>
<td>Time:</td>
<td>09:40:56</td>
<td>Df Model:</td>
<td>5</td>
</tr>
</tbody>
</table>

Mean Model

<table>
<thead>
<tr>
<th>coef</th>
<th>95.0% Conf. Int.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>[0.264, 1.003]</td>
</tr>
<tr>
<td>VIX Index[0:1]</td>
<td>[0.800, 1.058]</td>
</tr>
<tr>
<td>VIX Index[0:5]</td>
<td>[-0.158, 0.463e-02]</td>
</tr>
<tr>
<td>VIX Index[0:22]</td>
<td>[-1.076e-03, 0.124]</td>
</tr>
</tbody>
</table>

(continues on next page)
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.

```python
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())
```

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]  |

Covariance estimator: White’s Heteroskedasticity Consistent Estimator

```python
ax = vol_res.plot('D')
```

Zero Mean - GJR-GARCH Model Results

<table>
<thead>
<tr>
<th>Dep. Variable:</th>
<th>resid</th>
<th>R-squared:</th>
<th>0.000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Model:</td>
<td>Zero Mean</td>
<td>Adj. R-squared:</td>
<td>0.001</td>
</tr>
<tr>
<td>Vol Model:</td>
<td>GJR-GARCH</td>
<td>Log-Likelihood:</td>
<td>-1936.93</td>
</tr>
<tr>
<td>Distribution:</td>
<td>Normal</td>
<td>AIC:</td>
<td>3881.86</td>
</tr>
<tr>
<td>Method:</td>
<td>Maximum Likelihood</td>
<td>BIC:</td>
<td>3902.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No. Observations:</td>
<td>1237</td>
</tr>
<tr>
<td>Date:</td>
<td>Wed, Aug 28 2019</td>
<td>Df Residuals:</td>
<td>1233</td>
</tr>
<tr>
<td>Time:</td>
<td>09:40:56</td>
<td>Df Model:</td>
<td>4</td>
</tr>
</tbody>
</table>

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
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.

```python
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: 2, Func. Count: 20, Neg. LLF: 1936.2884244078432
Iteration: 3, Func. Count: 30, Neg. LLF: 1936.1738940313
Iteration: 8, Func. Count: 72, Neg. LLF: 1935.9470521933054
Optimization terminated successfully.  (Exit mode 0)
Current function value: 1935.947051582333
Iterations: 8
Function evaluations: 73

(continues on next page)
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
Date: Wed, Aug 28 2019  Df Residuals: 1232
Time: 09:40:57  Df Model: 5

Mean Model
==================================================================================================
  coef  std err  t     P>|t|  95.0% Conf. Int.
-------------------------------------------------------
Const    0.5584   0.153 3.661 0.000260 [0.260, 0.857]
VIX Index[0:1]  0.9376  0.03625 25.866 9.897e-147 [0.867, 1.009]
VIX Index[0:5] -0.0249  0.03782 -0.657 0.5109 [-9.899e-02, 0.028]
VIX Index[0:22]  0.0493  0.02102 2.344 3.709e-02 [0.008, 0.091]

Volatility Model
========================================================================
  coef  std err  t     P>|t|  95.0% Conf. Int.
------------------------------------------------
scale    0.9986  0.08081 12.358 4.420e-35 [0.840, 1.157]
========================================================================
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]: 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())

HAR - Fixed Variance (Unit Scale) Model Results

Dep. Variable: VIX Index  R-squared: 0.
Mean Model: HAR  Adj. R-squared: 0.
Distribution: Normal  AIC: 3879.48

(continues on next page)
Method: Maximum Likelihood  BIC: 3899.

No. Observations: 1237

Date: Wed, Aug 28 2019  Df Residuals: 1233

Time: 09:40:58  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 | 25.940 | 2.388e-148      | [ 0.867, 1.009]  |
| VIX Index[0:5] | -0.0262 | 3.774e-02 | -0.693 | 0.488           | [-0.100, 0.781e-02] |
| VIX Index[0:22] | 0.0499  | 2.099e-02 | 2.380 | 1.733e-02       | [8.810e-03, 0.099e-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**

<table>
<thead>
<tr>
<th>Dep. Variable:</th>
<th>VIX Index</th>
<th>R-squared:</th>
<th>0.876</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Model:</td>
<td>HAR</td>
<td>Adj. R-squared:</td>
<td>0.875</td>
</tr>
<tr>
<td>Distribution:</td>
<td>Normal</td>
<td>AIC:</td>
<td>3881.23</td>
</tr>
<tr>
<td>Method:</td>
<td>Maximum Likelihood</td>
<td>BIC:</td>
<td>3922.19</td>
</tr>
<tr>
<td>No. Observations:</td>
<td>1237</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Df Residuals:</td>
<td>1229</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Df Model:</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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] |

(continues on next page)
1.10 Distributions

A distribution is the final component of an ARCH Model.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Standard normal distribution for use with ARCH models</td>
</tr>
<tr>
<td>StudentsT</td>
<td>Standardized Student’s distribution for use with ARCH models</td>
</tr>
<tr>
<td>SkewStudent</td>
<td>Standardized Skewed Student’s distribution for use with ARCH models</td>
</tr>
<tr>
<td>GeneralizedError</td>
<td>Generalized Error distribution for use with ARCH models</td>
</tr>
</tbody>
</table>

1.10.1 arch.univariate.Normal

```python
class arch.univariate.Normal(random_state=None)
```

Standard normal distribution for use with ARCH models

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounds(resids)</td>
<td><strong>param resids</strong> Residuals to use when computing the bounds</td>
</tr>
<tr>
<td>cdf(resids[, parameters])</td>
<td>Cumulative distribution function</td>
</tr>
<tr>
<td>constraints()</td>
<td><strong>returns</strong></td>
</tr>
<tr>
<td></td>
<td>• A (ndarray) – Constraint loadings</td>
</tr>
<tr>
<td>loglikelihood(parameters, resids, sigma2[, ...])</td>
<td>Computes the log-likelihood of assuming residuals are normally distributed, conditional on the variance</td>
</tr>
<tr>
<td>parameter_names()</td>
<td>Names of distribution shape parameters</td>
</tr>
<tr>
<td>ppf(pits[, parameters])</td>
<td>Inverse cumulative density function (ICDF)</td>
</tr>
<tr>
<td>simulate(parameters)</td>
<td>Simulates i.i.d.</td>
</tr>
<tr>
<td>starting_values(std_resid)</td>
<td><strong>param std_resid</strong> Estimated standardized residuals to use in computing starting</td>
</tr>
</tbody>
</table>

Chapter 1. Univariate Volatility Models
arch.univariate.Normal.bounds

Normal.bounds(resids)

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, parameters=None)
Cumulative distribution function

Parameters
• resids (ndarray) – Values at which to evaluate the cdf
• parameters (ndarray) – Distribution parameters. Use None for parameterless dis-
tributions.

Returns f – CDF values
Return type ndarray

arch.univariate.Normal.constraints

Normal.constraints()

Returns
• A (ndarray) – Constraint loadings
• b (ndarray) – Constraint values

Notes
Parameters satisfy the constraints A.dot(parameters)-b >= 0

arch.univariate.Normal.loglikelihood

Normal.loglikelihood(parameters, resids, sigma2, individual=False)
Computes the log-likelihood of assuming residuals are normally distributed, conditional on the variance

Parameters
• parameters (ndarray) – The normal likelihood has no shape parameters. Empty
  since the standard normal has no shape parameters.
• resids (ndarray) – The residuals to use in the log-likelihood calculation
• sigma2 (ndarray) – Conditional variances of resids
• individual (bool, optional) – 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

$$\ln f(x) = -\frac{1}{2} \left( \ln 2\pi + \ln \sigma^2 + \frac{x^2}{\sigma^2} \right)$$

arch.univariate.Normal.parameter_names

Normal.parameter_names()

Names of distribution shape parameters

Returns names – Parameter names

Return type list (str)

arch.univariate.Normal.ppf

Normal.ppf(pits, parameters=None)

Inverse cumulative density function (ICDF)

Parameters

- pits (ndarray) – Probability-integral-transformed values in the interval (0, 1).
- parameters (ndarray, optional) – Distribution parameters. Use None for parameterless distributions.

Returns i – Inverse CDF values

Return type ndarray

arch.univariate.Normal.simulate

Normal.simulate(parameters)

Simulates i.i.d. draws from the distribution

Parameters parameters (ndarray) – 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.Normal.starting_values

Normal.starting_values(std_resid)

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 ndarray
Notes

Size of sv depends on the distribution

Properties

random_state
The NumPy RandomState attached to the distribution

arch.univariate.Normal.random_state

Normal.random_state
The NumPy RandomState attached to the distribution

1.10.2 arch.univariate.StudentsT

class arch.univariate.StudentsT(random_state=None)
Standardized Student’s distribution for use with ARCH models

Methods

bounds(resids)

param resids Residuals to use when computing the bounds

cdf(resids[, parameters])
Cumulative distribution function

constraints()
returns
• A (ndarray) – Constraint loadings

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.

parameter_names()
Names of distribution shape parameters

ppf(pits[, parameters])
Inverse cumulative density function (ICDF)

simulate(parameters)
Simulates i.i.d.

starting_values(std_resid)

param std_resid Estimated standardized residuals to use in computing starting

arch.univariate.StudentsT.bounds

StudentsT.bounds(resids)

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, parameters=None)
Cumulative distribution function

Parameters

- resids (ndarray) – Values at which to evaluate the cdf
- parameters (ndarray) – Distribution parameters. Use None for parameterless distributions.

Returns f – CDF values

Return type ndarray

arch.univariate.StudentsT.constraints

StudentsT.constraints()

Returns

- A (ndarray) – Constraint loadings
- b (ndarray) – Constraint values

Notes

Parameters satisfy the constraints A.dot(parameters)-b >= 0

arch.univariate.StudentsT.loglikelihood

StudentsT.loglikelihood(parameters, resids, sigma2, individual=False)
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 (ndarray) – Shape parameter of the t distribution
- resids (ndarray) – The residuals to use in the log-likelihood calculation
- sigma2 (ndarray) – Conditional variances of resids
- individual (bool, optional) – 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
\[
\ln \Gamma \left( \frac{\nu + 1}{2} \right) - \ln \Gamma \left( \frac{\nu}{2} \right) - \frac{1}{2} \ln(\pi(\nu - 2)\sigma^2) - \frac{\nu + 1}{2} \ln\left(1 + \frac{x^2}{(\sigma^2(\nu - 2))}\right)
\]

where $\Gamma$ is the gamma function.

arch.univariate.StudentsT.parameter_names

StudentsT.parameter_names()

Names of distribution shape parameters

Returns names – Parameter names

Return type list (str)

arch.univariate.StudentsT.ppf

StudentsT.ppf (pits, parameters=None)

Inverse cumulative density function (ICDF)

Parameters

• pits (ndarray) – Probability-integral-transformed values in the interval (0, 1).

• parameters (ndarray, optional) – Distribution parameters. Use None for parameterless distributions.

Returns i – Inverse CDF values

Return type ndarray

arch.univariate.StudentsT.simulate

StudentsT.simulate (parameters)

Simulates i.i.d. draws from the distribution

Parameters parameters (ndarray) – 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.StudentsT.starting_values

StudentsT.starting_values (std_resid)

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 ndarray

1.10. Distributions
Notes

Uses relationship between kurtosis and degree of freedom parameter to produce a moment-based estimator for the starting values.

Properties

\begin{verbatim}
random_state

The NumPy RandomState attached to the distribution
\end{verbatim}

arch.univariate.StudentsT.random_state

StudentsT.random_state

The NumPy RandomState attached to the distribution

1.10.3 arch.univariate.SkewStudent

class arch.univariate.SkewStudent (random_state=None)

Standardized Skewed Student’s distribution for use with ARCH models

Notes

The Standardized Skewed Student’s distribution takes two parameters, \( \eta \) and \( \lambda \). \( \eta \) controls the tail shape and is similar to the shape parameter in a Standardized Student’s t. \( \lambda \) controls the skewness. When \( \lambda = 0 \) the distribution is identical to a standardized Student’s t.

References

Methods

\begin{verbatim}
bounds(resids)

param resids Residuals to use when computing the bounds
\end{verbatim}

\begin{verbatim}
cdf(resids[, parameters])

returns

• A (ndarray) – Constraint loadings

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.

parameter_names()

Names of distribution shape parameters
\end{verbatim}

Table 40 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ppf(pits[, parameters])</code></td>
<td>Inverse cumulative density function (ICDF)</td>
</tr>
<tr>
<td><code>simulate(parameters)</code></td>
<td>Simulates i.i.d.</td>
</tr>
<tr>
<td><code>starting_values(std_resid)</code></td>
<td>Estimated standardized residuals to use in computing starting</td>
</tr>
</tbody>
</table>

**arch.univariate.SkewStudent.bounds**

SkewStudent.bounds(resids)

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, parameters=None)

Parameters
- `resids (ndarray)` – Values at which to evaluate the cdf
- `parameters (ndarray)` – Distribution parameters. Use None for parameterless distributions.

Returns
- `f` – CDF values

Return type
- ndarray

**arch.univariate.SkewStudent.constraints**

SkewStudent.constraints()

Returns
- `A (ndarray)` – Constraint loadings
- `b (ndarray)` – Constraint values

Notes

Parameters satisfy the constraints $A \cdot \mathbf{parameters} - b \geq 0$

**arch.univariate.SkewStudent.loglikelihood**

SkewStudent.loglikelihood(parameters, resids, sigma2, individual=False)

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** (*ndarray*) – Shape parameter of the skew-t distribution

• **resids** (*ndarray*) – The residuals to use in the log-likelihood calculation

• **sigma2** (*ndarray*) – Conditional variances of resids

• **individual** (*bool, optional*) – 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

\[
\ln \left[ \frac{bc}{\sigma} \left( 1 + \frac{1}{\eta - 2} \left( \frac{a + bx/\sigma}{1 + sgn(x/\sigma + a/b)\lambda} \right)^2 \right) \right]^{-(\eta + 1)/2},
\]

where \( 2 < \eta < \infty \), and \(-1 < \lambda < 1\). The constants \( a, b, \) and \( c \) are given by

\[
a = 4\lambda c \frac{\eta - 2}{\eta - 1}, \quad b^2 = 1 + 3\lambda^2 - a^2, \quad c = \frac{\Gamma \left( \frac{\eta + 1}{2} \right)}{\sqrt{\pi (\eta - 2)} \Gamma \left( \frac{\eta}{2} \right)},
\]

and \( \Gamma \) is the gamma function.

**arch.univariate.SkewStudent.parameter_names**

SkewStudent.**parameter_names**()

Names of distribution shape parameters

**Returns**

**names** – Parameter names

**Return type** list (str)

**arch.univariate.SkewStudent.ppf**

SkewStudent.**ppf**(*pits, parameters=None*)

Inverse cumulative density function (ICDF)

**Parameters**

• **pits** (*ndarray*) – Probability-integral-transformed values in the interval \((0, 1)\).

• **parameters** (*ndarray, optional*) – Distribution parameters. Use None for parameterless distributions.

**Returns**

**i** – Inverse CDF values

**Return type** *ndarray*
arch.univariate.SkewStudent.simulate

SkewStudent.simulate(parameters)
Simulates i.i.d. draws from the distribution

Parameters parameters (ndarray) – 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)

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 ndarray

Notes
Uses relationship between kurtosis and degree of freedom parameter to produce a moment-based estimator for the starting values.

Properties

random_state The NumPy RandomState attached to the distribution

arch.univariate.SkewStudent.random_state

SkewStudent.random_state
The NumPy RandomState attached to the distribution

1.10.4 arch.univariate.GeneralizedError

class arch.univariate.GeneralizedError (random_state=None)
Generalized Error distribution for use with ARCH models

Methods

bounds(resids)

param resids Residuals to use when computing the bounds

cdf(resids[, parameters])
Cumulative distribution function

Continued on next page
Table 42 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>constraints()</code></td>
<td>returns</td>
</tr>
<tr>
<td></td>
<td>• A (ndarray) – Constraint loadings</td>
</tr>
<tr>
<td><code>loglikelihood()</code></td>
<td>Computes the log-likelihood of assuming residuals are have a Generalized Error Distribution, conditional on the variance.</td>
</tr>
<tr>
<td><code>parameter_names()</code></td>
<td>Names of distribution shape parameters</td>
</tr>
<tr>
<td><code>ppf(pits[, parameters])</code></td>
<td>Inverse cumulative density function (ICDF)</td>
</tr>
<tr>
<td><code>simulate(parameters)</code></td>
<td>Simulates i.i.d.</td>
</tr>
<tr>
<td><code>starting_values(std_resid)</code></td>
<td>Estimated standardized residuals to use in computing starting</td>
</tr>
</tbody>
</table>

**arch.univariate.GeneralizedError.bounds**

GeneralizedError.bounds(resids)

- **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, parameters=None)

- **Parameters**
  - `resids` (ndarray) – Values at which to evaluate the cdf
  - `parameters` (ndarray) – Distribution parameters. Use None for parameterless distributions.
- **Returns**
  - `f` – CDF values
  - `Return type` ndarray

**arch.univariate.GeneralizedError.constraints**

GeneralizedError.constraints()

- **Returns**
  - • A (ndarray) – Constraint loadings
  - • b (ndarray) – Constraint values

**Notes**

Parameters satisfy the constraints A.dot(parameters)-b >= 0
arch.univariate.GeneralizedError.loglikelihood

**GeneralizedError.loglikelihood** *(parameters, resids, sigma2, individual=False)*
Computes the log-likelihood of assuming residuals are have a Generalized Error Distribution, conditional on the variance.

**Parameters**
- **parameters** *(ndarray)* – Shape parameter of the GED distribution
- **resids** *(ndarray)* – The residuals to use in the log-likelihood calculation
- **sigma2** *(ndarray)* – Conditional variances of resids
- **individual** *(bool, optional)* – 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

$$\ln \nu - \ln c - \ln \Gamma\left(\frac{1}{\nu}\right) + \left(1 + \frac{1}{\nu}\right) \ln 2 - \frac{1}{2} \ln \sigma^2 - \frac{1}{2} \ln \left| \frac{x}{c\sigma} \right|^\nu$$

where $\Gamma$ is the gamma function and $\ln c$ is

$$\ln c = \frac{1}{2} \left( -\frac{2}{\nu} \ln 2 + \ln \Gamma\left(\frac{1}{\nu}\right) - \ln \Gamma\left(\frac{3}{\nu}\right) \right).$$

arch.univariate.GeneralizedError.parameter_names

**GeneralizedError.parameter_names** ()
Names of distribution shape parameters

**Returns**
- **names** – Parameter names

**Return type** *list (str)*

arch.univariate.GeneralizedError.ppf

**GeneralizedError.ppf** *(pits, parameters=None)*
Inverse cumulative density function (ICDF)

**Parameters**
- **pits** *(ndarray)* – Probability-integral-transformed values in the interval (0, 1).
- **parameters** *(ndarray, optional)* – Distribution parameters. Use None for parameterless distributions.

**Returns**
- **i** – Inverse CDF values

**Return type** *ndarray*
arch.univariate.GeneralizedError.simulate

GeneralizedError.simulate(parameters)
Simulates i.i.d. draws from the distribution

Parameters
parameters (ndarray) – Distribution parameters

Returns
simulator – Callable that takes a single output size argument and returns i.i.d. draws from the distribution

Return type
callable

arch.univariate.GeneralizedError.starting_values

GeneralizedError.starting_values(std_resid)

Parameters
std_resid (ndarray) – Estimated standardized residuals to use in computing starting values for the shape parameter

Returns
sv – Array containing starting value for shape parameter

Return type
ndarray

Notes
Defaults to 1.5 which implies heavier tails than a normal

Properties

random_state
The NumPy RandomState attached to the distribution

arch.univariate.GeneralizedError.random_state

GeneralizedError.random_state
The NumPy RandomState attached to the distribution

1.10.5 Writing New Distributions

All distributions must inherit from :class:`Distribution` and provide all public methods.

class arch.univariate.distribution.Distribution(name, random_state=None)
Template for subclassing only

1.11 Utilities

Utilities that do not fit well on other pages.
1.11.1 Test Results

class arch.utility.testing.WaldTestStatistic(stat, df, null, alternative, name=None)
    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, optional) – Name of test

critical_values
    Critical values test for common test sizes

null
    Null hypothesis

pval
    P-value of test statistic

stat
    Test statistic

1.12 Theoretical Background

To be completed
CHAPTER 2

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]:

```python
import warnings
warnings.simplefilter('ignore')

%matplotlib inline
import seaborn
```

[2]:

```python
seaborn.mpl.rcParams['figure.figsize'] = (10.0, 6.0)
seaborn.mpl.rcParams['savefig.dpi'] = 90
```

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.

[3]:

```python
import numpy as np
import pandas as pd

import arch.data.frenchdata

ff = arch.data.frenchdata.load()
```
The data set contains the Fama-French factors, including the excess market return.

```python
excess_market = ff.iloc[:, 0]
print(ff.describe())
```

<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th>Mkt-RF</th>
<th>SMB</th>
<th>HML</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1109.000000</td>
<td>1109.000000</td>
<td>1109.000000</td>
<td>1109.000000</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>0.659946</td>
<td>0.206555</td>
<td>0.368864</td>
<td>0.274220</td>
<td></td>
</tr>
<tr>
<td>std</td>
<td>5.327524</td>
<td>3.191132</td>
<td>3.482352</td>
<td>0.253377</td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>-29.130000</td>
<td>-16.870000</td>
<td>-13.280000</td>
<td>-0.060000</td>
<td></td>
</tr>
<tr>
<td>25%</td>
<td>-1.970000</td>
<td>-1.560000</td>
<td>-1.320000</td>
<td>0.030000</td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td>1.020000</td>
<td>0.070000</td>
<td>0.140000</td>
<td>0.230000</td>
<td></td>
</tr>
<tr>
<td>75%</td>
<td>3.610000</td>
<td>1.730000</td>
<td>1.740000</td>
<td>0.430000</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>38.850000</td>
<td>36.700000</td>
<td>35.460000</td>
<td>1.350000</td>
<td></td>
</tr>
</tbody>
</table>

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.

```python
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.

```python
params = sharpe_ratio(excess_market)
params
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mu</td>
<td>7.919351</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sigma</td>
<td>18.455084</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR</td>
<td>0.429115</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtype: float64</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.

```python
from arch.bootstrap import StationaryBootstrap
bs = StationaryBootstrap(12, excess_market)
results = bs.apply(sharpe_ratio, 2500)
SR = pd.DataFrame(results[:, -1:, columns=['SR'])
fig = SR.hist(bins=40)
```
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.
2.1.2 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.

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.
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.

```python
[13]: 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.

```python
[14]: probit_wrap(endog, exog)
[[  3.00353578  0.00164268  0.45457492]]

dtype: float64
```

The wrapper can be directly used to estimate the parameter covariance or to construct confidence intervals.

```python
[15]: 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  0.435172 -8.601967e-05 -0.110662
gre  -0.000086  4.124129e-07 -0.000047
gpa  -0.110662 -4.692308e-05  0.040495

[16]: se = pd.Series(np.sqrt(np.diag(cov)), index=exog.columns)
print(se)

print('T-stats')
print(params / se)

    Const     gre     gpa
    0.659675  0.000642  0.201234
dtype: float64

T-stats
    Const  -4.553051
gre  2.557696
gpa  2.258936
dtype: float64

[17]: 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  -4.183013  0.000413  0.064824
  Upper  -1.671674  0.002832  0.840327
```
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.

```python
[18]: def probit_wrap_start_params(endog, exog, start_params=None):
    return sm.Probit(endog, exog).fit(start_params=start_params, disp=0).params
```

```python
[19]: bs.reset() # Reset to original state for comparability
cov = bs.cov(
    probit_wrap_start_params,
    extra_kwargs={'start_params': params.values})
cov = pd.DataFrame(cov, index=exog.columns, columns=exog.columns)
print(cov)
```

<table>
<thead>
<tr>
<th></th>
<th>Const</th>
<th>gre</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>0.435172</td>
<td>-8.601967e-05</td>
<td>-0.110662</td>
</tr>
<tr>
<td>gre</td>
<td>-0.000086</td>
<td>4.124129e-07</td>
<td>-0.000047</td>
</tr>
<tr>
<td>gpa</td>
<td>-0.110662</td>
<td>-4.692308e-05</td>
<td>0.040495</td>
</tr>
</tbody>
</table>

2.1.3 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.

```python
[20]: 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, random_state=rs)
print(bs.conf_int(mean_diff, method='studentized'))
```

```
[[0.1991302 ]
 [0.51317728]]
```

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

Chapter 2. Bootstrapping

156
• 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)

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!

```python
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 main function used will return a 3-element array containing the parameters.

```python
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.

```python
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 \( \hat{\theta}_b \) and then constructs the confidence interval as

\[
\left[ \hat{\theta} + \left( \hat{\theta} - \hat{\theta}^*_l \right), \hat{\theta} + \left( \hat{\theta} - \hat{\theta}^*_u \right) \right]
\]

where \( \hat{\theta}^*_l \) and \( \hat{\theta}^*_u \) are the \( \alpha/2 \) and \( 1 - \alpha/2 \) empirical quantiles of the bootstrap distribution. When \( \theta \) is a vector, the empirical quantiles are computed element-by-element.

```python
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, \( \hat{\theta}_b^* \). The confidence interval is then defined.

\[
\left[ \hat{\theta}^*_l, \hat{\theta}^*_u \right]
\]

where \( \hat{\theta}^*_l \) and \( \hat{\theta}^*_u \) are the \( \alpha/2 \) and \( 1 - \alpha/2 \) empirical quantiles of the bootstrap distribution.

```python
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

\[
\left[ \hat{\theta} + \hat{\sigma}^{-1} \Phi^{-1}(\alpha/2), \hat{\theta} - \hat{\sigma}^{-1} \Phi^{-1}(\alpha/2) \right]
\]

where \( \hat{\sigma} \) is the bootstrap estimate of the parameter standard error.

```python
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

\[
\left[ \hat{\theta} + \hat{\sigma} G^{-1}(\alpha/2), \hat{\theta} + \hat{\sigma} G^{-1}(1 - \alpha/2) \right]
\]
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.

```python
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
xi = 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.

```python
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.

```python
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
xi = 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 pattern `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.

```python
from arch.bootstrap import IIDBootstrap

bs = IIDBootstrap(returns)
xi = 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.

```python
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, $\hat{\theta}_b^*$, where the percentiles used are

$$
\Phi \left( \frac{\Phi^{-1}(\hat{b}) + z_\alpha}{1 - \hat{a} \left( \Phi^{-1}(\hat{b}) + z_\alpha \right)} \right)
$$

where $z_\alpha$ is the usual quantile from the normal distribution and $\hat{b}$ is the empirical bias estimate,

$$
\hat{b} = \# \left\{ \hat{\theta}_b^* < \hat{\theta} \right\} / B
$$

$a$ 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.

```python
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.

```python
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

```python
>>> params
mu 8.148534
sigma 14.508540
SR 0.561637
dtype: float64

>>> param_cov
mu   sigma    SR
mu  3.729435 -0.442891  0.273945
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.

```python
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(sharpe_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.

```python
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

```python
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.
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`.

```python
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.

```python
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.

```python
resids = y - x.dot(ols_semi_v2(y, x))
bs = IIDBootstrap(y, x, resids=resids)
ci = 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.

```python
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.

```python
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.
2.7.1 arch.bootstrap.IIDBootstrap

```python
class arch.bootstrap.IIDBootstrap(*args, **kwargs)
    Bootstrap using uniform resampling

Parameters
    • `args` – Positional arguments to bootstrap
    • `kwargs` – Keyword arguments to bootstrap

index
    The current index of the bootstrap
    Type ndarray

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

random_state
    RandomState instance used by bootstrap
    Type RandomState

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 accessible 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.

```python
>>> 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]
```
Set the random_state if reproducibility is required

```python
>>> from numpy.random import RandomState
>>> rs = RandomState(1234)
>>> bs = IIDBootstrap(x, y=y, z=z, random_state=rs)
```

See also:

`arch.bootstrap.IndependentSamplesBootstrap`

Methods

- **apply(func[, reps, extra_kwargs])**
  Applies a function to bootstrap replicated data

  Parameters

  - `func` (*callable*)
    Function the computes parameter values. See Notes for requirements

  - `reps` (*int, optional*)
    Number of bootstrap replications

  - `extra_kwargs` (*dict, optional*)
    Extra keyword arguments to use when calling `func`. Must not conflict with keyword arguments used to initialize bootstrap

  Returns `results` – reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

  Return type `ndarray`

`arch.bootstrap.IIDDBootstrap.apply`

IIDDBootstrap.**apply** (`func`, `reps=1000`, `extra_kwargs=None`)

Applies a function to bootstrap replicated data

Parameters

- `func` (*callable*)
  Function the computes parameter values. See Notes for requirements

- `reps` (*int, optional*)
  Number of bootstrap replications

- `extra_kwargs` (*dict, optional*)
  Extra keyword arguments to use when calling `func`. Must not conflict with keyword arguments used to initialize bootstrap

Returns `results` – reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

Return type `ndarray`
Notes

When there are no extra keyword arguments, the function is called

\[
\text{func}(\text{params}, \ast \text{args}, \ast \ast \text{kwargs})
\]

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

```python
>>> 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)

Iterator for use when bootstrapping

**Parameters**

reps (int) – Number of bootstrap replications

**Returns**

gen – Generator to iterate over in bootstrap calculations

**Return type**

generator

Example

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```python
>>> 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, **kwargs)
Clones the bootstrap using different data.

Parameters

- **args** – Positional arguments to bootstrap
- **kwargs** – Keyword arguments to bootstrap

Returns

Bootstrap instance

Return type

bs

arch.bootstrap.IIDBootstrap.conf_int

IIDBootstrap.conf_int(func, reps=1000, method='basic', size=0.95, tail='two', extra_kwargs=None, reuse=False, sampling='nonparametric', std_err_func=None, studentize_reps=1000)

Parameters

- **func** (*callable*) – Function the computes parameter values. See Notes for requirements
- **reps** (*int*, optional) – Number of bootstrap replications
- **method** (*string*, optional) – One of ‘basic’, ‘percentile’, ‘studentized’, ‘norm’ (identical to ‘var’, ‘cov’), ‘bc’ (identical to ‘debiased’, ‘bias-corrected’), or ‘bca’
- **size** (*float*, optional) – Coverage of confidence interval
- **tail** (*string*, optional) – One of ‘two’, ‘upper’ or ‘lower’.
- **reuse** (*bool*, optional) – 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** (*string*, optional) – 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*, optional) – Extra keyword arguments to use when calling func and std_err_func, when appropriate
- **std_err_func** (*callable*, optional) – 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*, optional) – Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when std_err_func is provided

Returns

intervals – 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

2-d array

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, reps=1000, recenter=True, extra_kwags=None)`

Compute parameter covariance using bootstrap

**Parameters**

- **func** (*callable*) – Callable function that returns the statistic of interest as a 1-d array
- **reps** (*int, optional*) – Number of bootstrap replications
- **recenter** (*bool, optional*) – 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_kwags** (*dict, optional*) – Dictionary of extra keyword arguments to pass to func

**Returns** **cov** – Bootstrap covariance estimator
Return type  ndarray

Notes

func must have the signature

```python
func(params, *args, **kwargs)
```

where params are a 1-dimensional array, and *args and **kwargs are data used in 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.

Example

Bootstrap covariance of the mean

```python
>>> 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

```python
>>> 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()

Gets the state of the bootstrap’s random number generator

Returns state – Array containing the state

Return type RandomState state vector

arch.bootstrap.IIDBootstrap.reset

IIDBootstrap.reset(use_seed=True)

Resets the bootstrap to either its initial state or the last seed.
Parameters `use_seed(bool, optional)`—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)`

Seeds the bootstrap’s random number generator

Parameters `value(int)`—Integer to use as the seed

`arch.bootstrap.IIDBootstrap.set_state`

`IIDBootstrap.set_state(state)`

Sets the state of the bootstrap’s random number generator

Parameters `state(RandomState state vector)`—Array containing the state

`arch.bootstrap.IIDBootstrap.update_indices`

`IIDBootstrap.update_indices()`

Update indices for the next iteration of the bootstrap. This must be overridden when creating new bootstraps.

`arch.bootstrap.IIDBootstrap.var`

`IIDBootstrap.var(func, reps=1000, recenter=True, extra_kwargs=None)`

Compute parameter variance using bootstrap

Parameters

- `func(callable)`—Callable function that returns the statistic of interest as a 1-d array
- `reps(int, optional)`—Number of bootstrap replications
- `recenter(bool, optional)`—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, optional)`—Dictionary of extra keyword arguments to pass to `func`

Returns `var`—Bootstrap variance estimator

Return type `ndarray`

Notes

func must have the signature

```
def func(params, *args, **kwargs):
```
where params are a 1-dimensional array, and *args and **kwargs are data used in 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.

### Example

**Bootstrap covariance of the mean**

```python
>>> 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**

```python
>>> 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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arch.bootstrap.IIDBootstrap.index</code></td>
<td>Returns the current index of the bootstrap</td>
</tr>
<tr>
<td><code>arch.bootstrap.IIDBootstrap.random_state</code></td>
<td>Set or get the instance random state</td>
</tr>
</tbody>
</table>

**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, **kwargs)

Bootstrap the independently resamples each input

Parameters

- **args** – Positional arguments to bootstrap
- **kwargs** – Keyword arguments to bootstrap

index

The current index of the bootstrap

Type ndarray

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

random_state

RandomState instance used by bootstrap

Type RandomState

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 accessible 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.
```python
>>> 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)

```  
```python
>>> 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
```  
```python
>>> from numpy.random import RandomState

>>> rs = RandomState(1234)

>>> bs = IndependentSamplesBootstrap(x, y=y, z=z, random_state=rs)
```

See also:

`arch.bootstrap.IIDBootstrap`

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>apply(func[, reps, extra_kwvars])</code></td>
<td>Applies a function to bootstrap replicated data</td>
</tr>
<tr>
<td><code>bootstrap(reps)</code></td>
<td>Iterator for use when bootstrapping</td>
</tr>
<tr>
<td><code>clone(*args, **kwargs)</code></td>
<td>Clones the bootstrap using different data.</td>
</tr>
<tr>
<td><code>conf_int(func[, reps, method, size, tail, ...])</code></td>
<td></td>
</tr>
</tbody>
</table>

**param func** Function the computes parameter values. See Notes for requirements

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cov(func[, reps, recenter, extra_kwvars])</code></td>
<td>Compute parameter covariance using bootstrap</td>
</tr>
<tr>
<td><code>get_state()</code></td>
<td>Gets the state of the bootstrap’s random number generator</td>
</tr>
<tr>
<td><code>reset([use_seed])</code></td>
<td>Resets the bootstrap to either its initial state or the last seed.</td>
</tr>
<tr>
<td><code>seed(value)</code></td>
<td>Seeds the bootstrap’s random number generator</td>
</tr>
<tr>
<td><code>set_state(state)</code></td>
<td>Sets the state of the bootstrap’s random number generator</td>
</tr>
<tr>
<td><code>update_indices()</code></td>
<td>Update indices for the next iteration of the bootstrap.</td>
</tr>
<tr>
<td><code>var(func[, reps, recenter, extra_kwvars])</code></td>
<td>Compute parameter variance using bootstrap</td>
</tr>
</tbody>
</table>

**arch.bootstrap.IndependentSamplesBootstrap.apply**

IndependentSamplesBootstrap.**apply** *(func, reps=1000, extra_kwvars=None)*

Applies a function to bootstrap replicated data

**Parameters**

- **func** *(callable)* – Function the computes parameter values. See Notes for requirements
- **reps** *(int, optional)* – Number of bootstrap replications
• **extra_kwargs** *(dict, optional)* – Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

**Returns**

**results** – reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

**Return type**  
ndarray

**Notes**

When there are no extra keyword arguments, the function is called

```python
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**

```python
generate
>>> 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)  
Iterator for use when bootstrapping

**Parameters**

**reps** *(int)* – Number of bootstrap replications

**Returns**

**gen** – Generator to iterate over in bootstrap calculations

**Return type**  
generator

**Example**

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```python
generate
>>> 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, **kwargs)

Clones the bootstrap using different data.

**Parameters**

- **args** – Positional arguments to bootstrap
- **kwargs** – Keyword arguments to bootstrap

**Returns** Bootstrap instance

**Return type** bs

**arch.bootstrap.IndependentSamplesBootstrap.conf_int**

IndependentSamplesBootstrap.conf_int(func, reps=1000, method='basic', size=0.95, tail='two', extra_kwargs=None, reuse=False, sampling='nonparametric', std_err_func=None, studentize_reps=1000)

**Parameters**

- **func** (callable) – Function the computes parameter values. See Notes for requirements
- **reps** (int, optional) – Number of bootstrap replications
- **method** (string, optional) – One of ‘basic’, ‘percentile’, ‘studentized’, ‘norm’ (identical to ‘var’, ‘cov’), ‘bc’ (identical to ‘debiased’, ‘bias-corrected’), or ‘bca’
- **size** (float, optional) – Coverage of confidence interval
- **tail** (string, optional) – One of ‘two’, ‘upper’ or ‘lower’.
- **reuse** (bool, optional) – 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** (string, optional) – 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, optional) – Extra keyword arguments to use when calling func and std_err_func, when appropriate
- **std_err_func** (callable, optional) – 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, optional) – Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when std_err_func is provided
**Returns intervals** – 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** 2-d array

### Examples

```python
>>> 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, reps=1000, recenter=True, extra_kwArgs=None)
```

Compute parameter covariance using bootstrap

**Parameters**

- **func** (callable) – Callable function that returns the statistic of interest as a 1-d array
- **reps** (int, optional) – Number of bootstrap replications
• **recenter** *(bool, optional)* – 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, optional)* – Dictionary of extra keyword arguments to pass to `func`

**Returns**

- `cov` – Bootstrap covariance estimator

**Return type** `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 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.

**Example**

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()
```

Gets the state of the bootstrap’s random number generator

**Returns**

- `state` – Array containing the state

**Return type** `RandomState state vector`
arch.bootstrap.IndependentSamplesBootstrap.reset

IndependentSamplesBootstrap.reset(use_seed=True)
Resets the bootstrap to either its initial state or the last seed.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>use_seed</td>
<td>bool, optional</td>
<td>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</td>
</tr>
</tbody>
</table>

arch.bootstrap.IndependentSamplesBootstrap.seed

IndependentSamplesBootstrap.seed(value)
Seeds the bootstrap’s random number generator

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>int</td>
<td>Integer to use as the seed</td>
</tr>
</tbody>
</table>

arch.bootstrap.IndependentSamplesBootstrap.set_state

IndependentSamplesBootstrap.set_state(state)
Sets the state of the bootstrap’s random number generator

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>RandomState state vector</td>
<td>Array containing the state</td>
</tr>
</tbody>
</table>

arch.bootstrap.IndependentSamplesBootstrap.update_indices

IndependentSamplesBootstrap.update_indices()
Update indices for the next iteration of the bootstrap. This must be overridden when creating new bootstraps.

arch.bootstrap.IndependentSamplesBootstrap.var

IndependentSamplesBootstrap.var(func, reps=1000, recenter=True, extra_kwarg=None)
Compute parameter variance using bootstrap

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>func</td>
<td>callable</td>
<td>Callable function that returns the statistic of interest as a 1-d array</td>
</tr>
<tr>
<td>reps</td>
<td>int, optional</td>
<td>Number of bootstrap replications</td>
</tr>
<tr>
<td>recenter</td>
<td>bool, optional</td>
<td>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.</td>
</tr>
<tr>
<td>extra_kwarg</td>
<td>dict, optional</td>
<td>Dictionary of extra keyword arguments to pass to func</td>
</tr>
</tbody>
</table>

**Returns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>ndarray</td>
<td>Bootstrap variance estimator</td>
</tr>
</tbody>
</table>

2.8. Independent Samples
Notes

func must have the signature

```python
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.

Example

Bootstrap covariance of the mean

```python
>>> 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

```python
>>> 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>index</code></td>
<td>Returns the current index of the bootstrap</td>
</tr>
<tr>
<td><code>random_state</code></td>
<td>Set or get the instance random state</td>
</tr>
</tbody>
</table>

`arch.bootstrap.IndependentSamplesBootstrap.index`

`IndependentSamplesBootstrap.index` Returns the current index of the bootstrap

- **Returns**: `index` – 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]]`
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.

### 2.9.1 arch.bootstrap.StationaryBootstrap

```python
class arch.bootstrap.StationaryBootstrap(block_size, *args, **kwargs)
    Politis and Romano (1994) bootstrap with expon. distributed block sizes
```

- **block_size** (`int`) – Average size of block to use
- **args** – Positional arguments to bootstrap
- **kwargs** – Keyword arguments to bootstrap

- **index**
  - Type: `ndarray`

- **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`

- **random_state**
  - RandomState instance used by bootstrap
  - Type: `RandomState`
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.

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>apply(func[, reps, extra_kwargs])</code></td>
<td>Applies a function to bootstrap replicated data</td>
</tr>
<tr>
<td><code>bootstrap(reps)</code></td>
<td>Iterator for use when bootstrapping</td>
</tr>
<tr>
<td><code>clone(*args, **kwargs)</code></td>
<td>Clones the bootstrap using different data.</td>
</tr>
<tr>
<td><code>conf_int(func[, reps, method, size, tail, ...])</code></td>
<td>Function the computes parameter values. See Notes for requirements</td>
</tr>
<tr>
<td><code>cov(func[, reps, recenter, extra_kwargs])</code></td>
<td>Compute parameter covariance using bootstrap</td>
</tr>
<tr>
<td><code>get_state()</code></td>
<td>Gets the state of the bootstrap’s random number generator</td>
</tr>
<tr>
<td><code>reset([use_seed])</code></td>
<td>Resets the bootstrap to either its initial state or the last seed.</td>
</tr>
<tr>
<td><code>seed(value)</code></td>
<td>Seeds the bootstrap’s random number generator</td>
</tr>
<tr>
<td><code>set_state(state)</code></td>
<td>Sets the state of the bootstrap’s random number generator</td>
</tr>
<tr>
<td><code>update_indices()</code></td>
<td>Update indices for the next iteration of the bootstrap.</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 8 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var(func[, reps, recenter, extra_kwargs])</code></td>
<td>Compute parameter variance using bootstrap</td>
</tr>
</tbody>
</table>

**arch.bootstrap.StationaryBootstrap.apply**

StationaryBootstrap.apply(func, reps=1000, extra_kwargs=None)

Applies a function to bootstrap replicated data

**Parameters**

- `func (callable)` – Function that computes parameter values. See Notes for requirements
- `reps (int, optional)` – Number of bootstrap replications
- `extra_kwargs (dict, optional)` – Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

**Returns results** – reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

**Return type** `ndarray`

**Notes**

When there are no extra keyword arguments, the function is called

```python
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**

```python
>>> 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)

Iterator for use when bootstrapping

**Parameters** `reps (int)` – Number of bootstrap replications

**Returns** `gen` – Generator to iterate over in bootstrap calculations

**Return type** `generator`

**Example**
The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```python
>>> 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, **kwargs)*

Clones the bootstrap using different data.

**Parameters**

- **args** – Positional arguments to bootstrap
- **kwargs** – Keyword arguments to bootstrap

**Returns** Bootstrap instance

**Return type** bs

---

**arch.bootstrap.StationaryBootstrap.conf_int**

*StationaryBootstrap.conf_int(func, reps=1000, method='basic', size=0.95, tail='two', extra_kwargs=None, reuse=False, sampling='nonparametric', std_err_func=None, studentize_reps=1000)*

**Parameters**

- **func** (*callable*) – Function the computes parameter values. See Notes for requirements
- **reps** (*int, optional*) – Number of bootstrap replications
- **method** (*string, optional*) – One of ‘basic’, ‘percentile’, ‘studentized’, ‘norm’ (identical to ‘var’, ‘cov’), ‘bc’ (identical to ‘debiased’, ‘bias-corrected’), or ‘bca’
- **size** (*float, optional*) – Coverage of confidence interval
- **tail** (*string, optional*) – One of ‘two’, ‘upper’ or ‘lower’.
- **reuse** (*bool, optional*) – 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**(string, optional) – 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, optional) – Extra keyword arguments to use when calling func and std_err_func, when appropriate

• **std_err_func**(callable, optional) – 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, optional) – Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when std_err_func is provided

**Returns intervals** – 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** 2-d array

### Examples

```python
>>> 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

```python
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

```python
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

#### 2.9. Time-series Bootstraps
• ‘bca’ - Bias corrected and accelerated, adding acceleration parameter to ‘bc’ method

arch.bootstrap.StationaryBootstrap.cov

StationaryBootstrap.cov(func, reps=1000, recenter=True, extra_kwargs=None)
Compute parameter covariance using bootstrap

Parameters

• **func** (callable) – Callable function that returns the statistic of interest as a 1-d array
• **reps** (int, optional) – Number of bootstrap replications
• **recenter** (bool, optional) – 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, optional) – Dictionary of extra keyword arguments to pass to **func**

Returns **cov** – Bootstrap covariance estimator

Return type 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 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.

Example

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'})
```
arch.Documentation, Release 4.9.1+4.g81ceedd

**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()

Gets the state of the bootstrap’s random number generator

**Returns** state – Array containing the state

**Return type** RandomState state vector

**arch.bootstrap.StationaryBootstrap.reset**

StationaryBootstrap.reset(use_seed=True)

Resets the bootstrap to either its initial state or the last seed.

**Parameters** use_seed (bool, optional) – 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)

Seeds the bootstrap’s random number generator

**Parameters** value (int) – Integer to use as the seed

**arch.bootstrap.StationaryBootstrap.set_state**

StationaryBootstrap.set_state(state)

Sets the state of the bootstrap’s random number generator

**Parameters** state (RandomState state vector) – Array containing the state

**arch.bootstrap.StationaryBootstrap.update_indices**

StationaryBootstrap.update_indices()

Update indices for the next iteration of the bootstrap. This must be overridden when creating new bootstraps.

**arch.bootstrap.StationaryBootstrap.var**

StationaryBootstrap.var(func, reps=1000, recenter=True, extra_kwargs=None)

Compute parameter variance using bootstrap

**Parameters**

- func (callable) – Callable function that returns the statistic of interest as a 1-d array
- reps (int, optional) – Number of bootstrap replications

---

2.9. Time-series Bootstraps 189
• **recenter** *(bool, optional)* – 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, optional)* – Dictionary of extra keyword arguments to pass to func

**Returns**  
var – Bootstrap variance estimator

**Return type**  
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 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.

**Example**

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**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Returns the current index of the bootstrap</td>
</tr>
<tr>
<td>random_state</td>
<td>Set or get the instance random state</td>
</tr>
</tbody>
</table>
arch.bootstrap.StationaryBootstrap.index

StationaryBootstrap.index
Returns the current index of the bootstrap

arch.bootstrap.StationaryBootstrap.random_state

StationaryBootstrap.random_state
Set or get the instance random state

2.9.2 arch.bootstrap.CircularBlockBootstrap

class arch.bootstrap.CircularBlockBootstrap(block_size, *args, **kwargs)
Bootstrap based on blocks of the same length with end-to-start wrap around

Parameters

- **block_size** (int) – Size of block to use
- **args** – Positional arguments to bootstrap
- **kwargs** – Keyword arguments to bootstrap

index
The current index of the bootstrap
Type ndarray
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
random_state
RandomState instance used by bootstrap
Type RandomState

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.

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_kwvars])` Applies a function to bootstrap replicated data
- `bootstrap(reps)` Iterator for use when bootstrapping
- `clone(*args, **kwargs)` Clones the bootstrap using different data.
- `conf_int(func[, reps, method, size, tail, ...])`
  - `param func` Function the computes parameter values. See Notes for requirements
- `cov(func[, reps, recenter, extra_kwvars])` 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)` Seeds 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_kwvars])` Compute parameter variance using bootstrap

```
arch.bootstrap.CircularBlockBootstrap.apply

CircularBlockBootstrap.apply (func, reps=1000, extra_kwvars=None)
  Applies a function to bootstrap replicated data

Parameters

- `func (callable) – Function the computes parameter values. See Notes for requirements`
- `reps (int, optional) – Number of bootstrap replications`
```
• **extra_kwargs** *(dict, optional)* – Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

**Returns**  results – reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

**Return type**  ndarray

**Notes**

When there are no extra keyword arguments, the function is called

```python
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**

```python
>>> 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)*

Iterator for use when bootstrapping

**Parameters**  reps *(int)* – Number of bootstrap replications

**Returns**  gen – Generator to iterate over in bootstrap calculations

**Return type**  generator

**Example**

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```python
>>> 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

2.9. Time-series Bootstraps 193
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, **kwargs)
Clones the bootstrap using different data.

Parameters

• **args** – Positional arguments to bootstrap
• **kwargs** – Keyword arguments to bootstrap

Returns Bootstrap instance

Return type bs

arch.bootstrap.CircularBlockBootstrap.conf_int

CircularBlockBootstrap.conf_int(func, reps=1000, method='basic', size=0.95, tail='two', extra_kwargs=None, reuse=False, sampling='nonparametric', std_err_func=None, studentize_reps=1000)

Parameters

• **func** (callable) – Function the computes parameter values. See Notes for requirements
• **reps** (int, optional) – Number of bootstrap replications
• **method** (string, optional) – One of ‘basic’, ‘percentile’, ‘studentized’, ‘norm’ (identical to ‘var’, ‘cov’), ‘bc’ (identical to ‘debiased’, ‘bias-corrected’), or ‘bca’
• **size** (float, optional) – Coverage of confidence interval
• **tail** (string, optional) – One of ‘two’, ‘upper’ or ‘lower’.
• **reuse** (bool, optional) – 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** (string, optional) – 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, optional) – Extra keyword arguments to use when calling func and std_err_func, when appropriate
• **std_err_func** (callable, optional) – 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, optional) – Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when std_err_func is provided
**Returns intervals** – 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** 2-d array

---

### Examples

```python
>>> 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, reps=1000, recenter=True, extra_kwargs=None)`

Compute parameter covariance using bootstrap

**Parameters**

- `func (callable)` – Callable function that returns the statistic of interest as a 1-d array
- `reps (int, optional)` – Number of bootstrap replications
• **recenter** *(bool, optional)* – 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, optional)* – Dictionary of extra keyword arguments to pass to func

**Returns**

- **cov** – Bootstrap covariance estimator

**Return type**

ndarray

**Notes**

func must have the signature

```python
func(params, *args, **kwargs)
```

where params is a 1-dimensional array, and *args and **kwargs are data used in 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.

**Example**

Bootstrap covariance of the mean

```python
>>> 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

```python
>>> 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()

Gets the state of the bootstrap’s random number generator

- **Returns**

  - **state** – Array containing the state

- **Return type**

  RandomState state vector
arch.bootstrap.CircularBlockBootstrap.reset

CircularBlockBootstrap.reset\(\text{use\_seed=\text{True}}\)
Resets the bootstrap to either its initial state or the last seed.

Parameters
\text{use\_seed (bool, optional)} – 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\(\text{value}\)
Seeds the bootstrap’s random number generator

Parameters
\text{value (int)} – Integer to use as the seed

arch.bootstrap.CircularBlockBootstrap.set_state

CircularBlockBootstrap.set\_state\(\text{state}\)
Sets the state of the bootstrap’s random number generator

Parameters
\text{state (RandomState state vector)} – Array containing the state

arch.bootstrap.CircularBlockBootstrap.update_indices

CircularBlockBootstrap.update\_indices()
Update indices for the next iteration of the bootstrap. This must be overridden when creating new bootstraps.

arch.bootstrap.CircularBlockBootstrap.var

CircularBlockBootstrap.var\(\text{func, reps=1000, recenter=\text{True}, extra\_kwargs=\text{None}}\)
Compute parameter variance using bootstrap

Parameters

- \text{func (callable)} – Callable function that returns the statistic of interest as a 1-d array
- \text{reps (int, optional)} – Number of bootstrap replications
- \text{recenter (bool, optional)} – 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.
- \text{extra\_kwargs (dict, optional)} – Dictionary of extra keyword arguments to pass to func

Returns
\text{var – Bootstrap variance estimator}

Return type
\text{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 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.

Example

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

<table>
<thead>
<tr>
<th>property</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Returns the current index of the bootstrap</td>
</tr>
<tr>
<td>random_state</td>
<td>Set or get the instance random state</td>
</tr>
</tbody>
</table>

```
arch.bootstrap.CircularBlockBootstrap.index
```

CircularBlockBootstrap.index

Returns the current index of the bootstrap

```
arch.bootstrap.CircularBlockBootstrap.random_state
```

CircularBlockBootstrap.random_state

Set or get the instance random state
2.9.3 arch.bootstrap.MovingBlockBootstrap

class arch.bootstrap.MovingBlockBootstrap(block_size, *args, **kwargs)
Bootstrap based on blocks of the same length without wrap around

Parameters

- block_size (int) – Size of block to use
- args – Positional arguments to bootstrap
- kwargs – Keyword arguments to bootstrap

index
The current index of the bootstrap
Type ndarray

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

c kw_data
Dictionary containing the keyword arguments
Type dict

random_state
RandomState instance used by bootstrap
Type RandomState

Notes
Supports numpy arrays and pandas Series and DataFrames. Data returned has the same type as the input data.
Data entered using keyword arguments is directly accessible 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.

```python
>>> 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):
```

(continues on next page)
Set the random_state if reproducibility is required

```python
>>> from numpy.random import RandomState
>>> rs = RandomState(1234)
>>> bs = MovingBlockBootstrap(7, x, y=y, z=z, random_state=rs)
```

## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>apply(func[, reps, extra_kwars])</code></td>
<td>Applies a function to bootstrap replicated data</td>
</tr>
<tr>
<td><code>bootstrap(reps)</code></td>
<td>Iterator for use when bootstrapping</td>
</tr>
<tr>
<td><code>clone(*args, **kwargs)</code></td>
<td>Clones the bootstrap using different data.</td>
</tr>
<tr>
<td><code>conf_int(func[, reps, method, size, tail, ...])</code></td>
<td>Compute parameter values. See Notes for requirements</td>
</tr>
<tr>
<td><code>cov(func[, reps, recenter, extra_kwars])</code></td>
<td>Compute parameter covariance using bootstrap</td>
</tr>
<tr>
<td><code>get_state()</code></td>
<td>Gets the state of the bootstrap’s random number generator</td>
</tr>
<tr>
<td><code>reset([use_seed])</code></td>
<td>Resets the bootstrap to either its initial state or the last seed.</td>
</tr>
<tr>
<td><code>seed(value)</code></td>
<td>Seeds the bootstrap’s random number generator</td>
</tr>
<tr>
<td><code>set_state(state)</code></td>
<td>Sets the state of the bootstrap’s random number generator</td>
</tr>
<tr>
<td><code>update_indices()</code></td>
<td>Update indices for the next iteration of the bootstrap.</td>
</tr>
<tr>
<td><code>var(func[, reps, recenter, extra_kwars])</code></td>
<td>Compute parameter variance using bootstrap</td>
</tr>
</tbody>
</table>

**arch.bootstrap.MovingBlockBootstrap.apply**

MovingBlockBootstrap.apply(func, reps=1000, extra_kwars=None)
Applies a function to bootstrap replicated data

**Parameters**

- **func** (*callable*) – Function the computes parameter values. See Notes for requirements
- **reps** (*int, optional*) – Number of bootstrap replications
- **extra_kwars** (*dict, optional*) – Extra keyword arguments to use when calling func. Must not conflict with keyword arguments used to initialize bootstrap

**Returns**

- **results** – reps by nparam array of computed function values where each row corresponds to a bootstrap iteration

**Return type**

ndarray
Notes

When there are no extra keyword arguments, the function is called

```python
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

```python
>>> 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**

```python
arch.bootstrap.MovingBlockBootstrap.bootstrap(reps)
```

Iterator for use when bootstrapping

- **Parameters**
  - `reps (int)`: Number of bootstrap replications
- **Returns**
  - `gen`: Generator to iterate over in bootstrap calculations

**Return type** generator

Example

The key steps are problem dependent and so this example shows the use as an iterator that does not produce any output

```python
>>> 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

2.9. Time-series Bootstraps
arch.bootstrap.MovingBlockBootstrap.clone

MovingBlockBootstrap.clone(*args, **kwargs)
Clones the bootstrap using different data.

Parameters

• **args** – Positional arguments to bootstrap
• **kwargs** – Keyword arguments to bootstrap

Returns Bootstrap instance

Return type bs

arch.bootstrap.MovingBlockBootstrap.conf_int

MovingBlockBootstrap.conf_int(func, reps=1000, method='basic', size=0.95, tail='two', extra_kwargs=None, reuse=False, sampling='nonparametric', std_err_func=None, studentize_reps=1000)

Parameters

• **func** (callable) – Function the computes parameter values. See Notes for requirements
• **reps** (int, optional) – Number of bootstrap replications
• **method** (string, optional) – One of ‘basic’, ‘percentile’, ‘studentized’, ‘norm’ (identical to ‘var’, ‘cov’), ‘bc’ (identical to ‘debiased’, ‘bias-corrected’), or ‘bca’
• **size** (float, optional) – Coverage of confidence interval
• **tail** (string, optional) – One of ‘two’, ‘upper’ or ‘lower’.
• **reuse** (bool, optional) – 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** (string, optional) – Type of sampling to use: ‘nonparametric’, ‘semiparametric’ (or ‘semi’) or ‘parametric’. The default is ‘nonparametric’. See notes about the changes to func required when using ‘semi’ or ‘parametric’.
• **extra_kwargs** (dict, optional) – Extra keyword arguments to use when calling func and std_err_func, when appropriate
• **std_err_func** (callable, optional) – 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, optional) – Number of bootstraps to use in the inner bootstrap when using the studentized bootstrap. Ignored when std_err_func is provided

Returns intervals – 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 2-d array
Examples

```python
>>> 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, reps=1000, recenter=True, extra_kwars=None)

Compute parameter covariance using bootstrap

Parameters

- `func (callable)` – Callable function that returns the statistic of interest as a 1-d array
- `reps (int, optional)` – Number of bootstrap replications
- `recenter (bool, optional)` – 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_kwars (dict, optional)` – Dictionary of extra keyword arguments to pass to func

2.9. Time-series Bootstraps
Returns  cov – Bootstrap covariance estimator

Return type  ndarray

Notes

func must have the signature

```
func(params, *args, **kwargs)
```

where params is a 1-dimensional array, and *args and **kwargs are data used in 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.

Example

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()

Gets the state of the bootstrap’s random number generator

Returns  state – Array containing the state

Return type  RandomState state vector

arch.bootstrap.MovingBlockBootstrap.reset

MovingBlockBootstrap.reset(use_seed=True)

Resets the bootstrap to either its initial state or the last seed.
Parameters `use_seed(bool, optional)` – 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)

Seeds the bootstrap’s random number generator

Parameters `value(int)` – Integer to use as the seed

`arch.bootstrap.MovingBlockBootstrap.set_state`

MovingBlockBootstrap.set_state(state)

Sets the state of the bootstrap’s random number generator

Parameters `state(RandomState state vector)` – Array containing the state

`arch.bootstrap.MovingBlockBootstrap.update_indices`

MovingBlockBootstrap.update_indices()

Update indices for the next iteration of the bootstrap. This must be overridden when creating new bootstraps.

`arch.bootstrap.MovingBlockBootstrap.var`

MovingBlockBootstrap.var(func, reps=1000, recenter=True, extra_kwars=None)

Compute parameter variance using bootstrap

Parameters

- `func(callable)` – Callable function that returns the statistic of interest as a 1-d array
- `reps(int, optional)` – Number of bootstrap replications
- `recenter(bool, optional)` – 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_kwars(dict, optional)` – Dictionary of extra keyword arguments to pass to func

Returns `var` – Bootstrap variance estimator

Return type `ndarray`

Notes

func must have the signature

```python
func(params, *args, **kwargs)
```
where params are a 1-dimensional array, and *args and **kwargs are data used in 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.

Example

Bootstrap covariance of the mean

```python
>>> 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

```python
>>> 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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Returns the current index of the bootstrap</td>
</tr>
<tr>
<td>random_state</td>
<td>Set or get the instance random state</td>
</tr>
</tbody>
</table>

arch.bootstrap.MovingBlockBootstrap.index

MovingBlockBootstrap.index

Returns the current index of the bootstrap

arch.bootstrap.MovingBlockBootstrap.random_state

MovingBlockBootstrap.random_state

Set or get the instance random state

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
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 warnings

# Reproducability
import numpy as np
import seaborn

warnings.simplefilter('ignore')

[2]: seaborn.mpl.rcParams['figure.figsize'] = (10.0, 6.0)
seaborn.mpl.rcParams['savefig.dpi'] = 90
seaborn.mpl.rcParams['font.family'] = 'sans-serif'
seaborn.mpl.rcParams['font.size'] = 14

np.random.seed(23456)
# Common seed used throughout
seed = np.random.randint(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 snoop, 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.

```python
[3]: from numpy.random import randn
    import statsmodels.api as sm

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.

```python
[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):
        (continues on next page)
# Add measurement noise
```python
model_factors[i] = factors + randn(1000, 3)
```

# Compute regression parameters
```python
model_beta = sm.OLS(y[500:], model_factors[i, :500]).fit().params
```

# Prediction and losses
```python
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

\[ lower \leq \text{consistent} \leq upper. \]

See the original papers for more details.

```python
[5]: from arch.bootstrap import SPA

spa = SPA(bm_losses, model_losses)
spa.seed(seed)
spa.compute()
spa.pvalues
```

```python
[5]: lower 0.520
consistent 0.723
upper 0.733
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).

```python
[6]: # Save the pvalues
pvalues = []
b = 100
seeds = np.random.randint(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))
```

(continues on next page)
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)
    spa.seed(seeds[j])
    spa.compute()
pvalues.append(spa.pvalues)

Finally the pvalues can be plotted. Ideally they should form a 45° 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, .995, 100)
fig = pvalues.plot()
Power

The SPA also has power to reject when 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.

```python
# Number of models
k = 500
model_factors = np.zeros((k, t, 3))
model_losses = np.zeros((500, k))
for i in range(k):
    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)
spa.seed(seed)
spa.compute()
spa.pvalues
```

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).
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.

```python
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:

```
```
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 p-value above the size are in the MCS. Small p-values indicate that the model is easily rejected from the set that includes the best.

```python
[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)
```

(continues on next page)
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

<table>
<thead>
<tr>
<th>Model name</th>
<th>Pvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>model.60</td>
<td>0.000</td>
</tr>
<tr>
<td>model.80</td>
<td>0.000</td>
</tr>
<tr>
<td>model.140</td>
<td>0.000</td>
</tr>
<tr>
<td>model.40</td>
<td>0.001</td>
</tr>
<tr>
<td>model.20</td>
<td>0.005</td>
</tr>
<tr>
<td>model.100</td>
<td>0.008</td>
</tr>
<tr>
<td>model.120</td>
<td>0.021</td>
</tr>
<tr>
<td>model.0</td>
<td>0.021</td>
</tr>
<tr>
<td>model.220</td>
<td>0.031</td>
</tr>
<tr>
<td>model.260</td>
<td>0.116</td>
</tr>
<tr>
<td>model.240</td>
<td>0.116</td>
</tr>
<tr>
<td>model.160</td>
<td>0.136</td>
</tr>
<tr>
<td>model.200</td>
<td>0.136</td>
</tr>
<tr>
<td>model.320</td>
<td>0.446</td>
</tr>
<tr>
<td>model.180</td>
<td>0.446</td>
</tr>
<tr>
<td>model.420</td>
<td>0.478</td>
</tr>
<tr>
<td>model.400</td>
<td>0.693</td>
</tr>
<tr>
<td>model.360</td>
<td>0.889</td>
</tr>
<tr>
<td>model.340</td>
<td>0.889</td>
</tr>
<tr>
<td>model.280</td>
<td>0.889</td>
</tr>
<tr>
<td>model.460</td>
<td>0.889</td>
</tr>
<tr>
<td>model.380</td>
<td>0.889</td>
</tr>
<tr>
<td>model.300</td>
<td>0.889</td>
</tr>
<tr>
<td>model.480</td>
<td>0.889</td>
</tr>
<tr>
<td>model.440</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Included

Excluded
['0', '100', '120', '140', '20', '220', '40', '60', '80']

[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`

\[
\text{SPA}(\text{benchmark}, \text{models}[, \text{block} \_\text{size}, \text{reps}, \ldots])
\]

Implementation of the Test of Superior Predictive Ability (SPA), which is also known as the Reality Check or Bootstrap Data Snooper.
arch.bootstrap.SPA

class arch.bootstrap.SPA(benchmark, models, block_size=None, reps=1000, bootstrap='stationary', studentize=True, nested=False)

Implementation of the Test of Superior Predictive Ability (SPA), which is also known as the Reality Check or Bootstrap Data Snooper.

Parameters

- **benchmark** *(ndarray, Series)* – T element array of benchmark model losses
- **models** *(ndarray, DataFrame)* – T by k element array of alternative model losses
- **block_size** *(int, optional)* – 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, optional)* – Number of bootstrap replications to uses. Default is 1000.
- **bootstrap** *(str, optional)* – 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)* – Flag indicating to studentize loss differentials. Default is True
- **nested=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.

**compute()**

Compute the bootstrap pvalue. Must be called before accessing the pvalue

**seed()**

Pass seed to bootstrap implementation

**reset()**

Reset the bootstrap to its initial state

**better_models()**

Produce a list of column indices or names (if models is a DataFrame) that are rejected given a test size

**References**


**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 also:**

*StepM*
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>better_models([pvalue, pvalue_type])</code></td>
<td>Returns set of models rejected as being equal-or-worse than the benchmark</td>
</tr>
<tr>
<td><code>compute()</code></td>
<td>Compute the bootstrap p-value</td>
</tr>
<tr>
<td><code>critical_values([pvalue])</code></td>
<td>Returns data-dependent critical values</td>
</tr>
<tr>
<td><code>reset()</code></td>
<td>Reset the bootstrap to its initial state.</td>
</tr>
<tr>
<td><code>seed(value)</code></td>
<td>Seeds the bootstrap’s random number generator</td>
</tr>
<tr>
<td><code>subset(selector)</code></td>
<td>Sets a list of active models to run the SPA on.</td>
</tr>
</tbody>
</table>

**arch.bootstrap.SPA.better_models**

`SPA.better_models(pvalue=0.05, pvalue_type='consistent')`

Returns set of models rejected as being equal-or-worse than the benchmark

**Parameters**

- `pvalue` (*float, optional*) – P-value in (0,1) to use when computing superior models
- `pvalue_type` (*str, optional*) – String in ‘lower’, ‘consistent’, or ‘upper’ indicating which critical value to use.

**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()`

Compute the bootstrap p-value

**arch.bootstrap.SPA.critical_values**

`SPA.critical_values(pvalue=0.05)`

Returns data-dependent critical values

**Parameters**

- `pvalue` (*float, optional*) – 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** Series
arch.bootstrap.SPA.reset

SPA.reset()
Reset the bootstrap to its initial state.

arch.bootstrap.SPA.seed

SPA.seed(value)
Seeds the bootstrap’s random number generator

Parameters value (int) – Integer to use as the seed

arch.bootstrap.SPA.subset

SPA.subset(selector)
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

SPA.pvalues
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 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, ...]) Implementation of Romano and Wolf’s StepM multiple comparison procedure

arch.bootstrap.StepM

class arch.bootstrap.StepM(benchmark, models, size=0.05, block_size=None, reps=1000, bootstrap='stationary', studentize=True, nested=False)

Implementation of Romano and Wolf’s StepM multiple comparison procedure
Parameters

- **benchmark**({ndarray, Series}) – T element array of benchmark model losses
- **models**({ndarray, DataFrame}) – T by k element array of alternative model losses
- **size**({float, optional}) – Value in (0,1) to use as the test size when implementing the comparison. Default value is 0.05.
- **block_size**({int, optional}) – 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, optional}) – Number of bootstrap replications to use. Default is 1000.
- **bootstrap**({str, optional}) – 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, optional}) – Flag indicating to studentize loss differentials. Default is True
- **nested**({bool, optional}) – 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.

**compute**()
Compute the set of superior models.

References


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 also:

SPA

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>compute()</strong></td>
<td>Computes the set of superior models</td>
</tr>
<tr>
<td><strong>reset()</strong></td>
<td>Reset the bootstrap to it’s initial state.</td>
</tr>
<tr>
<td><strong>seed(value)</strong></td>
<td>Seeds the bootstrap’s random number generator</td>
</tr>
</tbody>
</table>

**arch.bootstrap.StepM.compute**

StepM.**compute**()
Computes the set of superior models
arch.bootstrap.StepM.reset

StepM.reset()
Reset the bootstrap to its initial state.

arch.bootstrap.StepM.seed

StepM.seed(value)
Seeds the bootstrap’s random number generator

Parameters value (int) – 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

StepM.superior_models
List of the indices or column names of the superior models

Returns superior_models – 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, ...]) Implementation of the Model Confidence Set (MCS)

arch.bootstrap.MCS

class arch.bootstrap.MCS(losses, size, reps=1000, block_size=None, method='R', bootstrap='stationary')
Implementation of the Model Confidence Set (MCS)

Parameters

• losses (ndarray, DataFrame) – T by k array containing losses from a set of models
• size (float, optional) – Value in (0,1) to use as the test size when implementing the mcs. Default value is 0.05.
• block_size (int, optional) – 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** (['max', 'R'], optional) – MCS test and elimination implementation method, either ‘max’ or ‘R’. Default is ‘R’.
- **reps** (int, optional) – Number of bootstrap replications to use. Default is 1000.
- **bootstrap** (str, optional) – Bootstrap to use. Options are ‘stationary’ or ‘sb’: Stationary bootstrap (Default) ‘circular’ or ‘cbb’: Circular block bootstrap ‘moving block’ or ‘mbb’: Moving block bootstrap

```python
compute()
```
Compute the set of models in the confidence set.

**References**


**Methods**

- **compute()**
  Computes the model confidence set
- **reset()**
  Reset the bootstrap to its initial state.
- **seed(value)**
  Seeds the bootstrap’s random number generator

**arch.bootstrap.MCS.compute**

MCS.compute()
Computes the model confidence set

**arch.bootstrap.MCS.reset**

MCS.reset()
Reset the bootstrap to its initial state.

**arch.bootstrap.MCS.seed**

MCS.seed(value)
Seeds the bootstrap’s random number generator

**Parameters**

- **value (int)** – 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

MCS.excluded
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

MCS.included
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

MCS.pvalues
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 DataFrame

3.3 References

Articles used in the creation of this module include
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 (arch.unitroot.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
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.

```python
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*

```python
[1]: import warnings
warnings.simplefilter('ignore')

%matplotlib inline
import seaborn

import seaborn.mpl.rcParams['figure.figsize'] = (10.0, 6.0)
seaborn.mpl.rcParams['savefig.dpi'] = 90
seaborn.mpl.rcParams['font.family'] = 'sans-serif'
seaborn.mpl.rcParams['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.

```python
import pandas as pd
import statsmodels.api as sm
import arch.data.default

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.

```python
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.

```python
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

```python
[6]:
    adf.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.

```python
[7]:
    adf.trend = 'ct'
    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 accessed using regression.

```python
[8]:
    reg_res = adf.regression
    print(reg_res.summary().as_text())
```
**OLS Regression Results**

<table>
<thead>
<tr>
<th>Dep. Variable:</th>
<th>y</th>
<th>R-squared:</th>
<th>0.095</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model:</td>
<td>OLS</td>
<td>Adj. R-squared:</td>
<td>0.090</td>
</tr>
<tr>
<td>Method:</td>
<td>Least Squares</td>
<td>F-statistic:</td>
<td>17.83</td>
</tr>
<tr>
<td>Date:</td>
<td>Wed, 28 Aug 2019</td>
<td>Prob (F-statistic):</td>
<td>1.30e-22</td>
</tr>
<tr>
<td>Time:</td>
<td>09:40:35</td>
<td>Log-Likelihood:</td>
<td>630.15</td>
</tr>
<tr>
<td>Df Model:</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Covariance Type:</td>
<td>nonrobust</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 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 |
| 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 |

**Warnings:**

[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.

```python
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.

```python
[10]: 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 (\texttt{trend='c'} or \texttt{trend='ct'}).

The results of this test agree with the ADF results.

\begin{verbatim}
[11]: 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.
\end{verbatim}

The trend can be altered using \texttt{trend}. The conclusion is the same.

\begin{verbatim}
[12]: dfgls.trend = 'ct'
print(dfgls.summary().as_text())
\end{verbatim}
Dickey–Fuller GLS Results

Test Statistic -3.464
P-value 0.009
Lags 21

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`.

```python
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.

```python
pp.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.
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'.

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.

Changing the trend does not alter the conclusion.
```python
[18]: kpss.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.

```python
[19]: 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.

```python
[20]: import numpy as np
    import pandas as pd
    import arch.data.frenchdata
    ff = arch.data.frenchdata.load()
    excess_market = ff.iloc[:, 0]  # Excess Market
    print(ff.describe())

```

(continues on next page)
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).

```python
[21]: from arch.unitroot import VarianceRatio
define 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.

```python
[22]: warnings.simplefilter('always') # Restore warnings
    vr.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

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 GLS version of the Dickey-Fuller test

**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($y$, lags=None, trend='c', max_lags=None, method='AIC', low_memory=None)

Augmented Dickey-Fuller unit root test

**Parameters**

- **y** *(ndarray, Series)* – The data to test for a unit root
- **lags** *(int, optional)* – 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** *({'nc', 'c', 'ct', 'ctt'}, optional)* – The trend component to include in the ADF test. ‘nc’ - 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, optional)* – The maximum number of lags to use when selecting lag length
- **method** *({'AIC', 'BIC', 't-stat'}, optional)* – 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)* – 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.

stat
pvalue
critical_values
null_hypothesis
alternative_hypothesis
summary
regression
valid_trends
y
trend

4.3. The Unit Root Tests 237
lags

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 p-value 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

```python
>>> 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

```python
summary() Summary of test, containing statistic, p-value and
critical values
```

arch.unitroot.ADF.summary

ADF.summary() Summary of test, containing statistic, p-value and critical values

Properties
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>alternative_hypothesis</code></td>
<td>The alternative hypothesis</td>
</tr>
<tr>
<td><code>critical_values</code></td>
<td>Dictionary containing critical values specific to the test, number of</td>
</tr>
<tr>
<td></td>
<td>observations and included deterministic trend terms.</td>
</tr>
<tr>
<td><code>lags</code></td>
<td>Sets or gets the number of lags used in the model.</td>
</tr>
<tr>
<td><code>max_lags</code></td>
<td>Sets or gets the maximum lags used when automatically selecting lag length.</td>
</tr>
<tr>
<td><code>nobs</code></td>
<td>The number of observations used when computing the test statistic.</td>
</tr>
<tr>
<td><code>null_hypothesis</code></td>
<td>The null hypothesis</td>
</tr>
<tr>
<td><code>pvalue</code></td>
<td>Returns the p-value for the test statistic</td>
</tr>
<tr>
<td><code>regression</code></td>
<td>Returns the OLS regression results from the ADF model estimated</td>
</tr>
<tr>
<td><code>stat</code></td>
<td>The test statistic for a unit root</td>
</tr>
<tr>
<td><code>trend</code></td>
<td>Sets or gets the deterministic trend term used in the test.</td>
</tr>
<tr>
<td><code>valid_trends</code></td>
<td>List of valid trend terms.</td>
</tr>
<tr>
<td><code>y</code></td>
<td>Returns the data used in the test statistic</td>
</tr>
</tbody>
</table>

**arch.unitroot.ADF.alternative_hypothesis**

`ADF.alternative_hypothesis`

The alternative hypothesis

**arch.unitroot.ADF.critical_values**

`ADF.critical_values`

Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

**arch.unitroot.ADF.lags**

`ADF.lags`

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**

`ADF.max_lags`

Sets or gets the maximum lags used when automatically selecting lag length

**arch.unitroot.ADF.nobs**

`ADF.nobs`

The number of observations used when computing the test statistic. Accounts for loss of data due to lags for regression-based bootstrap.
archDocumentation, Release 4.9.1+4.g81ceedd

arch.unitroot.ADF.null_hypothesis

ADF.null_hypothesis
The null hypothesis

arch.unitroot.ADF.pvalue

ADF.pvalue
Returns the p-value for the test statistic

arch.unitroot.ADF.regression

ADF.regression
Returns the OLS regression results from the ADF model estimated

arch.unitroot.ADF.stat

ADF.stat
The test statistic for a unit root

arch.unitroot.ADF.trend

ADF.trend
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

ADF.valid_trends
List of valid trend terms.

arch.unitroot.ADF.y

ADF.y
Returns the data used in the test statistic

4.3.2 arch.unitroot.DFGLS

class arch.unitroot.DFGLS(y, lags=None, trend='c', max_lags=None, method='AIC', low_memory=None)
Elliott, Rothenberg and Stock’s GLS version of the Dickey-Fuller test

Parameters

• y (ndarray, Series) – The data to test for a unit root
• lags (int, optional) – 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'], optional) – The trend component to include in the ADF test ‘c’ - Include a constant (Default) ‘ct’ - Include a constant and linear time trend
• **max_lags** *(int, optional)* – The maximum number of lags to use when selecting lag length

• **method** *({'AIC', 'BIC', 't-stat'}, optional)* – 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

```python
>>> 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 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**

DFGLS . alternative_hypothesis
The alternative hypothesis

**arch.unitroot.DFGLS.critical_values**

DFGLS . critical_values
Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.
arch.unitroot.DFGLS.lags

DFGLS.lags
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

DFGLS.max_lags
Sets or gets the maximum lags used when automatically selecting lag length.

arch.unitroot.DFGLS.nobs

DFGLS.nobs
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

DFGLS.null_hypothesis
The null hypothesis.

arch.unitroot.DFGLS.pvalue

DFGLS.pvalue
Returns the p-value for the test statistic.

arch.unitroot.DFGLS.regression

DFGLS.regression
Returns the OLS regression results from the ADF model estimated.

arch.unitroot.DFGLS.stat

DFGLS.stat
The test statistic for a unit root.

arch.unitroot.DFGLS.trend

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

DFGLS.valid_trends
List of valid trend terms.
arch.unitroot.DFGLS.y

DFGLS.y

Returns the data used in the test statistic

### 4.3.3 arch.unitroot.PhillipsPerron

class arch.unitroot.PhillipsPerron(y, lags=None, trend='c', test_type='tau')

Phillips-Perron unit root test

**Parameters**

- **y** *(ndarray, Series)* – The data to test for a unit root
- **lags** *(int, optional)* – 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** *({'nc', 'c', 'ct'}, optional)* – The trend component to include in the ADF test
  - ‘nc’ - No trend components
  - ‘c’ - Include a constant (Default)
  - ‘ct’ - Include a constant and linear time trend
- **test_type** *({'tau', 'rho'})* – 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

```python
>>> 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
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 of test, containing statistic, p-value and critical values

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>alternative_hypothesis</code></td>
<td>The alternative hypothesis</td>
</tr>
<tr>
<td><code>critical_values</code></td>
<td>Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.</td>
</tr>
<tr>
<td><code>lags</code></td>
<td>Sets or gets the number of lags used in the model.</td>
</tr>
<tr>
<td><code>nobs</code></td>
<td>The number of observations used when computing the test statistic.</td>
</tr>
<tr>
<td><code>null_hypothesis</code></td>
<td>The null hypothesis</td>
</tr>
<tr>
<td><code>pvalue</code></td>
<td>Returns the p-value for the test statistic</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 7 – continued from previous page

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stat</code></td>
<td>The test statistic for a unit root</td>
</tr>
<tr>
<td><code>test_type</code></td>
<td>Gets or sets the test type returned by <code>stat</code>.</td>
</tr>
<tr>
<td><code>trend</code></td>
<td>Sets or gets the deterministic trend term used in the test.</td>
</tr>
<tr>
<td><code>valid_trends</code></td>
<td>List of valid trend terms.</td>
</tr>
<tr>
<td><code>y</code></td>
<td>Returns the data used in the test statistic</td>
</tr>
</tbody>
</table>

---

**arch.unitroot.PhillipsPerron.alternative_hypothesis**

`PhillipsPerron.alternative_hypothesis`

The alternative hypothesis

**arch.unitroot.PhillipsPerron.critical_values**

`PhillipsPerron.critical_values`

Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

**arch.unitroot.PhillipsPerron.lags**

`PhillipsPerron.lags`

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**

`PhillipsPerron.nobs`

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**

`PhillipsPerron.null_hypothesis`

The null hypothesis

**arch.unitroot.PhillipsPerron.pvalue**

`PhillipsPerron.pvalue`

Returns the p-value for the test statistic

**arch.unitroot.PhillipsPerron.stat**

`PhillipsPerron.stat`

The test statistic for a unit root
arch.unitroot.PhillipsPerron.test_type

PhillipsPerron.test_type
    Gets or sets the test type returned by stat. Valid values are ‘tau’ or ‘rho’

arch.unitroot.PhillipsPerron.trend

PhillipsPerron.trend
    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

PhillipsPerron.valid_trends
    List of valid trend terms.

arch.unitroot.PhillipsPerron.y

PhillipsPerron.y
    Returns the data used in the test statistic

4.3.4 arch.unitroot.ZivotAndrews

class arch.unitroot.ZivotAndrews(y, lags=None, trend='c', trim=0.15, max_lags=None, method='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 (array_like) – data series
• lags (int, optional) – 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 ({'nc', 'c', 'ct', 'ctt'}, optional) – The trend component to include in the Zivot-Andrews test ‘c’ - Include a constant (Default) ‘t’ - Include a linear time trend ‘ct’ - Include a constant and linear time trend
• trim (float) – percentage of series at begin/end to exclude from break-period calculation in range [0, 0.333] (default=0.15)
• max_lags (int, optional) – The maximum number of lags to use when selecting lag length
• method ({'AIC', 'BIC', 't-stat'}, optional) – 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

stat
pvalue
critical_values

4.3. The Unit Root Tests
null_hypothesis
alternative_hypothesis
summary
regression
valid_trends
y
trend
lags

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 of test, containing statistic, p-value and critical values

Properties

<table>
<thead>
<tr>
<th>alternative_hypothesis</th>
<th>The alternative hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical_values</td>
<td>Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.</td>
</tr>
<tr>
<td>lags</td>
<td>Sets or gets the number of lags used in the model.</td>
</tr>
<tr>
<td>nobs</td>
<td>The number of observations used when computing the test statistic.</td>
</tr>
<tr>
<td>null_hypothesis</td>
<td>The null hypothesis</td>
</tr>
<tr>
<td>pvalue</td>
<td>Returns the p-value for the test statistic</td>
</tr>
<tr>
<td>stat</td>
<td>The test statistic for a unit root</td>
</tr>
</tbody>
</table>

Continued on next page
Table 9 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>trend</code></td>
<td>Sets or gets the deterministic trend term used in the test.</td>
</tr>
<tr>
<td><code>valid_trends</code></td>
<td>List of valid trend terms.</td>
</tr>
<tr>
<td><code>y</code></td>
<td>Returns the data used in the test statistic</td>
</tr>
</tbody>
</table>

### `arch.unitroot.ZivotAndrews.alternative_hypothesis`

A list of valid trend terms.

The alternative hypothesis

### `arch.unitroot.ZivotAndrews.critical_values`

Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

### `arch.unitroot.ZivotAndrews.lags`

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`

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`

The null hypothesis

### `arch.unitroot.ZivotAndrews.pvalue`

Returns the p-value for the test statistic

### `arch.unitroot.ZivotAndrews.stat`

The test statistic for a unit root

### `arch.unitroot.ZivotAndrews.trend`

Sets or gets the deterministic trend term used in the test. See valid_trends for a list of supported trends

---

4.3. The Unit Root Tests
archDocumentation, Release 4.9.1+4.g81ceedd

```
arch unitroot ZivotAndrews valid trends

ZivotAndrews.valid_trends
List of valid trend terms.

arch unitroot ZivotAndrews y

ZivotAndrews y
Returns the data used in the test statistic

4.3.5 arch unitroot VarianceRatio

class arch unitroot VarianceRatio (y, lags=2, trend='c', debiased=True, robust=True, overlap=True)

Variance Ratio test of a random walk.

Parameters
- y (ndarray, Series) – The data to test for a random walk
- lags (int) – 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 ('nc', 'c'), optional) – ‘c’ allows for a non-zero drift in the random walk, while ‘nc’ requires that the increments to y are mean 0
- overlap (bool, optional) – 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, optional) – Indicates whether to use heteroskedasticity robust inference. Default is True.
- debiased (bool, optional) – Indicates whether to use a debiased version of the test. Default is True. Only applicable if overlap is True.

stat
pvalue
critical_values
null_hypothesis
alternative_hypothesis
summary
valid_trends
y
trend
lags
overlap
robust
debiased
```
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

```python
>>> from arch.unitroot import VarianceRatio
>>> import datetime as dt
>>> import pandas_datareader as pdr

>>> data = pdr.get_data_fred('DJIA')
>>> data = data.resample('M').last()  # End of month
>>> returns = data['DJIA'].pct_change().dropna()
>>> vr = VarianceRatio(returns, lags=12)
>>> print('{:.04f}'.format(vr.pvalue))
0.0000
```

References

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>summary()</code></td>
<td>Summary of test, containing statistic, p-value and critical values</td>
</tr>
</tbody>
</table>

**arch.unitroot.VarianceRatio.summary**

VarianceRatio . `summary()`

Summary of test, containing statistic, p-value and critical values

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>alternate_hypothesis</code></td>
<td>The alternative hypothesis</td>
</tr>
<tr>
<td><code>critical_values</code></td>
<td>Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.</td>
</tr>
<tr>
<td><code>debiased</code></td>
<td>Sets of gets the indicator to use debiased variances in the ratio</td>
</tr>
<tr>
<td><code>lags</code></td>
<td>Sets or gets the number of lags used in the model.</td>
</tr>
<tr>
<td><code>nobs</code></td>
<td>The number of observations used when computing the test statistic.</td>
</tr>
<tr>
<td><code>null_hypothesis</code></td>
<td>The null hypothesis</td>
</tr>
<tr>
<td><code>overlap</code></td>
<td>Sets of gets the indicator to use overlapping returns in the long-period variance estimator</td>
</tr>
<tr>
<td><code>pvalue</code></td>
<td>Returns the p-value for the test statistic</td>
</tr>
<tr>
<td><code>robust</code></td>
<td>Sets of gets the indicator to use a heteroskedasticity robust variance estimator</td>
</tr>
<tr>
<td><code>stat</code></td>
<td>The test statistic for a unit root</td>
</tr>
</tbody>
</table>

Continued on next page
Table 11 – continued from previous page

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>trend</td>
<td>Sets or gets the deterministic trend term used in the test.</td>
</tr>
<tr>
<td>valid_trends</td>
<td>List of valid trend terms.</td>
</tr>
<tr>
<td>vr</td>
<td>The ratio of the long block lags-period variance to the 1-period variance</td>
</tr>
<tr>
<td>y</td>
<td>Returns the data used in the test statistic</td>
</tr>
</tbody>
</table>

**arch.unitroot.VarianceRatio.alternative_hypothesis**

VarianceRatio.alternative_hypothesis

The alternative hypothesis

**arch.unitroot.VarianceRatio.critical_values**

VarianceRatio.critical_values

Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

**arch.unitroot.VarianceRatio.debiased**

VarianceRatio.debiased

Sets of gets the indicator to use debiased variances in the ratio

**arch.unitroot.VarianceRatio.lags**

VarianceRatio.lags

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**

VarianceRatio.nobs

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**

VarianceRatio.null_hypothesis

The null hypothesis

**arch.unitroot.VarianceRatio.overlap**

VarianceRatio.overlap

Sets of gets the indicator to use overlapping returns in the long-period variance estimator
arch.unitroot.VarianceRatio.pvalue

VarianceRatio.pvalue
Returns the p-value for the test statistic

arch.unitroot.VarianceRatio.robust

VarianceRatio.robust
Sets or gets the indicator to use a heteroskedasticity robust variance estimator

arch.unitroot.VarianceRatio.stat

VarianceRatio.stat
The test statistic for a unit root

arch.unitroot.VarianceRatio.trend

VarianceRatio.trend
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

VarianceRatio.valid_trends
 List of valid trend terms.

arch.unitroot.VarianceRatio.vr

VarianceRatio.vr
The ratio of the long block lags-period variance to the 1-period variance

arch.unitroot.VarianceRatio.y

VarianceRatio.y
Returns the data used in the test statistic

4.3.6 arch.unitroot.KPSS

class arch.unitroot.KPSS(y, lags=None, trend='c')
Kwiatkowski, Phillips, Schmidt and Shin (KPSS) stationarity test

Parameters

- y(ndarray, Series) – The data to test for stationarity
- lags(int, optional) – 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'}, optional) –
The trend component to include in the ADF test ‘c’ - Include a constant (Default) ‘ct’ - Include a constant and linear time trend

stat
pvalue
critical_values
null_hypothesis
alternative_hypothesis
summary
valid_trends
y
trend
lags

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

```python
>>> 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:.4f}'.format(kpss.stat))
0.2870
>>> print('{0:.4f}'.format(kpss.pvalue))
0.1473
>>> kpss.trend = 'ct'
>>> print('{0:.4f}'.format(kpss.stat))
0.2075
>>> print('{0:.4f}'.format(kpss.pvalue))
0.0128
```

References

Methods
Summary of test, containing statistic, p-value and critical values

arch.unitroot.KPSS.summary

KPSS.summary()
Summary of test, containing statistic, p-value and critical values

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alternative_hypothesis</td>
<td>The alternative hypothesis</td>
</tr>
<tr>
<td>critical_values</td>
<td>Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.</td>
</tr>
<tr>
<td>lags</td>
<td>Sets or gets the number of lags used in the model.</td>
</tr>
<tr>
<td>nobss</td>
<td>The number of observations used when computing the test statistic.</td>
</tr>
<tr>
<td>null_hypothesis</td>
<td>The null hypothesis</td>
</tr>
<tr>
<td>pvalue</td>
<td>Returns the p-value for the test statistic.</td>
</tr>
<tr>
<td>stat</td>
<td>The test statistic for a unit root</td>
</tr>
<tr>
<td>trend</td>
<td>Sets or gets the deterministic trend term used in the test.</td>
</tr>
<tr>
<td>valid_trends</td>
<td>List of valid trend terms.</td>
</tr>
<tr>
<td>y</td>
<td>Returns the data used in the test statistic.</td>
</tr>
</tbody>
</table>

arch.unitroot.KPSS.alternative_hypothesis

KPSS.alternative_hypothesis
The alternative hypothesis

arch.unitroot.KPSS.critical_values

KPSS.critical_values
Dictionary containing critical values specific to the test, number of observations and included deterministic trend terms.

arch.unitroot.KPSS.lags

KPSS.lags
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

KPSS.nobs
The number of observations used when computing the test statistic. Accounts for loss of data due to lags for regression-based bootstrap.

4.3. The Unit Root Tests
**arch.unitroot.KPSS.null_hypothesis**

**KPSS.null_hypothesis**
The null hypothesis

**arch.unitroot.KPSS.pvalue**

**KPSS.pvalue**
Returns the p-value for the test statistic

**arch.unitroot.KPSS.stat**

**KPSS.stat**
The test statistic for a unit root

**arch.unitroot.KPSS.trend**

**KPSS.trend**
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**

**KPSS.valid_trends**
List of valid trend terms.

**arch.unitroot.KPSS.y**

**KPSS.y**
Returns the data used in the test statistic

### 4.3.7 Automatic Bandwidth Selection

**auto_bandwidth**

**auto_bandwidth**(y[, kernel])

**arch.unitroot.auto_bandwidth**

**arch.unitroot.auto_bandwidth**(y, kernel='ba')

**Parameters**

- **y**(ndarray, Series) – Data on which to apply the bandwidth selection
- **kernel**(str) – 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
5.1 Version 4

5.1.1 Changes Since 4.9

- Fixed a bug in arch_lm_test that assumed that the model data is contained in a pandas Series. (GH313).

5.1.2 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).
5.1.3 Release 4.8.1

- Fixed a bug which prevented extension modules from being correctly imported.

5.1.4 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).

5.1.5 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 bug where parameter-less models where incorrectly reported as having constant variance (GH248).

5.1.6 Release 4.6

- Added support for MIDAS volatility processes using Hyperbolic weighting in `MidasHyperbolic` (GH233).

5.1.7 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.

5.1.8 Release 4.4

- Added named parameters to Dickey-Fuller regressions.
- Removed use of the module-level NumPy RandomState. All random number generators use separate Random-State 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.
5.1.9 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.

5.1.10 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).

5.2 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`.

5.3 Version 2

5.3.1 Version 2.2

- Added multiple comparison procedures
- Typographical and other small changes

5.3.2 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
5.4 Version 1

5.4.1 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`
This package should be cited using Zenodo. For example, for the 4.8.1 release,
CHAPTER 7

Index

• genindex
• modindex
Bibliography


a
arch.bootstrap, 151
arch.bootstrap.multiple_comparison, 217
arch.unitroot, 236
arch.utility.testing, 148
Index

A

ADF (class in arch.unitroot), 237
alternative_hypothesis (arch.unitroot.ADF attribute), 237, 239
alternative_hypothesis (arch.unitroot.DFGLS attribute), 241, 242
alternative_hypothesis (arch.unitroot.KPSS attribute), 254, 255
alternative_hypothesis (arch.unitroot.PhilipsPerron attribute), 244, 246
alternative_hypothesis (arch.unitroot.VarianceRatio attribute), 250, 252
alternative_hypothesis (arch.unitroot.ZivotAndrews attribute), 248, 249
apply() (arch.bootstrap.CircularBlockBootstrap method), 192
apply() (arch.bootstrap.IIDBootstrap method), 168
apply() (arch.bootstrap.IndependentSamplesBootstrap method), 176
apply() (arch.bootstrap.MovingBlockBootstrap method), 200
apply() (arch.bootstrap.StationaryBootstrap method), 185
ARCH (class in arch.univariate), 111
arch.bootstrap (module), 151
arch.bootstrap.multiple_comparison (module), 217
arch.unitroot (module), 236
arch.utility.testing (module), 148
arch_lm_test() (arch.univariate.base.ARCHModelFixedResult method), 11
arch_lm_test() (arch.univariate.base.ARCHModelResult method), 7
arch_model() (in module arch), 5
ARCHModel (class in arch.univariate.base), 82
ARCHModelFixedResult (class in arch.univariate.base), 10
ARCHModelForecast (class in arch.univariate.base), 29
ARCHModelForecastSimulation (class in arch.univariate.base), 30
ARCHModelResult (class in arch.univariate.base), 6
ARX (class in arch.univariate), 62
auto_bandwidth() (in module arch.unitroot), 256

B

backcast() (arch.univariate.ARCH method), 112
backcast() (arch.univariate.ConstantVariance method), 83
backcast() (arch.univariate.EGARCH method), 98
backcast() (arch.univariate.EWMAVariance method), 117
backcast() (arch.univariate.FIGARCH method), 93
backcast() (arch.univariate.FixedVariance method), 126
backcast() (arch.univariate.GARCH method), 88
backcast() (arch.univariate.HARCH method), 103
backcast() (arch.univariate.MIDASHyperbolic method), 107
backcast() (arch.univariate.RiskMetrics2006 method), 121
backcast_transform() (arch.univariate.ARCH method), 112
backcast_transform() (arch.univariate.ConstantVariance method), 83
backcast_transform() (arch.univariate.EGARCH method), 98
backcast_transform() (arch.univariate.EWMAVariance method), 117
backcast_transform() (arch.univariate.FIGARCH method), 93
backcast_transform() (arch.univariate.FixedVariance method), 126
backcast_transform() (arch.univariate.GARCH method), 88
backcast_transform() (arch.univariate.HARCH method), 103
backcast_transform() (arch.univariate.MIDASHyperbolic method), 107
backcast_transform() (arch.univariate.RiskMetrics2006 method), 121
better_models() (arch.bootstrap.SPA method), 218, 219
bootstrap() (arch.bootstrap.CircularBlockBootstrap method), 193
bootstrap() (arch.bootstrap.IIDBootstrap method), 169
bootstrap() (arch.bootstrap.IndependentSamplesBootstrap method), 177
bootstrap() (arch.bootstrap.MovingBlockBootstrap method), 201
bootstrap() (arch.bootstrap.StationaryBootstrap method), 185
bounds() (arch.univariate.ARCH method), 112
bounds() (arch.univariate.ARX method), 63
bounds() (arch.univariate.ConstantMean method), 56
bounds() (arch.univariate.ConstantVariance method), 84
bounds() (arch.univariate.EGARCH method), 98
bounds() (arch.univariate.EWMAVariance method), 117
bounds() (arch.univariate.FIGARCH method), 94
bounds() (arch.univariate.FixedVariance method), 126
bounds() (arch.univariate.GARCH method), 89
bounds() (arch.univariate.GeneralizedError method), 146
bounds() (arch.univariate.HARCH method), 103
bounds() (arch.univariate.HARX method), 70
bounds() (arch.univariate.LS method), 77
bounds() (arch.univariate.MIDASHyperbolic method), 108
bounds() (arch.univariate.Normal method), 137
bounds() (arch.univariate.RiskMetrics2006 method), 122
bounds() (arch.univariate.SkewStudent method), 143
bounds() (arch.univariate.StudentsT method), 139
bounds() (arch.univariate.ZeroMean method), 50

cdf() (arch.univariate.GeneralizedError method), 146
cdf() (arch.univariate.Normal method), 137
cdf() (arch.univariate.SkewStudent method), 143
cdf() (arch.univariate.StudentsT method), 140
CircularBlockBootstrap (class in arch.bootstrap), 191
clone() (arch.bootstrap.CircularBlockBootstrap method), 194
clone() (arch.bootstrap.IIDBootstrap method), 170
clone() (arch.bootstrap.IndependentSamplesBootstrap method), 178
clone() (arch.bootstrap.MovingBlockBootstrap method), 202
clone() (arch.bootstrap.StationaryBootstrap method), 186
compute() (arch.bootstrap.MCS method), 223
compute() (arch.bootstrap.SPA method), 218, 219
compute() (arch.bootstrap.StepM method), 221
compute_param_cov() (arch.univariate.ARX method), 63
compute_param_cov() (arch.univariate.ConstantMean method), 57
compute_param_cov() (arch.univariate.HARCH method), 70
compute_param_cov() (arch.univariate.LS method), 77
compute_param_cov() (arch.univariate.ZeroMean method), 50
compute_variance() (arch.univariate.ARCH method), 112
compute_variance() (arch.univariate.ConstantVariance method), 84
compute_variance() (arch.univariate.EGARCH method), 99
compute_variance() (arch.univariate.EWMAVariance method), 117
compute_variance() (arch.univariate.FIGARCH method), 94
compute_variance() (arch.univariate.FixedVariance method), 126
compute_variance() (arch.univariate.GARCH method), 89
compute_variance() (arch.univariate.HARCH method), 103
compute_variance() (arch.univariate.MIDASHyperbolic method), 108
compute_variance() (arch.univariate.RiskMetrics2006 method), 122
conf_int() (arch.bootstrap.CircularBlockBootstrap method), 194
conf_int() (arch.bootstrap.IIDBootstrap method), 170
conf_int() (arch.bootstrap.IndependentSamplesBootstrap method), 178
conf_int() (arch.bootstrap.MovingBlockBootstrap method), 202
method), 202  
conf_int() (arch.bootstrap.StationaryBootstrap method), 186  
conf_int() (arch.univariate.base.ARCHModelResult method), 7  
ConstantMean (class in arch.univariate), 55  
ConstantVariance (class in arch.univariate), 83  
constraints() (arch.univariate.ARCH method), 113  
constraints() (arch.univariate.ARX method), 63  
constraints() (arch.univariate.ConstantMean method), 57  
constraints() (arch.univariate.ConstantVariance method), 84  
constraints() (arch.univariate.EGARCH method), 99  
constraints() (arch.univariate.EWMAVariance method), 117  
constraints() (arch.univariate.FIGARCH method), 94  
constraints() (arch.univariate.FixedVariance method), 127  
constraints() (arch.univariate.GARCH method), 89  
constraints() (arch.univariate.GeneralizedError method), 146  
constraints() (arch.univariate.HARCH method), 103  
constraints() (arch.univariate.HARX method), 70  
constraints() (arch.univariate.LS method), 77  
constraints() (arch.univariate.MIDASHyperbolic method), 108  
constraints() (arch.univariate.Normal method), 137  
constraints() (arch.univariate.RiskMetrics2006 method), 122  
constraints() (arch.univariate.SkewStudent method), 143  
constraints() (arch.univariate.StudentsT method), 140  
constraints() (arch.univariate.ZeroMean method), 50  
cov() (arch.bootstrap.CircularBlockBootstrap method), 195  
cov() (arch.bootstrap.IIDBootstrap method), 171  
cov() (arch.bootstrap.IndependentSamplesBootstrap method), 179  
cov() (arch.bootstrap.MovingBlockBootstrap method), 203  
cov() (arch.bootstrap.StationaryBootstrap method), 188  
critical_values (arch.unitroot.ADF attribute), 237, 239  
critical_values (arch.unitroot.DFGLS attribute), 241, 242  
critical_values (arch.unitroot.KPSS attribute), 254, 255  
critical_values (arch.unitroot.PhilipsPerron attribute), 244, 246  
critical_values (arch.unitroot.VarianceRatio attribute), 250, 252  
critical_values (arch.unitroot.ZivotAndrews attribute), 247, 249  
critical_values (arch.utility.testing.WaldTestStatistic attribute), 149  
critical_values() (arch.bootstrap.SPA method), 219  
data (arch.bootstrap.CircularBlockBootstrap attribute), 191  
data (arch.bootstrap.IIDBootstrap attribute), 167  
data (arch.bootstrap.IndependentSamplesBootstrap attribute), 175  
data (arch.bootstrap.MovingBlockBootstrap attribute), 199  
data (arch.bootstrap.StationaryBootstrap attribute), 183  
debiased (arch.unitroot.VarianceRatio attribute), 250, 252  
DFGLS (class in arch.unitroot), 240  
distribution (arch.univariate.ARX attribute), 68  
distribution (arch.univariate.ConstantMean attribute), 61  
distribution (arch.univariate.HARX attribute), 75  
distribution (arch.univariate.LS attribute), 81  
distribution (arch.univariate.ZeroMean attribute), 55  
Distribution (class in arch.univariate.distribution), 148  
EGARCH (class in arch.univariate), 97  
EWMAVariance (class in arch.univariate), 116  
excluded (arch.bootstrap.MCS attribute), 224  
FIGARCH (class in arch.univariate), 92  
fitted() (arch.univariate.ARX method), 64  
fitted() (arch.univariate.ConstantMean method), 57  
fitted() (arch.univariate.HARX method), 71  
fitted() (arch.univariate.LS method), 77  
fitted() (arch.univariate.ZeroMean method), 51  
fix() (arch.univariate.ARX method), 64  
fix() (arch.univariate.ConstantMean method), 58  
fix() (arch.univariate.HARX method), 71  
fix() (arch.univariate.LS method), 78  
FixedVariance (class in arch.univariate), 125  
forecast() (arch.univariate.ARCH method), 113
nobs (arch.univariate.ADF attribute), 239
nobs (arch.univariate.DFGLS attribute), 243
nobs (arch.univariate.KPSS attribute), 255
nobs (arch.univariate.PhillipsPerron attribute), 246
nobs (arch.univariate.VarianceRatio attribute), 252
nobs (arch.univariate.ZivotAndrews attribute), 249
Normal (class in arch.univariate), 136
null (arch.utility.testing.WaldTestStatistic attribute), 149
null_hypothesis (arch.univariate.ADF attribute), 237, 240
null_hypothesis (arch.univariate.DFGLS attribute), 241, 243
null_hypothesis (arch.univariate.KPSS attribute), 254, 256
null_hypothesis (arch.univariate.PhillipsPerron attribute), 244, 246
null_hypothesis (arch.univariate.VarianceRatio attribute), 250, 252
null_hypothesis (arch.univariate.ZivotAndrews attribute), 247, 249
num_params (arch.univariate.ARCH attribute), 111
num_params (arch.univariate.ARX attribute), 68
num_params (arch.univariate.ConstantMean attribute), 61
num_params (arch.univariate.EGARCH attribute), 97
num_params (arch.univariate.EWMAVariance attribute), 116
num_params (arch.univariate.FIGARCH attribute), 92
num_params (arch.univariate.HARCH attribute), 87
num_params (arch.univariate.HARX attribute), 101
num_params (arch.univariate.LS attribute), 82
num_params (arch.univariate.MIDASHyperbolic attribute), 106
num_params (arch.univariate.RiskMetrics2006 attribute), 120
num_params (arch.univariate.ZeroMean attribute), 55

O
overlap (arch.univariate.VarianceRatio attribute), 250, 252

P
param_cov (arch.univariate.base.ARCHModelResult attribute), 7
parameter_names () (arch.univariate.ARCH method), 114
parameter_names () (arch.univariate.ARX method), 66
parameter_names () (arch.univariate.ConstantMean method), 60
parameter_names () (arch.univariate.ConstantVariance method), 85
parameter_names () (arch.univariate.EGARCH method), 100
parameter_names () (arch.univariate.EWMAVariance method), 119
parameter_names () (arch.univariate.FIGARCH method), 95
parameter_names () (arch.univariate.FixedVariance method), 128
parameter_names () (arch.univariate.GARCH method), 90
parameter_names () (arch.univariate.GeneratlzedError method), 147
parameter_names () (arch.univariate.HARCH method), 105
parameter_names () (arch.univariate.HARX method), 73
parameter_names () (arch.univariate.LS method), 80
parameter_names () (arch.univariate.MIDASHyperbolic method), 109
parameter_names () (arch.univariate.Normal method), 138
parameter_names () (arch.univariate.RiskMetrics2006 method), 123
parameter_names () (arch.univariate.SkewStudent method), 144
parameter_names () (arch.univariate.StudentsT method), 141
parameter_names () (arch.univariate.ZeroMean method), 53
params (arch.univariate.base.ARCHModelFixedResult attribute), 10
params (arch.univariate.base.ARCHModelResult attribute), 7
PhillipsPerron (class in arch.unitroot), 244
plot () (arch.univariate.base.ARCHModelFixedResult method), 10, 13
plot () (arch.univariate.base.ARCHModelResult method), 7, 9
pos_data (arch.bootstrap.CircularBlockBootstrap attribute), 191
pos_data (arch.bootstrap.IIDBootstrap attribute), 167
pos_data (arch.bootstrap.IndependentSamplesBootstrap attribute), 175
pos_data (arch.bootstrap.MovingBlockBootstrap attribute), 199
pos_data (arch.bootstrap.StationaryBootstrap attribute), 183

Index 275
ppf() (arch.univariate.GeneralizedError method), 147
ppf() (arch.univariate.Normal method), 138
ppf() (arch.univariate.SkewStudent method), 144
pval (arch.utility.testing.WaldTestStatistic attribute), 149
pvalue (arch.unitroot.ADF attribute), 237, 240
pvalue (arch.unitroot.DFGLS attribute), 241, 243
pvalue (arch.unitroot.KPSS attribute), 254, 256
pvalue (arch.unitroot.PhillipsPerron attribute), 244, 246
pvalue (arch.unitroot.VarianceRatio attribute), 250, 253
pvalue (arch.unitroot.ZivotAndrews attribute), 247, 249
pvalues (arch.bootstrap.MCS attribute), 224
pvalues (arch.bootstrap.SPA attribute), 220

R
random_state (arch.bootstrap.CircularBlockBootstrap attribute), 191, 198
random_state (arch.bootstrap.IIDBootstrap attribute), 167, 174
random_state (arch.bootstrap.IndependentSamplesBootstrap attribute), 175, 183
random_state (arch.bootstrap.MovingBlockBootstrap attribute), 199, 206
random_state (arch.bootstrap.StationaryBootstrap attribute), 183, 191
random_state (arch.univariate.GeneralizedError attribute), 148
random_state (arch.univariate.Normal attribute), 139
random_state (arch.univariate.SkewStudent attribute), 145
random_state (arch.univariate.StudentsT attribute), 142
regression (arch.unitroot.ADF attribute), 237, 240
regression (arch.unitroot.DFGLS attribute), 241, 243
regression (arch.unitroot.ZivotAndrews attribute), 248
reset() (arch.bootstrap.CircularBlockBootstrap method), 197
reset() (arch.bootstrap.IIDBootstrap method), 172
reset() (arch.bootstrap.IndependentSamplesBootstrap method), 181
reset() (arch.bootstrap.MCS method), 223
reset() (arch.bootstrap.MovingBlockBootstrap method), 204
reset() (arch.bootstrap.SPA method), 218, 220
reset() (arch.bootstrap.StationaryBootstrap method), 189
reset() (arch.bootstrap.StepM method), 222
resid (arch.univariate.base.ARCHModelFixedResult attribute), 11

S
seed() (arch.bootstrap.CircularBlockBootstrap method), 197
seed() (arch.bootstrap.IIDBootstrap method), 173
seed() (arch.bootstrap.IndependentSamplesBootstrap method), 181
seed() (arch.bootstrap.MCS method), 223
seed() (arch.bootstrap.MovingBlockBootstrap method), 205
seed() (arch.bootstrap.SPA method), 218, 220
seed() (arch.bootstrap.StationaryBootstrap method), 189
seed() (arch.bootstrap.StepM method), 222
set_state() (arch.bootstrap.CircularBlockBootstrap method), 197
set_state() (arch.bootstrap.IndependentSamplesBootstrap method), 173
set_state() (arch.bootstrap.IIDBootstrap method), 181
set_state() (arch.bootstrap.MovingBlockBootstrap method), 205
set_state() (arch.bootstrap.StationaryBootstrap method), 189
simulate() (arch.univariate.ARCH method), 114
simulate() (arch.univariate.ARX method), 67
simulate() (arch.univariate.ConstantMean method), 60
simulate() (arch.univariate.ConstantVariance method), 85
simulate() (arch.univariate.EGARCH method), 100
simulate() (arch.univariate.EWMAVariance method), 119
simulate() (arch.univariate.FIGARCH method), 95
simulate() (arch.univariate.FixedVariance method), 128
simulate() (arch.univariate.GARCH method), 90
simulate() (arch.univariate.GeneralizedError method), 148
simulate() (arch.univariate.HARCH method), 105
simulate() (arch.univariate.HARX method), 74
simulate() (arch.univariate.LS method), 80
simulate() (arch.univariate.MIDASHyperbolic method), 110
simulate() (arch.univariate.Normal method), 138
simulate() (arch.univariate.RiskMetrics2006 method), 123
simulate() (arch.univariate.SkewStudent method), 145
simulate() (arch.univariate.StudentsT method), 141
simulate() (arch.univariate.ZeroMean method), 54
SkewStudent (class in arch.univariate), 142
SPA (class in arch.bootstrap), 218
start (arch.univariate.ARCH attribute), 115
start (arch.univariate.ConstantVariance attribute), 86
start (arch.univariate.EGARCH attribute), 101
start (arch.univariate.EWMAVariance attribute), 120
start (arch.univariate.FIGARCH attribute), 96
start (arch.univariate.FixedVariance attribute), 129
start (arch.univariate.GARCH attribute), 91
start (arch.univariate.HARCH attribute), 106
start (arch.univariate.MIDASHyperbolic attribute), 111
start (arch.univariate.RiskMetrics2006 attribute), 125
starting_values() (arch.univariate.ARCH method), 67
starting_values() (arch.univariate.ARX method), 67
starting_values() (arch.univariate.ConstantMean method), 61
starting_values() (arch.univariate.ConstantVariance method), 86
starting_values() (arch.univariate.EGARCH method), 101
starting_values() (arch.univariate.EWMAVariance method), 119
starting_values() (arch.univariate.FIGARCH method), 96
starting_values() (arch.univariate.FixedVariance method), 128
starting_values() (arch.univariate.GARCH method), 91
starting_values() (arch.univariate.GeneralizedError method), 148
starting_values() (arch.univariate.HARCH method), 105
starting_values() (arch.univariate.HARX method), 74
starting_values() (arch.univariate.LS method), 81
starting_values() (arch.univariate.MIDASHyperbolic method), 110
starting_values() (arch.univariate.Normal method), 138
starting_values() (arch.univariate.RiskMetrics2006 method), 124
starting_values() (arch.univariate.SkewStudent method), 145
starting_values() (arch.univariate.StudentsT method), 141
starting_values() (arch.univariate.ZeroMean method), 54
stat (arch.unitroot.ADF attribute), 237, 240
stat (arch.unitroot.DFGLS attribute), 241, 243
stat (arch.unitroot.KPSS attribute), 254, 256
stat (arch.unitroot.PhilipsPerron attribute), 244, 246
stat (arch.unitroot.VarianceRatio attribute), 250, 253
stat (arch.unitroot.ZivotAndrews attribute), 247, 249
stat (arch.utility.testing.WaldTestStatistic attribute), 149
StationaryBootstrap (class in arch.bootstrap), 183
StepM (class in arch.bootstrap), 220
stop (arch.univariate.ARCH attribute), 115
stop (arch.univariate.ConstantVariance attribute), 87
stop (arch.univariate.EGARCH attribute), 101
stop (arch.univariate.EWMAVariance attribute), 120
stop (arch.univariate.FIGARCH attribute), 97
stop (arch.univariate.FixedVariance attribute), 129
stop (arch.univariate.GARCH attribute), 92
stop (arch.univariate.HARCH attribute), 106
stop (arch.univariate.MIDASHyperbolic attribute), 111
stop (arch.univariate.RiskMetrics2006 attribute), 125
StudentsT (class in arch.univariate), 139
subset() (arch.bootstrap.SPA method), 220
summary (arch.unitroot.ADF attribute), 237
summary (arch.unitroot.DFGLS attribute), 241
summary (arch.unitroot.KPSS attribute), 254
summary (arch.unitroot.PhilipsPerron attribute), 244
summary (arch.unitroot.VarianceRatio attribute), 250
summary (arch.unitroot.ZivotAndrews attribute), 248
summary() (arch.unitroot.ADF method), 238
summary() (arch.unitroot.DFGLS method), 242
summary() (arch.unitroot.KPSS method), 255
summary() (arch.unitroot.PhilipsPerron method), 245
summary() (arch.unitroot.VarianceRatio method), 251
summary() (arch.unitroot.ZivotAndrews method), 248
summary() (arch.univariate.base.ARCHModelFixedResult method), 10, 13
arch Documentation, Release 4.9.1+4.g81ceedd

summary() (arch.univariate.base.ARCHModelResult method), 7, 10
superior_models (arch.bootstrap.StepM attribute), 222

T

test_type (arch.unitroot.PhilipsPerron attribute), 244, 247
trend (arch.unitroot.ADF attribute), 237, 240
trend (arch.unitroot.DFGLS attribute), 241, 243
trend (arch.unitroot.KPSS attribute), 254, 256
trend (arch.unitroot.PhilipsPerron attribute), 244, 247
trend (arch.unitroot.VarianceRatio attribute), 250, 253
trend (arch.unitroot.ZivotAndrews attribute), 248, 249
truncation (arch.univariate.FIGARCH attribute), 97

U

update_indices() (arch.bootstrap.CircularBlockBootstrap method), 197
update_indices() (arch.bootstrap.IIDBootstrap method), 173
update_indices() (arch.bootstrap.IndependentSamplesBootstrap method), 181
update_indices() (arch.bootstrap.MovingBlockBootstrap method), 205
update_indices() (arch.bootstrap.StationaryBootstrap method), 189

V

valid_trends (arch.unitroot.ADF attribute), 237, 240
valid_trends (arch.unitroot.DFGLS attribute), 241, 243
valid_trends (arch.unitroot.KPSS attribute), 254, 256
valid_trends (arch.unitroot.PhilipsPerron attribute), 244, 247
valid_trends (arch.unitroot.VarianceRatio attribute), 250, 253
valid_trends (arch.unitroot.ZivotAndrews attribute), 248, 250
values (arch.univariate.base.ARCHModelForecastSimulation attribute), 30
var() (arch.bootstrap.CircularBlockBootstrap method), 197
var() (arch.bootstrap.IIDBootstrap method), 173
var() (arch.bootstrap.IndependentSamplesBootstrap method), 181
var() (arch.bootstrap.MovingBlockBootstrap method), 205
var() (arch.bootstrap.StationaryBootstrap method), 189
variance (arch.univariate.base.ARCHModelForecast attribute), 30

WaldTestStatistic (class in arch.utility.testing), 149

X

y (arch.unitroot.ADF attribute), 237, 240
y (arch.unitroot.DFGLS attribute), 241, 244
y (arch.unitroot.KPSS attribute), 254, 256
y (arch.unitroot.PhilipsPerron attribute), 244, 247
y (arch.unitroot.VarianceRatio attribute), 250, 253
y (arch.unitroot.ZivotAndrews attribute), 248, 250

variance_bounds() (arch.univariate.ARCH method), 115
variance_bounds() (arch.univariateConstantVariance method), 86
variance_bounds() (arch.univariate.EGARCH method), 101
variance_bounds() (arch.univariate.EWMAVariance method), 120
variance_bounds() (arch.univariate.FIGARCH method), 96
variance_bounds() (arch.univariate.FixedVariance method), 129
variance_bounds() (arch.univariate.GARCH method), 91
variance_bounds() (arch.univariate.HARCH method), 105
variance_bounds() (arch.univariate.MIDASHyperbolic method), 110
variance_bounds() (arch.univariate.RiskMetrics2006 method), 124
VarianceRatio (class in arch.unitroot), 250
variances (arch.univariate.base.ARCHModelForecastSimulation attribute), 30
volatility (arch.univariate.ARX attribute), 68
volatility (arch.univariate.ConstantMean attribute), 61
volatility (arch.univariate.HARX attribute), 75
volatility (arch.univariate.LS attribute), 82
volatility (arch.univariate.ZeroMean attribute), 55
VolatilityProcess (class in arch.univariate.volatility), 129
vr (arch.unitroot.VarianceRatio attribute), 253
y (arch.univariate.ARX attribute), 68
y (arch.univariate.ConstantMean attribute), 62
y (arch.univariate.HARX attribute), 75
y (arch.univariate.LS attribute), 82
y (arch.univariate.ZeroMean attribute), 55

Z

ZeroMean (class in arch.univariate), 49
ZivotAndrews (class in arch.unitroot), 247