PyDSE Documentation

Release 0.2.post.dev1

Blue Yonder

October 27, 2015

Toolset for Dynamic System Estimation for time series inspired by DSE. It is in a beta state and only includes ARMA models right now.

CHAPTER 1

Contents

1.1 Create and Simulate

The definition of an ARMA model is:

$$A(L)y_t = B(L)e_t + C(L)u_t$$

where L is the *lag* operator, y_t a p-dimensional vector of observed output variables, e_t a p-dimensional vector of white noise and u_t a m-dimensional vector of input variables. Since A, B and C are matrices in the lag shift operator, we have $A(L)$ is a $a \times\ p \times\ p$ tensor to define auto-regression, $B(L)$ is a $b \times\ p \times\ p$ tensor to moving-average and $C(L)$ is a $c \times\ p \times\ m$ tensor to account for the input variables.

We create a simple ARMA model for a two dimensional output vector with matrices:

$$A(L) = \left(\begin{array}{cc} 1 + 0.5L_1 + 0.3L_2 & 0 + 0.2L_1 + 0.1L_2 \\ 0 + 0.2L_1 + 0.05L_2 & 1 + 0.5L_1 + 0.3L_2 \end{array} \right),$$

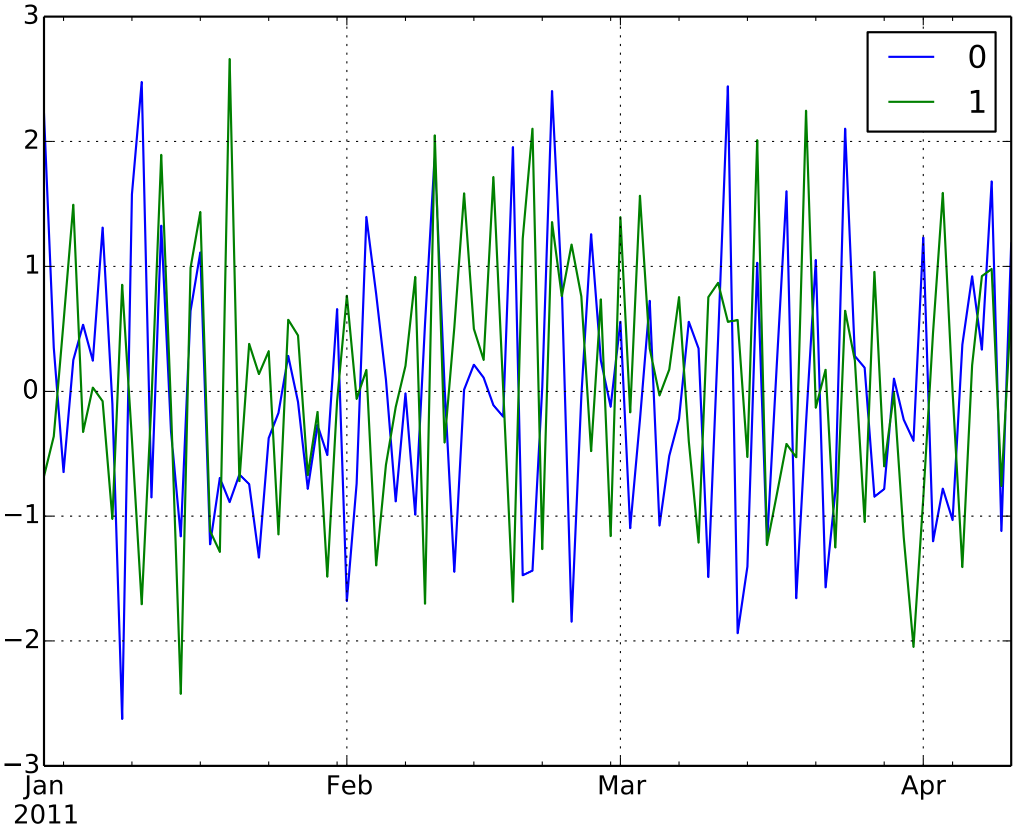
$$B(L) = \left(\begin{array}{cc} 1 + 0.2L_1 & 0 + 0.1L_1 \\ 0 + 0.0L_1 & 1 + 0.3L_1 \end{array} \right)$$

In order to set this matrix we just write the entries left to right, up to down into an array and define the shape of this array in a second array:

```
import pandas as pd
import numpy as np
import matplotlib.pylab as plt
from pydse.arma import ARMA

AR = (np.array([1, .5, .3, 0, .2, .1, 0, .2, .05, 1, .5, .3]), np.array([3, 2, 2]))
MA = (np.array([1, .2, 0, .1, 0, 0, 1, .3]), np.array([2, 2, 2]))
arma = ARMA(A=AR, B=MA, rand_state=0)
```

Note that we set the random state to seed 0 to get the same results. Then by simulating we get:

```
sim_data = arma.simulate(sampleT=100)
sim_index = pd.date_range('1/1/2011', periods=sim_data.shape[0], freq='d')
df = pd.DataFrame(data=sim_data, index=sim_index)
df.plot()
```

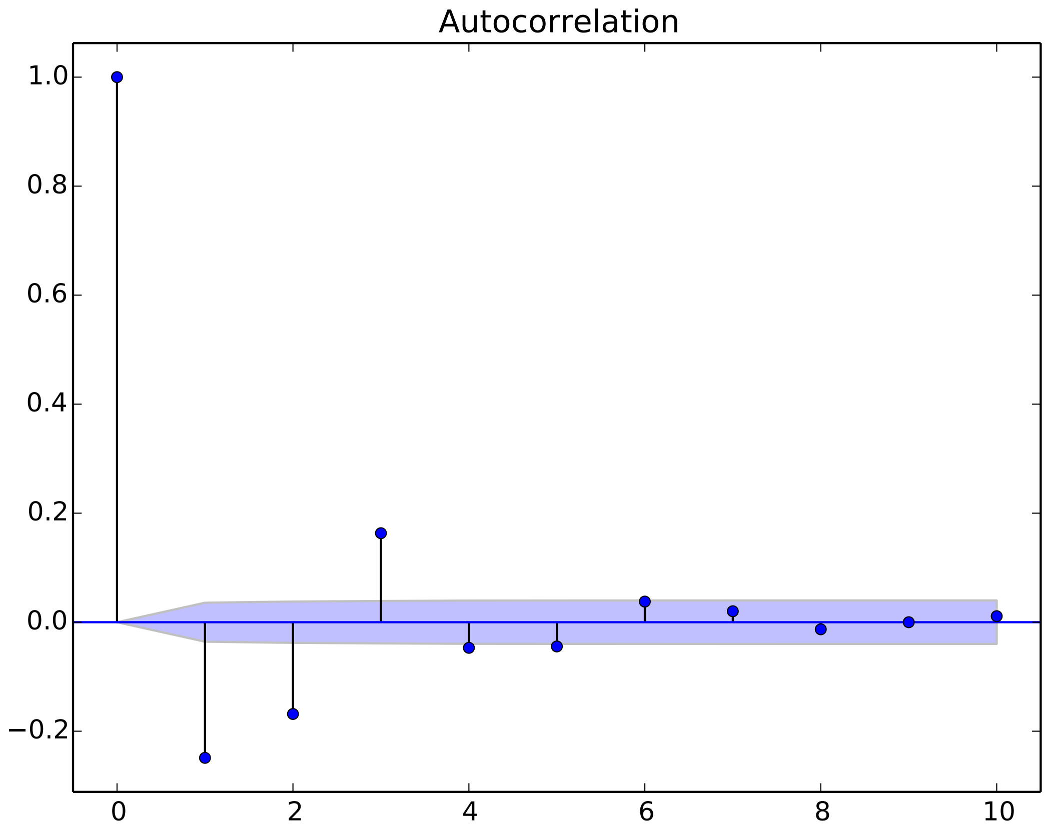


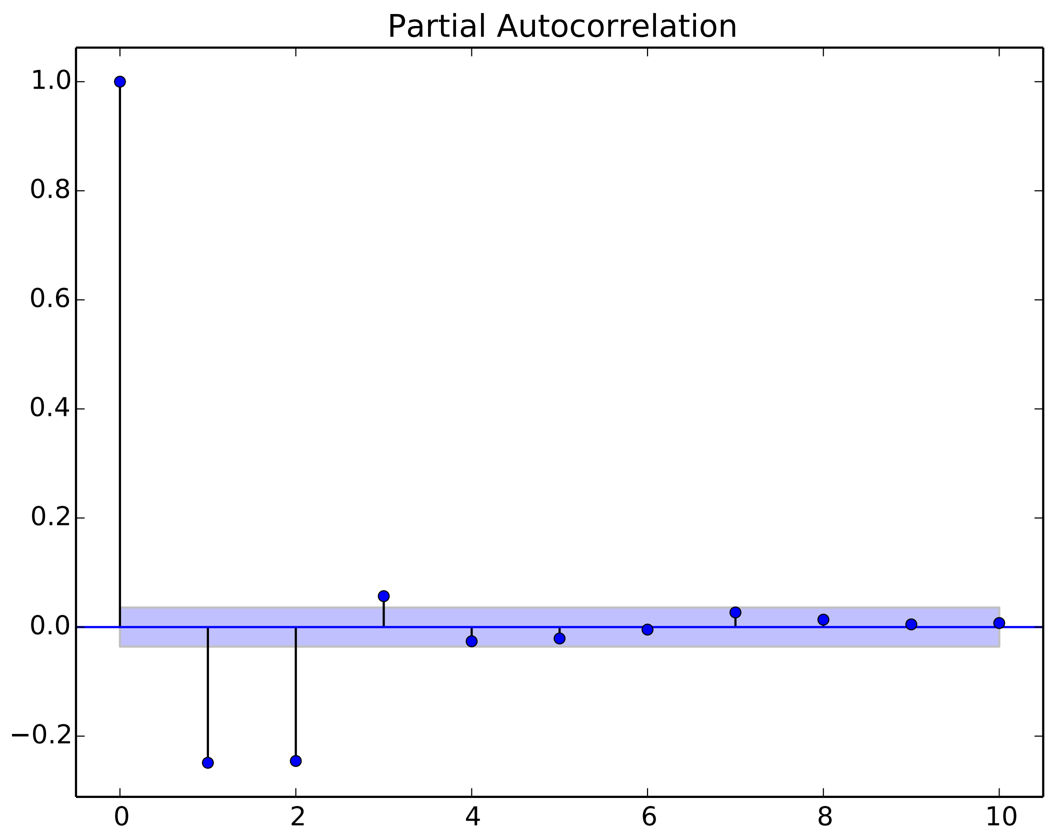
Let's create a simpler ARMA model with scalar output variable.

```
AR = (np.array([1, .5, .3]), np.array([3, 1, 1]))
MA = (np.array([1, .2]), np.array([2, 1, 1]))
arma = ARMA(A=AR, B=MA, rand_state=0)
```

Quite often you wanna check the autocorrelation function and partial autocorrelation function:

```
from statsmodels.graphics.tsaplots import plot_pacf, plot_acf

sim_data = arma.simulate(sampleT=3000)
sim_index = pd.date_range('1/1/2011', periods=sim_data.shape[0], freq='d')
df = pd.DataFrame(data=sim_data, index=sim_index)
plot_acf(df[0], lags=10)
plot_pacf(df[0], lags=10)
plt.show()
```


Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
2
4
6
8
10

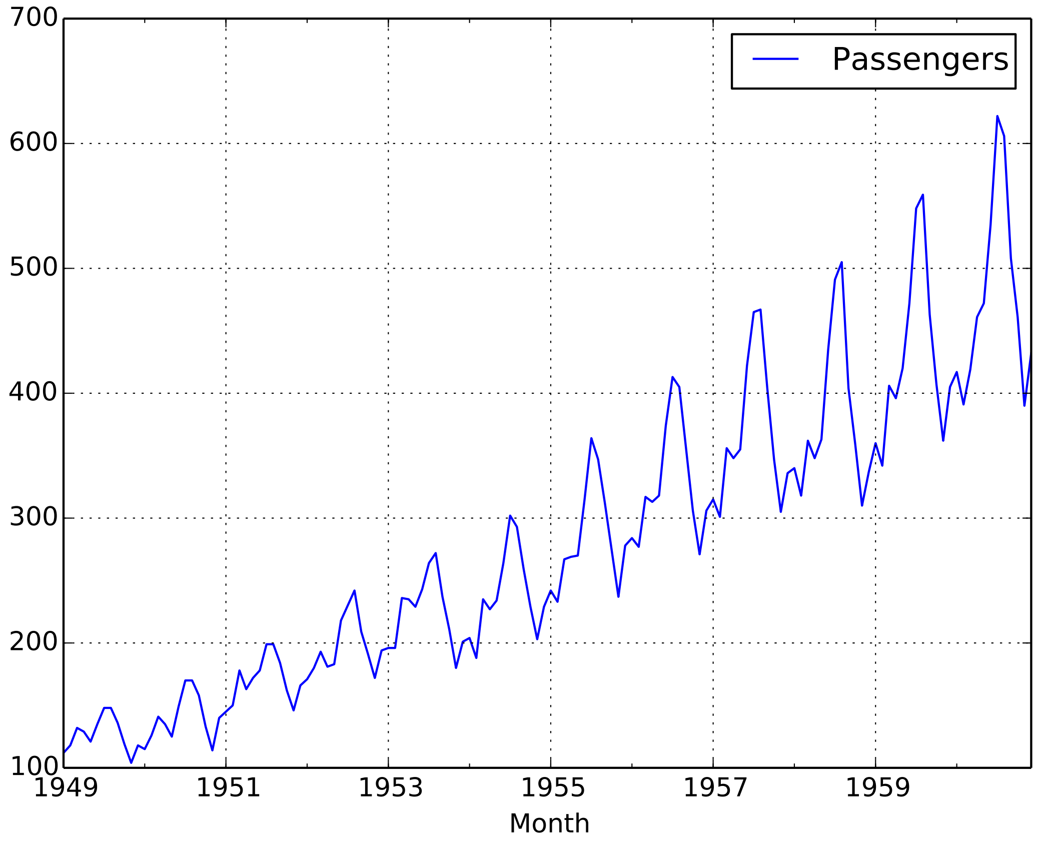


Find a good introduction to ARMA on the Decision 411 course page.

1.2 Estimation of Parameters

In this section we estimate the parameters for the time series of the monthly passengers of an international airline.

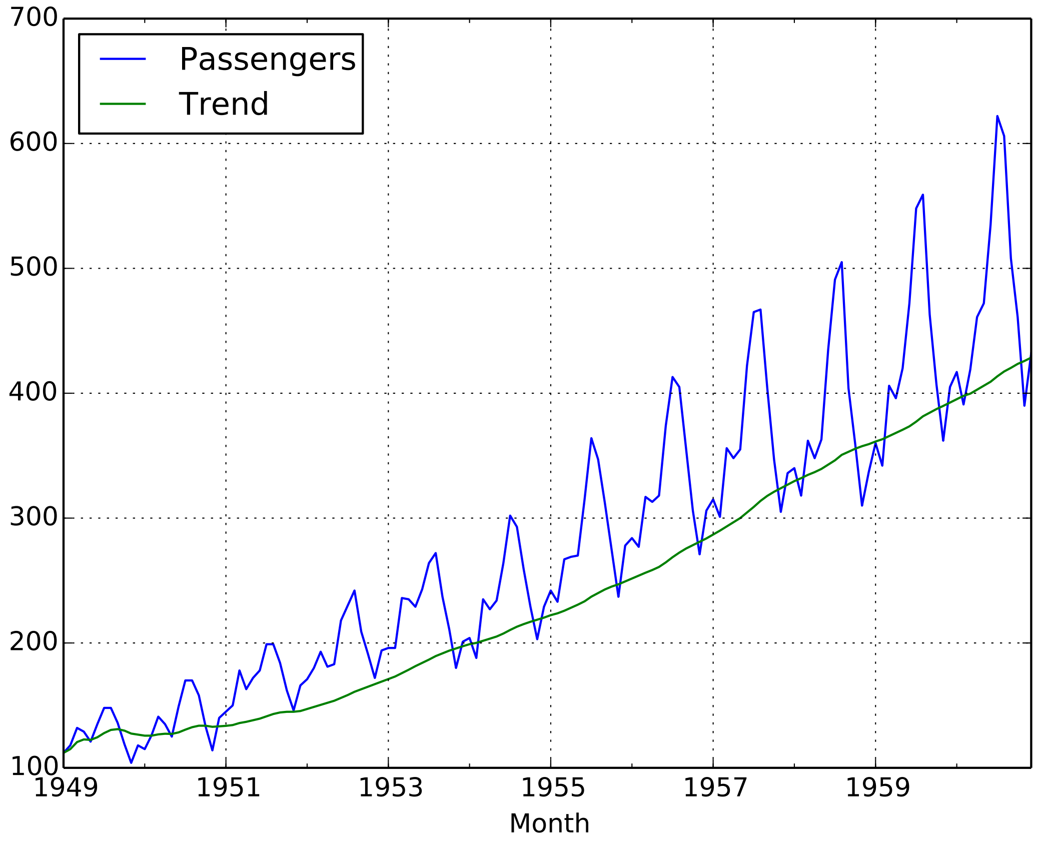
```
import numpy as np
from pydse import data

df = data.airline_passengers()
df.plot()
```



```
plt.close()
np.random.seed(0)
```

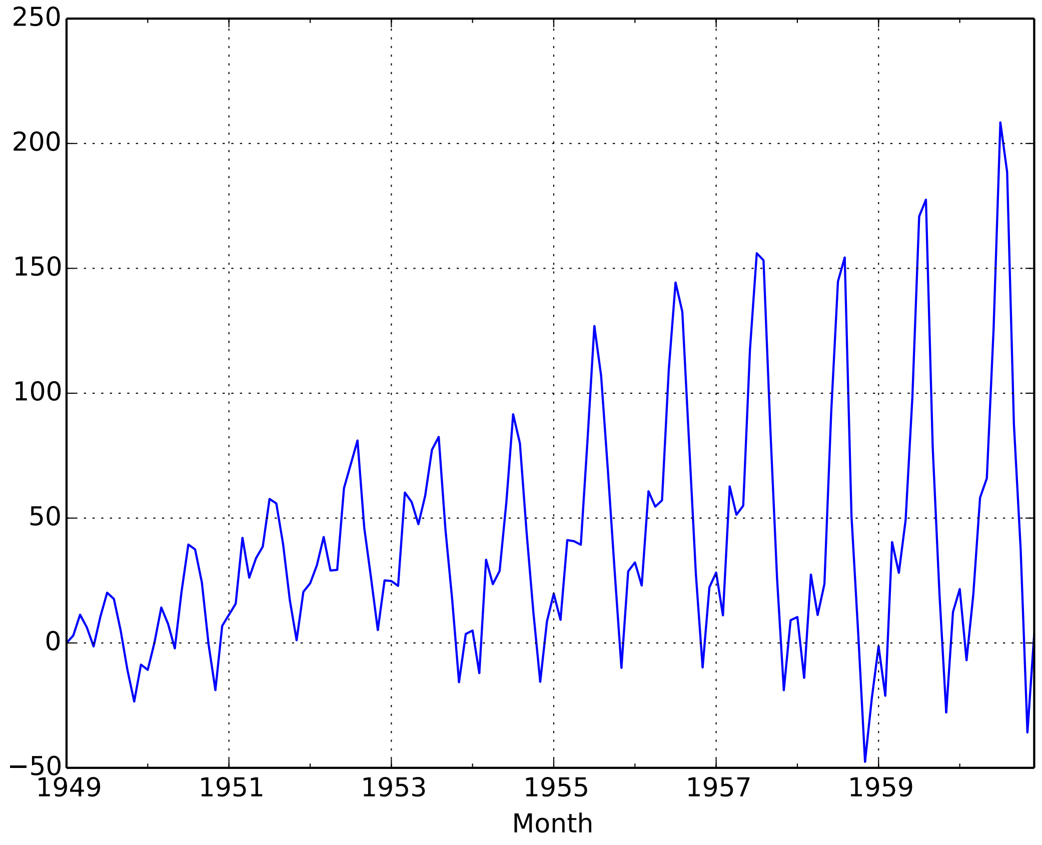
Obviously, there is a strong trend in the data. Since ARMA can handle only stationary time series we have to remove it. In order to do that, we would like to smooth the time series. We see that there is 12 month seasonality, and therefore taking 3 years as a window for a smoothing function should be alright. An option would be a rolling mean:

```
from pandas.stats.moments import rolling_mean
df['Trend'] = rolling_mean(df['Passengers'], window=36, min_periods=1)
df.plot()
```



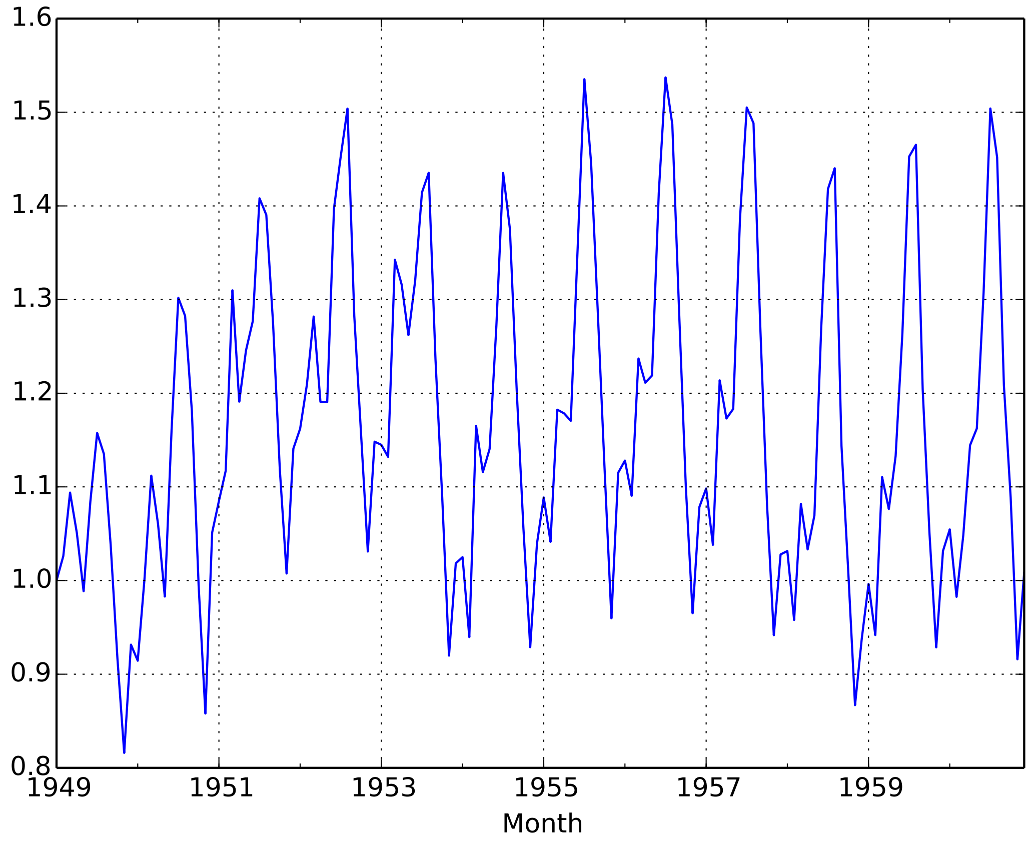
Our first guess is now to remove the trend by subtracting the *Trend* from our time series:

```
residual = df['Passengers'] - df['Trend']
residual.plot()
```



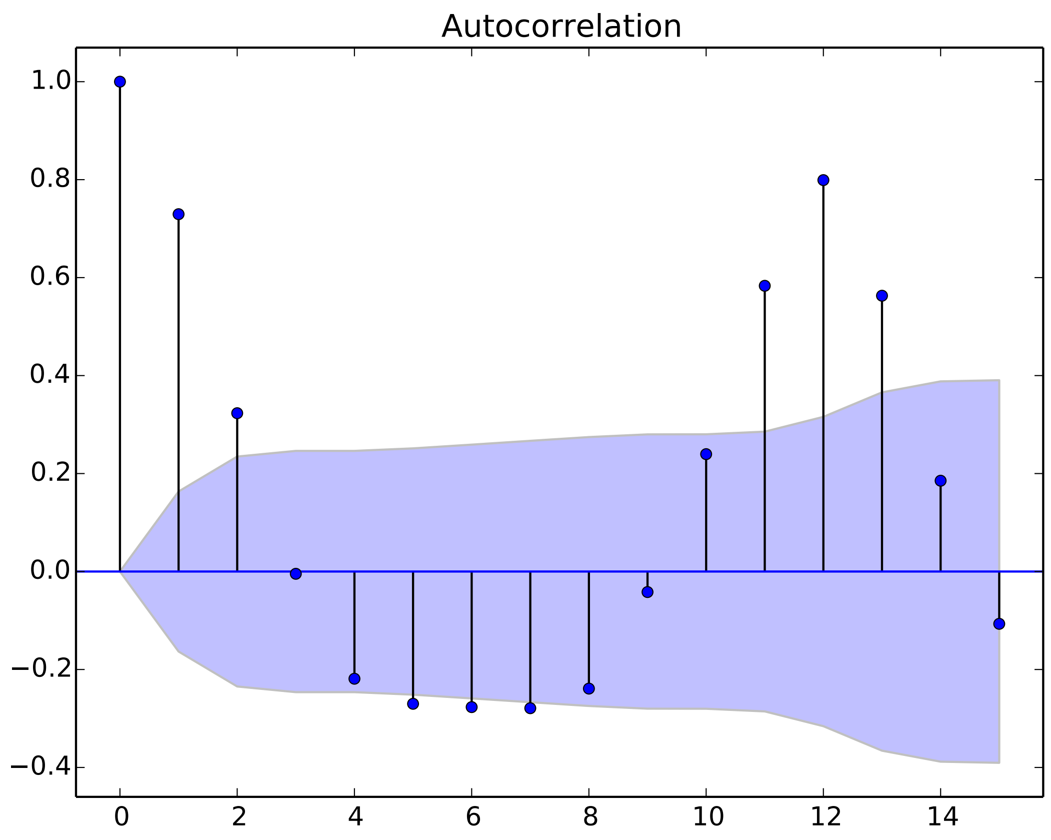
Obviously the trend is removed but the variance does not seem to be stationary, i.e. there is heteroscedasticity. Since the variance seems to be related with the absolut value of the time series we use another ansatz:

```
residual = df['Passengers'] / df['Trend']
residual.plot()
```

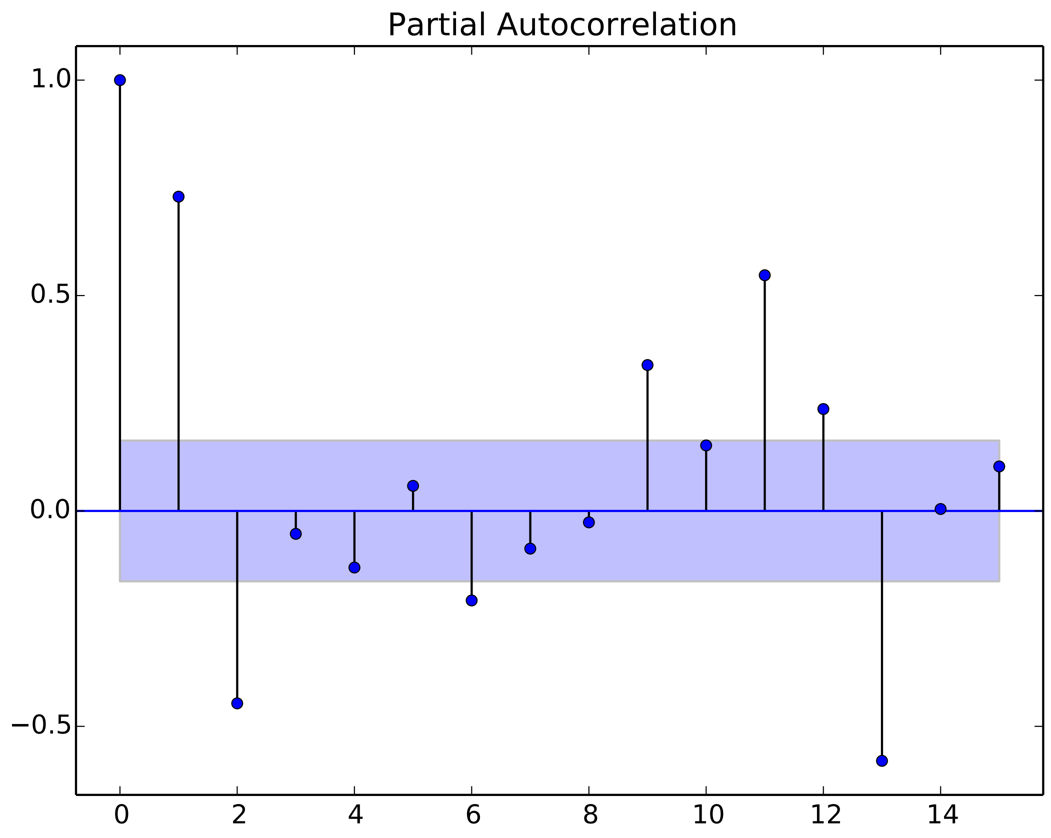


This time the series looks like a stationary process. Again, we look at the ACF and PACF plots.

```
from statsmodels.graphics.tsaplots import plot_pacf, plot_acf
plot_acf(residual, lags=15)
```



```
plot_pacf(residual, lags=15)
```



These plots show us the strong seasonality of 12 months. Due to this plots, we want to estimate an ARMA model where the *AR* term has only lag of 12 and the *MA* has lags 1 and 13. All other lags (except of 0 of course) should be equal to zero.

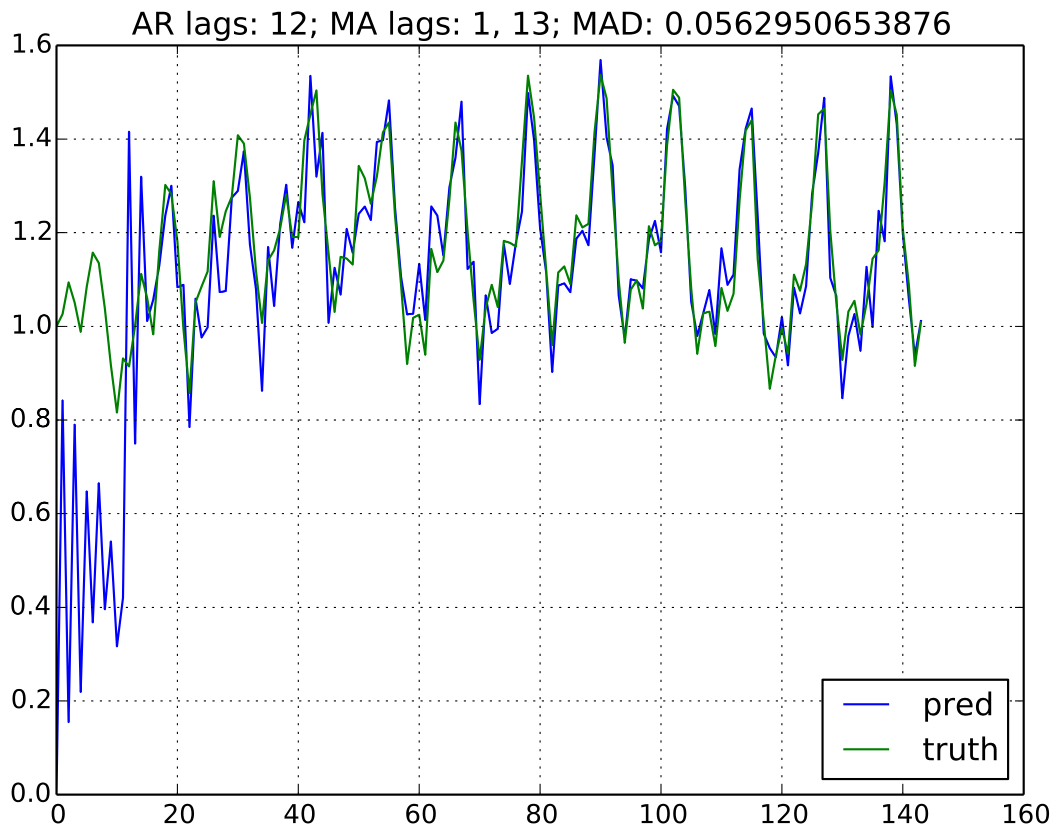
```
from pydse.arma import ARMA

AR = (np.array([1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.01]),
      np.array([13, 1, 1]))
MA = (np.array([1, 0.01, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.01]),
      np.array([14, 1, 1]))
arma = ARMA(A=AR, B=MA, rand_state=0)
arma.fix_constants()
```

The `fix_constants` function determines the constants of our model. Every parameter that has less or equal than one decimal place is considered constant. Now the only remaining parameters are the ones that we set to *0.01*. In order to estimate those we call `est_params` with our residual time series:

```
arma.est_params(residual)
```

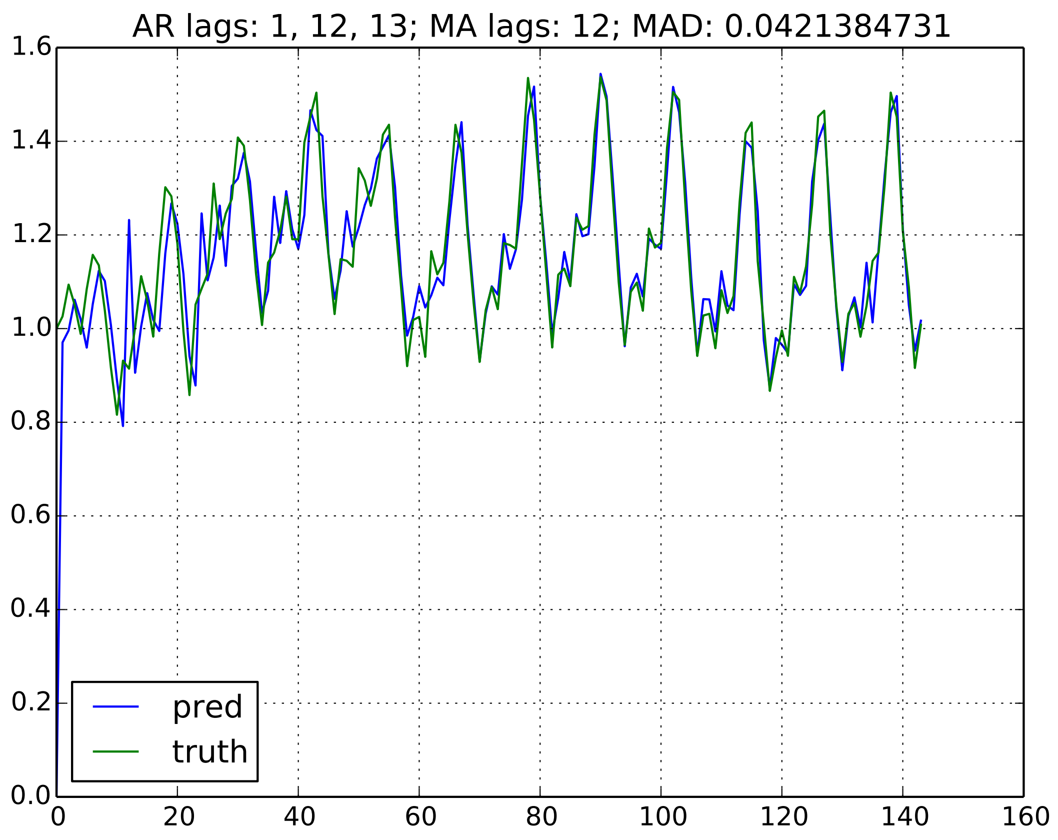
The output of this command tells us if our opimization method converged. We can now take a look if our estimated ARMA process produces a similar time series than residual. To quantify this similarity, we should take a look at the Mean Absolute Deviation (MAD) where we are in this case only interested in predictions starting from month 20 since it takes a while for ARMA to adjust .

```
import pandas as pd
result = pd.DataFrame({'pred': arma.forecast(residual)[:, 0],
                            'truth': residual.values})
MAD = np.mean(np.abs(result['pred'][20:] - result['truth'][20:]))
result.plot(title="AR lags: 12; MA lags: 1, 13; MAD: {}".format(MAD))
```



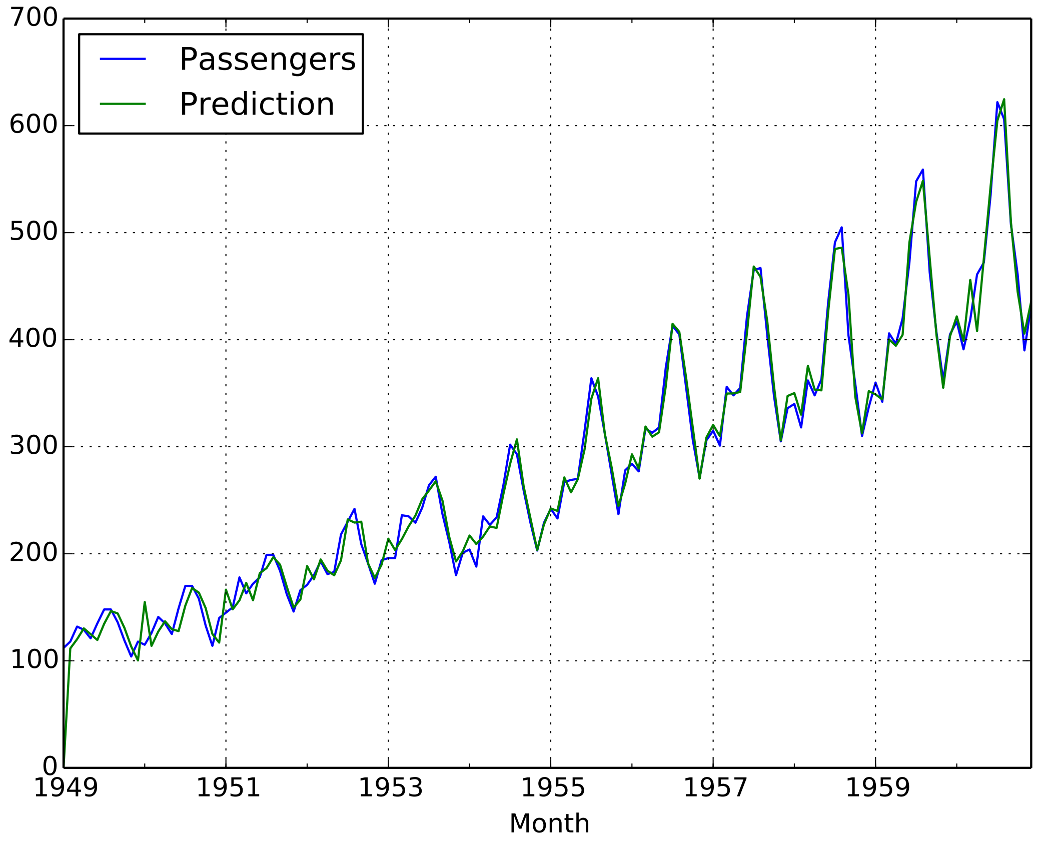
Instead of guessing the possible parameters by looking at the ACF and PACF plots, we can also use the *minic* function. This function takes a set of possible AR and MA lags to consider, calculates for each combination some information criterion and chooses the most likely. Let's say we are quite unsure how to interpret ACF and PACF plots and we just use our gut feeling that lag 1 and maybe lag 11, 12 as well as 13 could be useful as AR and MA lags. We just provide those guesses to *minic* and get the best AR and MA lags. Then, we apply the *make_lag_arr* function to generate one dimensional lag matrices that we use as inputs for our ARMA model as before. There we go:

```
from pydse.arma import minic
from pydse.utils import make_lag_arr

best_ar_lags, best_ma_lags = minic([1, 11, 12, 13], [1, 11, 12, 13], residual)
arma = ARMA(A=make_lag_arr(best_ar_lags),
            B=make_lag_arr(best_ma_lags),
            rand_state=0)
arma.fix_constants()
arma.est_params(residual)
result = pd.DataFrame({'pred': arma.forecast(residual)[:, 0],
                       'truth': residual.values})
MAD = np.mean(np.abs(result['pred'][20:] - result['truth'][20:]))
result.plot(title="AR lags: {}; MA lags: {}; MAD: {}".format(
    ", ".join(map(str, best_ar_lags)), ", ".join(map(str, best_ma_lags)), MAD))
```



Finally, we will apply the necessary back transformation to our time series:

```
df['Prediction'] = result['pred'].values * df['Trend'].values
del df['Trend']
df.plot()
```



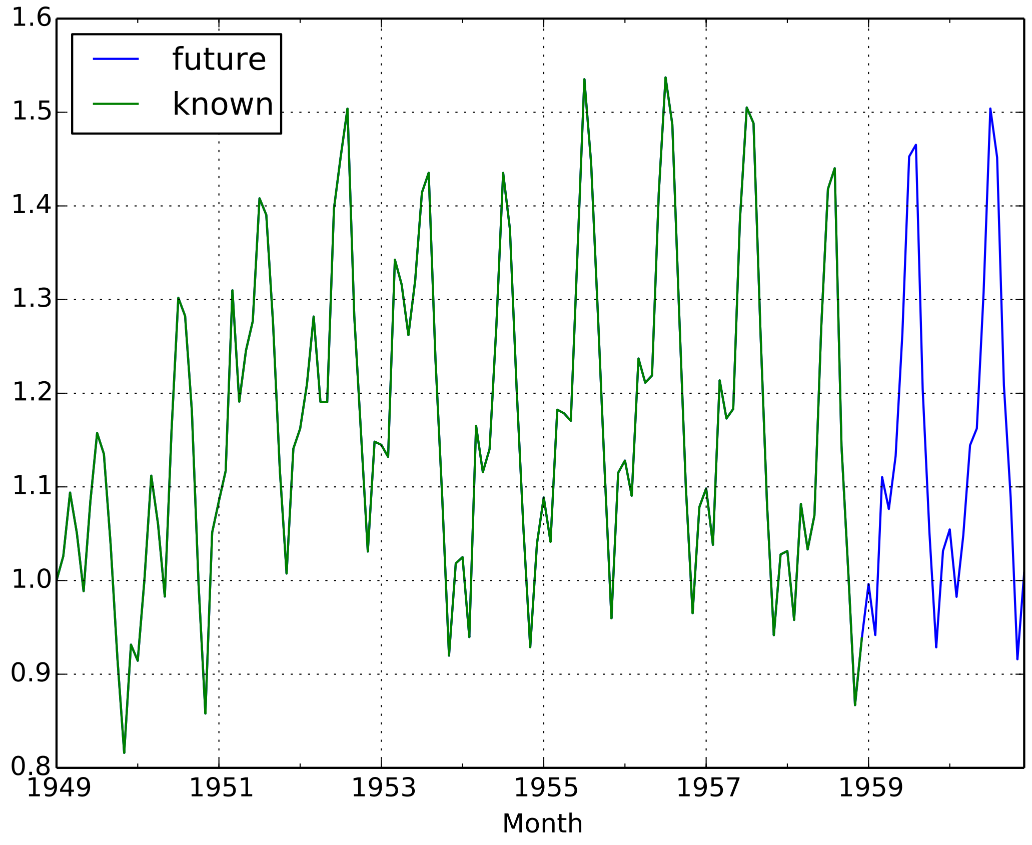
1.3 Forecast with Horizon

In chapter *Estimation of Parameters* only one-step ahead predictions were applied. So in our example of monthly data, all data up to the last month was used to predict the number of passengers in the current month. This is possible up to the last date with the known truth.

In order to do forecasts beyond the period where all data is available, we can provide a `horizon` to the *`forecast`* function. The horizon specifies the number of one-step ahead predictions that should be done after the last known data point.

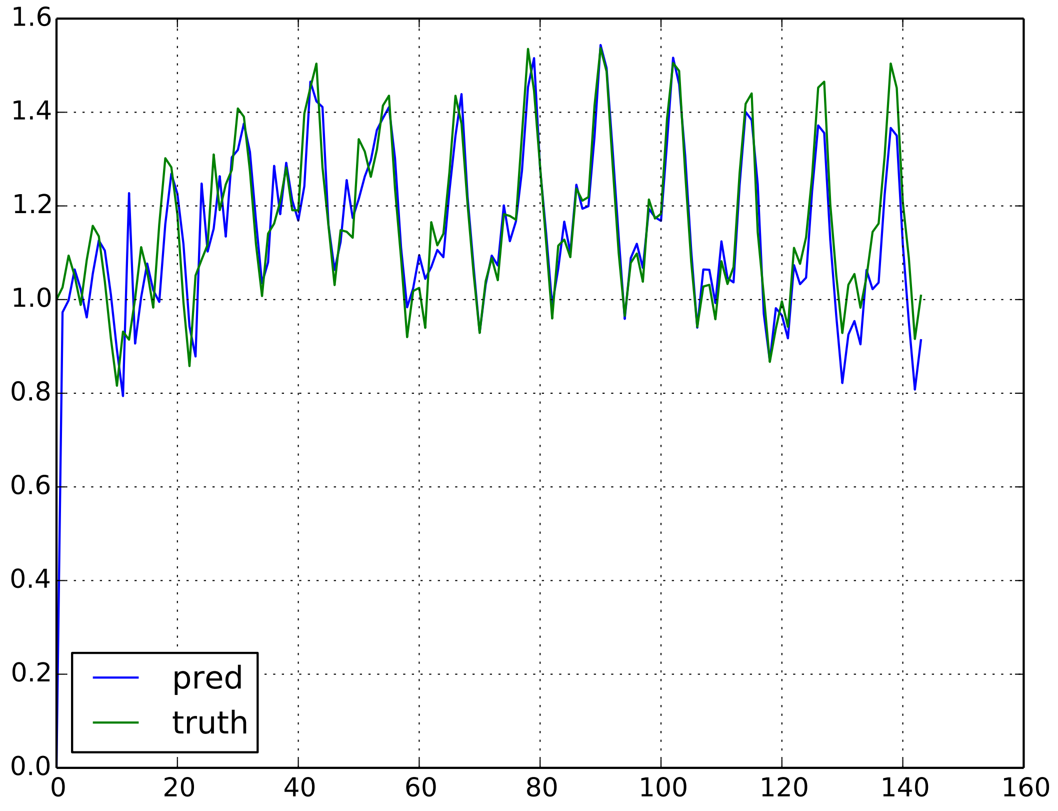
To illustrate this, we take the same example as in the last chapter and remove the last two years from the data:

```
from pydse import data
from pydse.arma import ARMA
from pydse.utils import make_lag_arr
import pandas as pd
from pandas.stats.moments import rolling_mean

df = data.airline_passengers()
df['Trend'] = rolling_mean(df['Passengers'], window=36, min_periods=1)
residual_all = df['Passengers'] / df['Trend']
residual_known = residual_all[:-24]
pd.DataFrame({'future': residual_all, 'known': residual_known}).plot()
```



Now, we fit an ARMA model with the *known* data and forecast the following two years after the known time period:

```
arma = ARMA(A=make_lag_arr([1, 12, 13]), B=make_lag_arr([12]), rand_state=0)
arma.fix_constants()
arma.est_params(residual_known)
pred = arma.forecast(residual_known, horizon=24)
result = pd.DataFrame({'pred': pred[:, 0], 'truth': residual_all.values})
result.plot()
```



By eye, it can be seen that our predictions are still quite accurate but not as good as using one-step ahead predictions with data up to the previous month.

1.4 Contribution

PyDSE is developed by currently one developer in his spare time out of pure interest in ARMA models. You are very welcome to join in this effort if you would like to contribute.

1.4.1 Bug Reports

If you experience bugs or in general issues with PyDSE, please file a bug report to our Bug Tracker.

1.4.2 Code

If you would like to contribute to PyDSE, fork the main repository on GitHub, then submit a "pull request" (PR):

1. Create an account on GitHub if you do not already have one.
2. Fork the project repository: click on the *Fork* button near the top of the page. This creates a copy of the code under your account on the GitHub server.
3. Clone this copy to your local disk:

```
git clone git@github.com:YourLogin/pydse.git
```

4. Create a branch to hold your changes:

```
git checkout -b my-feature
```

   and start making changes. Never work in the master branch!

5. Work on this copy, on your computer, using Git to do the version control. When you're done editing, do:

```
git add modified_files
git commit
```

   to record your changes in Git, then push them to GitHub with:

```
git push -u origin my-feature
```

6. Go to the web page of your PyDSE fork, and click "Create pull request" to send your changes to the maintainers for review. Find more detailed information here.

1.5 License

```
Licence summary
You may copy and redistribute the data. You may make derivative works from the
data. You may use the data for commercial purposes. You may not sublicence the
data when redistributing it. You may not redistribute the data under a
different license. Source attribution on any use of this data: Must refer
source.
```

1.6 Release Notes

1.6.1 Version 0.2.1, 2015-10-27

- Fix unit tests due to changes in Pandas

1.6.2 Version 0.2, 2014-12-23

- Update to PyScaffold 1.3.1
- Minor fixes in docs
- Added exercises and presentation
- Added plot_forecast function

1.6.3 Version 0.1, 2014-07-11

- First release

1.7 pydse

1.7.1 pydse package

Subpackages

pydse.data package

Module contents Example data files taken from DataMarket. Data is under the default open license:

`pydse.data.`**`airline_passengers`**`()`
Monthly totals of international airline passengers in thousands, Jan 1949 - Dec 1960

`pydse.data.`**`cpi_canada`**`()`
Monthly CPI, Canada, Jan 1950 - Dec 1973

`pydse.data.`**`m1_us`**`()`
Monthly M1 U.S., Jan 1959 - Feb 1992

`pydse.data.`**`sales_cola`**`()`
Monthly sales of Tasty Cola, Jan 2001 - Mar 2012

`pydse.data.`**`sales_petroleum`**`()`
Monthly sales of petroleum and related products in the U.S., Jan 1971 - Dec 1991

`pydse.data.`**`sales_product`**`()`
: Monthly sales of a plastic manufacturer's product, Jan 2001 - May 2012

`pydse.data.`**`sales_shampoo`**`()`
: Monthly sales of shampoo, Jan 2001 - Mar 2012

Submodules

pydse.arma module

All functionality related to ARMA models.

class `pydse.arma.`**`ARMA`**`(`*A*, *B=None*, *C=None*, *TREND=None*, *rand_state=None*`)`

Bases: *`pydse.utils.UnicodeMixin`*

A(L)y(t) = B(L)e(t) + C(L)u(t) - TREND(t)

- L: Lag/Shift operator,
- A: (axpxp) tensor to define auto-regression,
- B: (bxpxp) tensor to define moving-average,
- C: (cxpxm) tensor for external input,
- e: (txp) matrix of unobserved disturbance (white noise),
- y: (txp) matrix of observed output variables,
- u: (mxt) matrix of input variables,
- TREND: (txp) matrix like y or a p-dim vector.

If B is net set, fall back to VAR, i.e. B(L) = I.

`est_params``(`*y*`)`

Maximum likelihood estimation of the ARMA model's coefficients.

Parameters **`y`** – output series

Returns optimization result (`OptimizeResult`)

`fix_constants``(`*fuzz=1e-05*, *prec=1*`)`

Fix some coefficients as constants depending on their value.

Coefficient with a absolute difference of `fuzz` to a value of precision `prec` are considered constants.

For example:

- 1.1 is constant since abs(1.1 - round(1.1, prec)) < fuzz
- 0.01 is non constant since abs(0.01 - round(0.01, prec)) > fuzz

`forecast``(`*y*, *horizon=0*, *u=None*`)`

Calculate an one-step-ahead forecast.

Parameters

- **`y`** – output time series
- **`horizon`** – number of predictions after y[T_max]
- **`u`** – external input time series

Returns predicted time series as array

non_consts
Parameters of the ARMA model that are non-constant.

Returns array

plot_forecast (*all_y*, *horizon=0*, *u=None*)
Calculate an one-step-ahead forecast and plot prediction and truth.

Parameters

- **y** – output time series
- **horizon** – number of predictions after y[T_max]
- **u** – external input time series

Returns predicted time series as array

simulate (*y0=None*, *u0=None*, *u=None*, *sampleT=100*, *noise=None*)
Simulate an ARMA model.

Parameters

- **y0** – lagged values of y prior to t=0 in reversed order
- **u0** – lagged values of u prior to t=0 in reversed order
- **u** – external input time series
- **sampleT** – length of the sample to simulate
- **noise** – tuple (w0, w) of a random noise time series. w0 are the lagged values of w prior to t=0 in reversed order. By default a normal distribution for the white noise is assumed.

Returns simulated time series as array

exception pydse.arma.**ARMAError**
Bases: exceptions.Exception, *pydse.utils.UnicodeMixin*

pydse.arma.**minic** (*ar_lags*, *ma_lags*, *y*, *crit=u'BIC'*)
Minimum information criterion method to fit ARMA.

Use the Akaike information criterion (AIC) or Bayesian information criterion (BIC) to determine the most promising AR and MA lags for an ARMA model.

This method only works for scalar time series, i.e. dim(y[0]) = 1.

Parameters

- **ar_lags** – list of AR lags to consider
- **ma_lags** – list of MA lags to consider
- **y** – target vector or scalar time series
- **crit** – information criterion (‘BIC’ or ‘AIC’)

Returns tuple of AR lags and MA lags

pydse.stats module

General statistical functions and related tools.

pydse.stats.**aic** (*L*, *k*)
Akaike information criterion.

Parameters

- **L** – maximized value of the negative log likelihood function
- **k** – number of free parameters

Returns AIC

`pydse.stats.`**`bic`**(*L*, *k*, *n*)

Bayesian information criterion.

Parameters

- **L** – maximized value of the negative log likelihood function
- **k** – number of free parameters
- **n** – number of data points

Returns BIC

`pydse.stats.`**`negloglike`**(*pred*, *y*)

Negative log-likelihood of the residual of two time series.

Parameters

- **pred** – predicted time series
- **y** – target time series

Returns scalar negative log-likelihood

pydse.utils module

Various different utilities and tools.

class `pydse.utils.`**`UnicodeMixin`**

Bases: `object`

Mixin class to handle defining the proper __str__/__unicode__ methods in Python 2 or 3.

`pydse.utils.`**`atleast_2d`**(*arr*)

Ensure that an array has at least dimension 2.

Parameters **arr** – array

Returns array with dimension of at least 2

`pydse.utils.`**`make_lag_arr`**(*lags*, *fuzz=0.01*)

Create a lag polynomial that can be used as 1-dim A and B for ARMA.

This function creates a lag polynomial, i.e. 1-dim lag matrix, and sets to parameters that are to be estimated to the `fuzz` value. Check *`fix_constants`* for further information.

Parameters

- **lags** – list of lags
- **fuzz** – fill value to mark non-constant lags

Returns tuple of the lag array and its shape

`pydse.utils.`**`powerset`**(*iterable*)

Calculates the power set of an iterable.

Parameters **iterable** – iterable set

Returns iterable power set

Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index