monte_carlo_simulation

May 12, 2020

[1]:
```
%matplotlib inline

## Load packages
import os
import pickle
import datetime
import numpy as np
import pandas as pd
from numba import jit

from scipy import stats
from statsmodels.tsa import stattools
from statsmodels.tsa.stattools import acf, pacf, ARMA
from statsmodels.stats.stattools import durbin_watson
from statsmodels.tsa.seasonal import STL
from statsmodels.tsa.ar_model import AutoReg
import statsmodels.api as sm

from dateutil.relativedelta import relativedelta

import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import pylab
import seaborn as sns
```

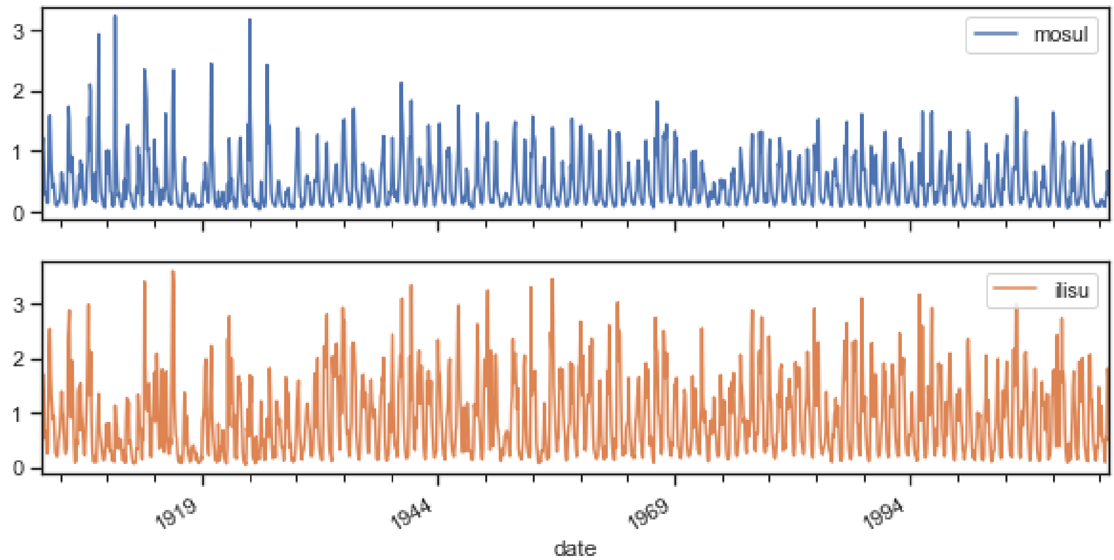
1 CCIWR class project

This code implements monte carlo simulation and prediction based on autoregressive processes.

[2]:
```
# Some plotting defaults
sns.set(style="ticks")
plt.rcParams.update({'font.size': 18})
plt.rcParams.update({'mathtext.default':'regular'})
plt.rcParams["figure.figsize"] = [10, 5]
```

[3]:
```
# Load the data
df = pd.read_csv("grun_data.csv", index_col='date', usecols=['date', 'mosul',␣
↪'ilisu'], parse_dates=True)
df.plot(subplots=True)
```

[3]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x1268394d0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x129886f90>],
      dtype=object)
```

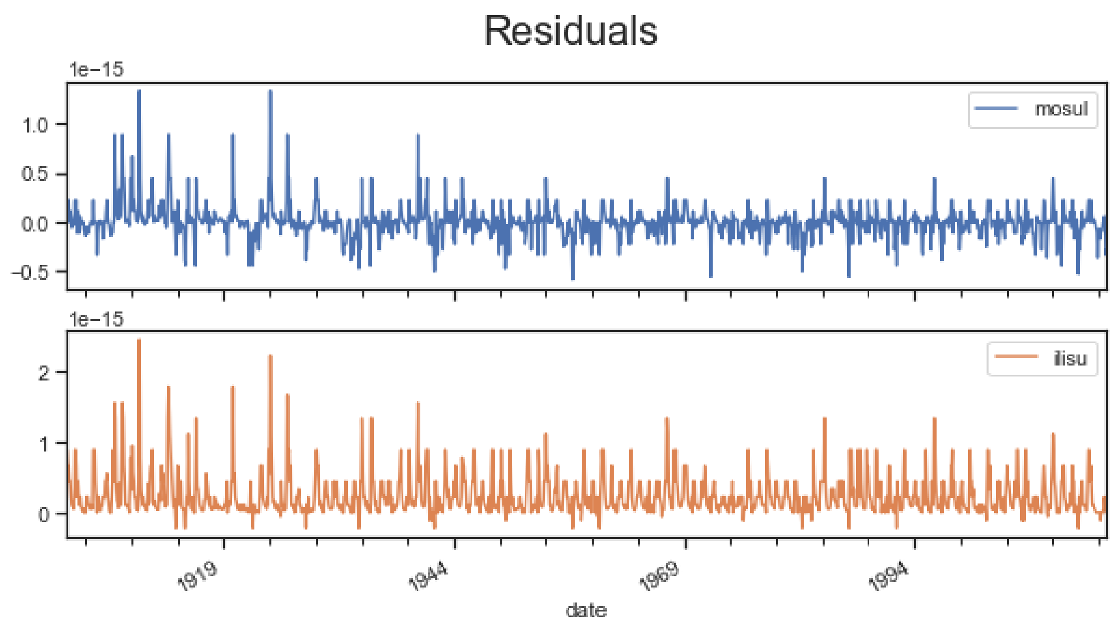


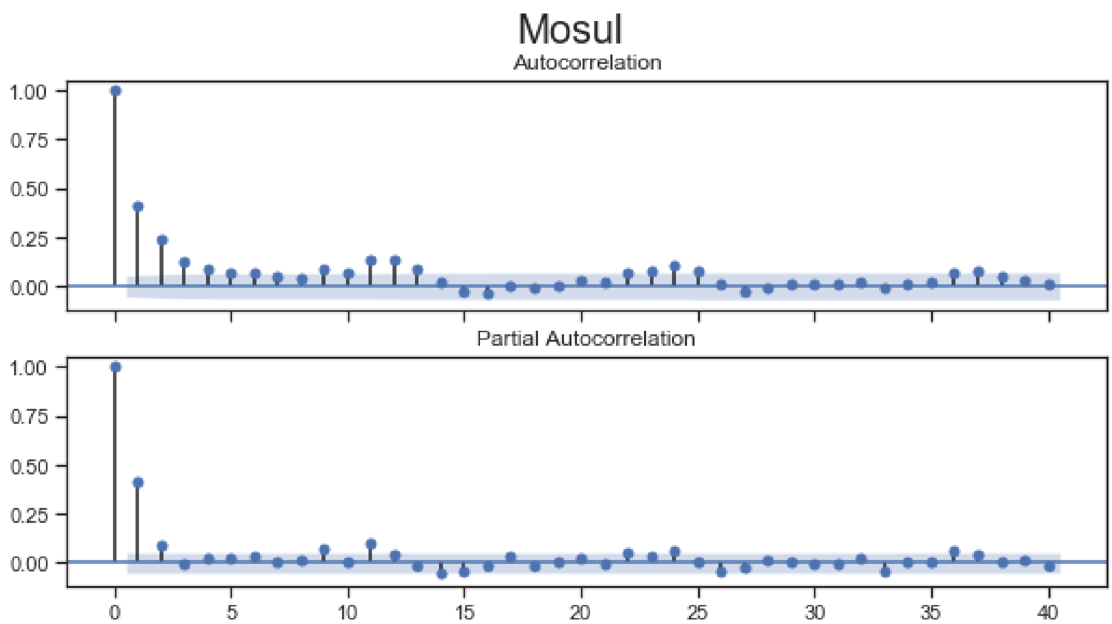
Find order of auto-regressive process (monthly sampling): AR(1) seems likely

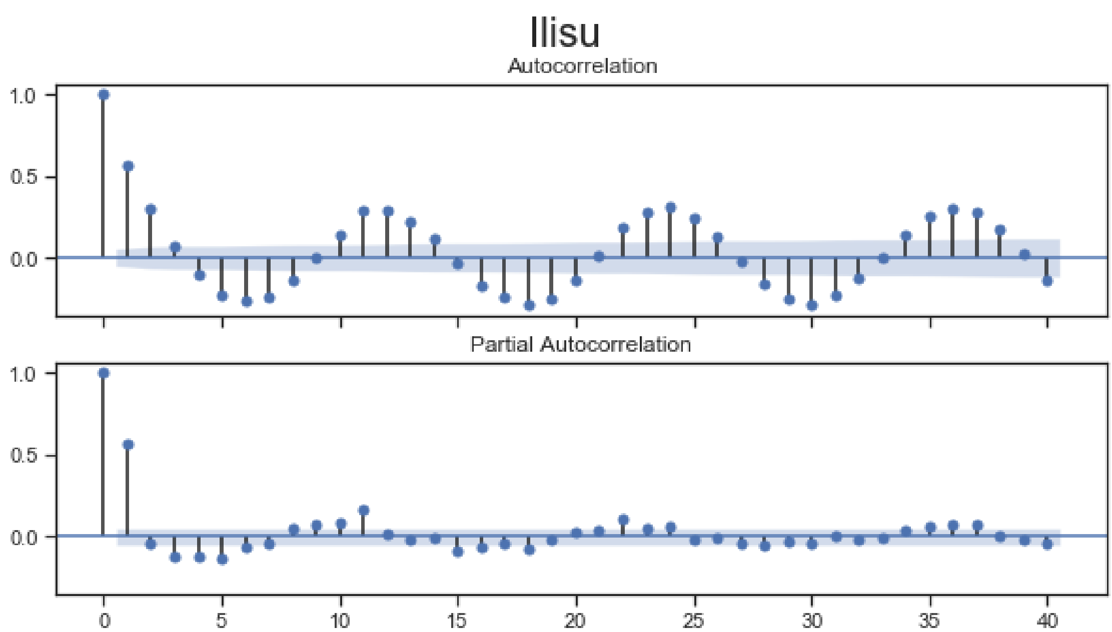
[4]:
```
def detrend(s):
    y = s.values
    x = list(range(1, len(s) + 1))
    x = np.reshape(x, (-1, 1))
    x = sm.add_constant(x)  # to add intercept"
    trd = sm.OLS(y, x).fit()  # fit regression"
    trend = trd.predict(x)  # compute trend line"
    return pd.Series(data=(y-trend), index=s.index)

# detrend
df_residuals = df.apply(detrend, axis=1)
df_residuals.plot(subplots=True, title="Residuals")

# acf and pacf
fig, axes = plt.subplots(nrows=2, ncols=1, sharex=True, sharey=True)
x = sm.graphics.tsa.plot_acf(df_residuals.mosul.values.squeeze(), lags=40,␣
↪ax=axes[0])
```

```
y = sm.graphics.tsa.plot_pacf(df_residuals.mosul.values.squeeze(), lags=40,␣
↪ax=axes[1])
plt.suptitle("Mosul")


fig, axes = plt.subplots(nrows=2, ncols=1, sharex=True, sharey=True)
x = sm.graphics.tsa.plot_acf(df_residuals.ilisu.values.squeeze(), lags=40,␣
↪ax=axes[0])
y = sm.graphics.tsa.plot_pacf(df_residuals.ilisu.values.squeeze(), lags=40,␣
↪ax=axes[1])
plt.suptitle("Ilisu")
```

[4]: Text(0.5, 0.98, 'Ilisu')







An auto-regressive model of order 1 ("AR(1)" process) is described by

$X_t = c + \phi * X_{t-1} + \epsilon_t$.

It specifies that the output variable depends linearly on its own previous values lagged by t and on a stochastic term ϵ_t in the form of a stochastic difference equation. In the regression analysis above we made the iid assumptions. Since the data seem to follow an AR(1) process, the assumption of independence between residuals ϵ_t is flawed. This leads to a biased trend estimate.

We need these 3 formulas:

1) $x_t = \beta_0 + \beta_1 * t + \epsilon_t$
2) $\epsilon_t = \epsilon_{t-1} * \alpha + w_t$
3) Combined: $x_t = \beta_0 + \beta_1 * t + x_{t-1} * \alpha + w_t$

[5]:
```
class TimeSeriesModel(object):
    def __init__(self, ts):
        assert type(ts) is pd.Series, "Class expects a pd.Series as input."
        self.ts = ts
        self.freq = self.ts.index.freq
        self.y = ts.values
        self.idx = ts.index.values # store datetime index
        self.slen = ts.shape[0] # length of SI data
        self.x = np.arange(1, self.slen + 1, 1) # discrete vector

        # fix as column vectors
        self.x.shape = (self.slen, 1)
        self.y.shape = (self.slen, 1)

        # set seed
        # np.random.seed(42)

    def fit(self, ar_order=1, ar_trend='n'):
        """fit the 4 model parameters"""
        # OLS
        self.ar_order = ar_order
        self.ar_trend = ar_trend
        xvec = np.reshape(self.x, (-1, 1))
        xvec = sm.add_constant(xvec)  # to add intercept
        ols = sm.OLS(self.y, xvec).fit()  # fit regression
        self.beta0 = ols.params[0] # OLS intercept
        self.beta1 = ols.params[1] # OLS slope
        self.trend_line = ols.predict(xvec) # trend line
        self.trend_line.shape = (self.slen, 1)
        self.residuals = self.y - self.trend_line # residuals

        # AR
        AR = AutoReg(endog=self.residuals, lags=self.ar_order, trend=self.
↪ar_trend) # fit AR process
        ARfit = AR.fit(cov_type="HC0") # robust SE
        html = ARfit.summary().as_html() # save model results
        self.sd = float(pd.read_html(html)[0].iloc[2,3]) # get sd
        self.alpha = float(pd.read_html(html)[1].iloc[1,1]) # get autocorrelation
        return self
```

```
    def monte_carlo(self, n=1):
        """Do n simulations."""
        self.out_shape = (self.slen, n)

        # trend matrix
        trend_mat = np.tile(self.trend_line, (1, n))

        # white noise matrix
        white_noise = np.random.normal(0, self.sd, size=self.out_shape)

        # red noise matrix: fill iteratively
        red_noise = np.empty_like(white_noise)
        for pos in np.arange(1, self.slen):
            red_noise[pos,:] = self.alpha * red_noise[pos-1,:] +␣
↪white_noise[pos,:]

        # compute sum
        xt = trend_mat + red_noise

        # to dataframe
        self.simulation = pd.DataFrame(data=xt, index=self.idx, columns=np.
↪arange(1, xt.shape[1] + 1))
        return self

    def plot(self, what='obs'):
        """Plot obs, sim or extrp"""
        if what not in ['obs', 'sim', 'extrp']:
            raise IOError("Data does not exist.")

        if what == 'obs':
            data = self.ts
        elif what == 'sim':
            data = self.simulation
        else:
            data = self.extrapolation

        # compute statistics
        if data.shape[1] > 1:
            self.quantiles = data.quantile(q=[0, 0.025, 0.25, 0.5, 0.75, 0.975,␣
↪1], axis=1).transpose()

            # plot simulation results
            # ------------------------------
            f, ax = plt.subplots(figsize=(12,8))
            # median
            self.quantiles[0.5].plot(ax=ax, color='blue', alpha=0.3)
```

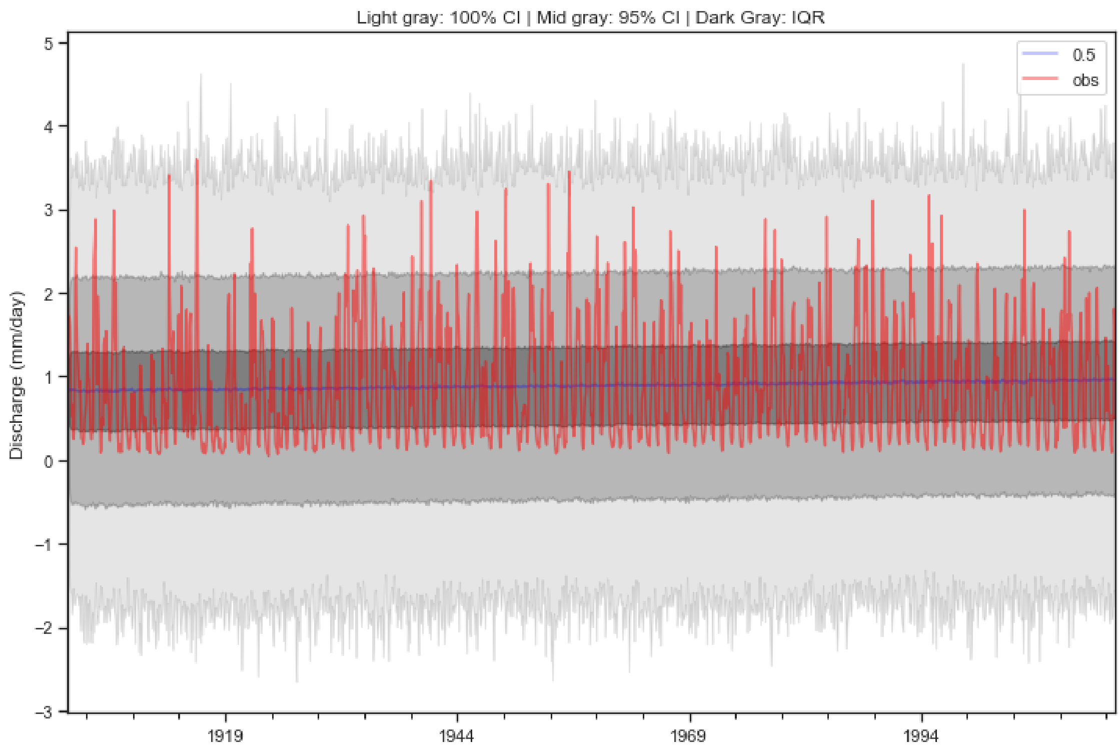
```
            # observations
            ts_obs = pd.Series(data=np.ravel(self.y), index=self.idx)
            ts_obs.name = 'obs'

            # min - max
            ax.fill_between(self.quantiles.index.values,
                            self.quantiles[0],
                            self.quantiles[1],
                            color='black', alpha=0.1)

            # 2.5% - 97.5%
            ax.fill_between(self.quantiles.index.values,
                            self.quantiles[0.025],
                            self.quantiles[0.975],
                            color='black', alpha=0.2)

            # 25% - 75%
            ax.fill_between(self.quantiles.index.values,
                            self.quantiles[0.25],
                            self.quantiles[0.75],
                            color='black', alpha=0.3)
            ts_obs.plot(ax=ax, color='red', alpha=0.5)
           ax.set_title("Light gray: 100% CI | Mid gray: 95% CI | Dark Gray: IQR")
        else:
            f, ax = plt.subplots(figsize=(12,8))
            data.plot(ax=ax, color='blue', alpha=0.3)
            #self.ts.plot(ax=ax, color='red', alpha=0.3)
        # common stuff

        ax.set_ylabel("Discharge (mm/day)")
        plt.legend()
        plt.show()

    def extrapolate(self, until, n=1):
        """Extrapolate time series into the future based on the fitted AR model.
↪"""

        # construct index
        last_obs = self.ts.index[-1]
        first_extrp = last_obs + relativedelta(months=1)
        new_idx = pd.date_range(first_extrp, until, freq='M')
        combined_idx = pd.date_range(self.ts.index[0], until, freq='M')

        # extrapolate trend
        new_len = len(new_idx) + self.slen
        new_x = np.arange(1, new_len + 1)[self.slen:]
```

```
        trd_extrp = self.beta0 + self.beta1 * new_x
        trd_extrp.shape = (len(trd_extrp), 1)
        self.slen_extrp = len(new_idx)
        out_shape_extrp = (self.slen_extrp, n)

        # TODO: initialise AR process at last observation

        # simulate AR process
        trend_mat = np.tile(trd_extrp, (1, n))

        # extrapolate stochastic components
        white_noise = np.random.normal(0, self.sd, size=out_shape_extrp)

        # red noise matrix: fill iteratively
        red_noise = np.empty_like(white_noise)

        for pos in np.arange(1, self.slen_extrp):
            red_noise[pos,:] = self.alpha * red_noise[pos-1,:] +␣
↪white_noise[pos,:]

        # compute sum
        xt = trend_mat + red_noise


        # combine with observations
        observations = np.tile(self.y, (1, n))
        combined_series = np.concatenate((observations, xt), axis=0)

        # to series
        self.extrapolation = pd.DataFrame(data=combined_series,
                                          index=combined_idx,
                                          columns=np.arange(1, combined_series.
↪shape[1] + 1))
        return self
```

1.1 Monte Carlo Simulation for Ilisu

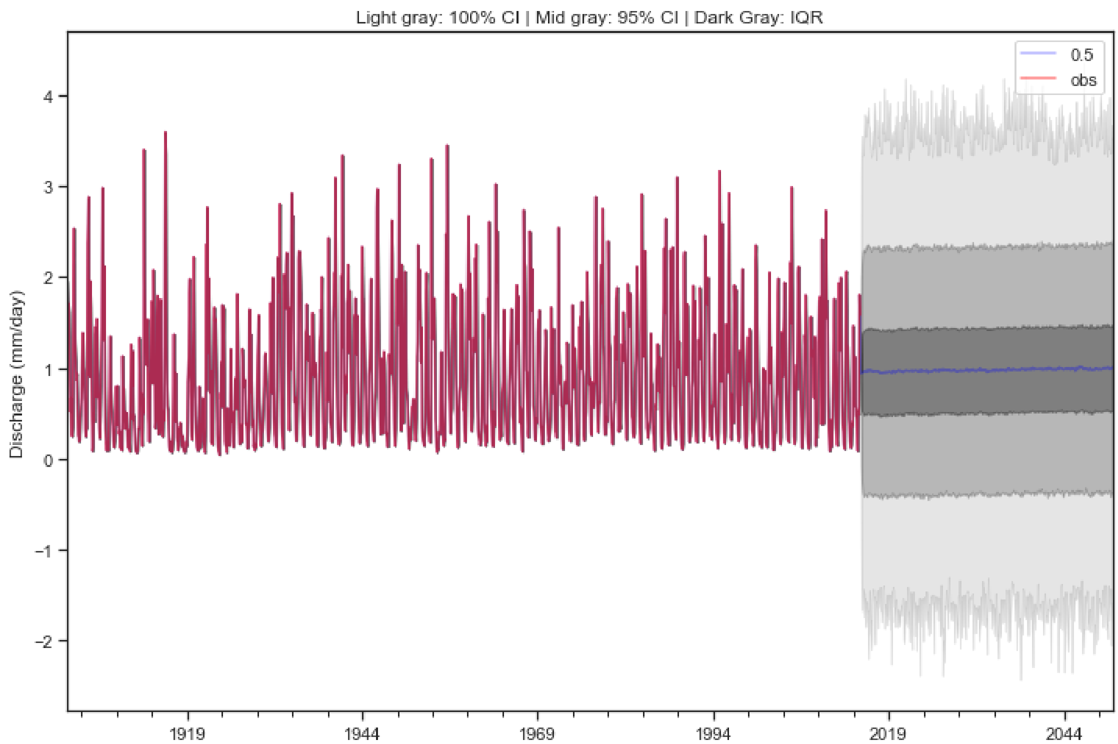
[6]:
```
# run simulation
n_realisations = 10000
model = TimeSeriesModel(ts=df.ilisu)
model = model.fit()
model = model.monte_carlo(n=n_realisations)
```

[7]:
```
model.plot(what='sim')
```



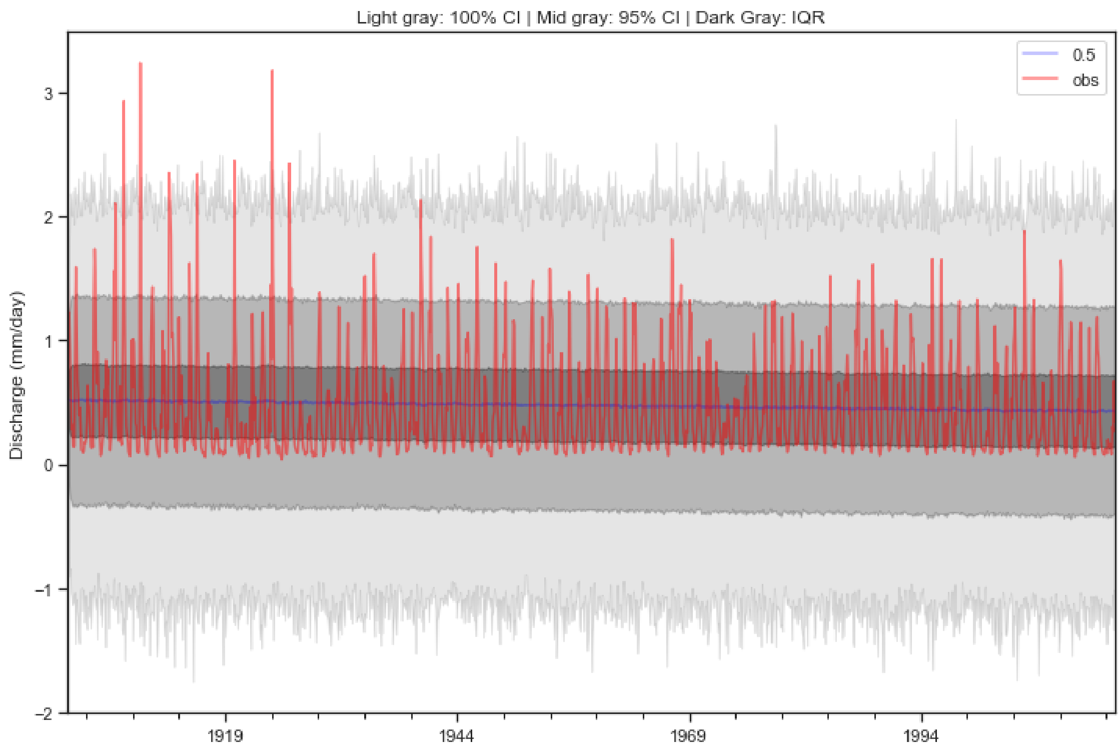
extrapolate to ~ 2030. TODO: figure out where exactly to initialise the AR process (last obs, right?)

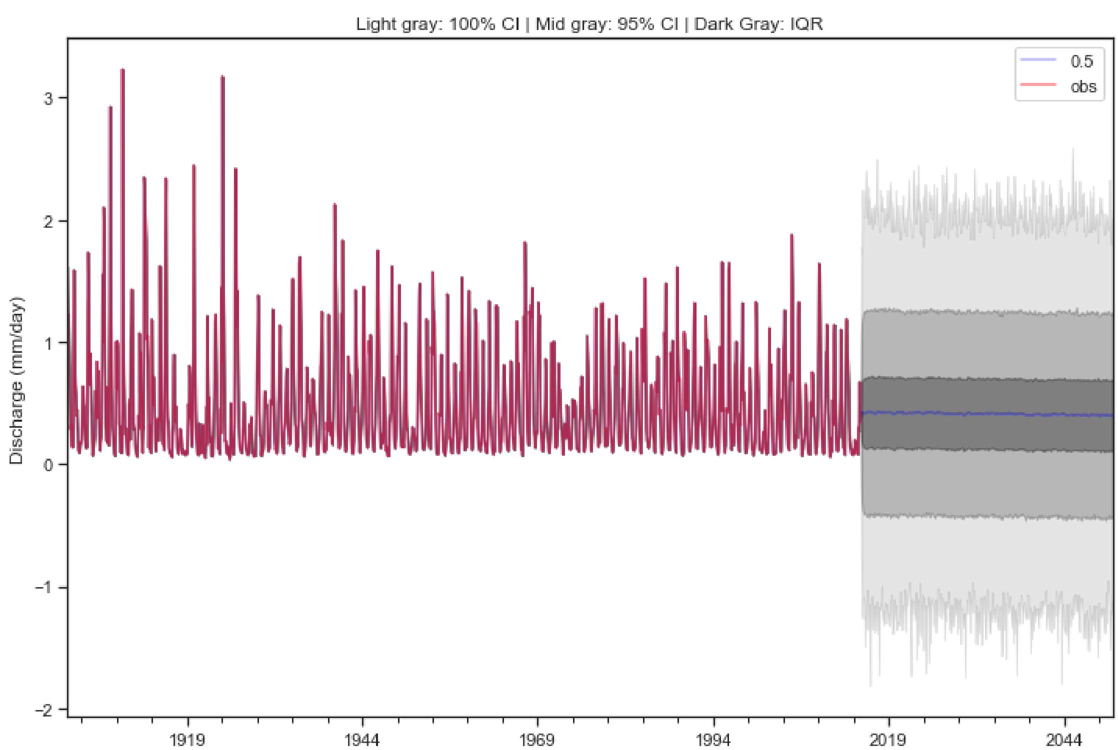
[8]:
```
model = model.extrapolate(until='2050-12-31', n=n_realisations)
model.plot(what='extrp')
```



1.2 Monte Carlo Simulation for Mosul

[9]:
```
# run simulation
model = TimeSeriesModel(ts=df.mosul)
model = model.fit()
model = model.monte_carlo(n=n_realisations)

# plot simulations
model.plot(what='sim')

# extrapolate
model = model.extrapolate(until='2050-12-31', n=n_realisations)
model.plot(what='extrp')
```


Light gray: 100% CI | Mid gray: 95% CI | Dark Gray: IQR
0.5
obs
Discharge (mm/day)
3
2
1
0
−1
−2
1919
1944
1969
1994


Light gray: 100% CI | Mid gray: 95% CI | Dark Gray: IQR
0.5
obs
Discharge (mm/day)
3
2
1
0
−1
−2
1919
1944
1969
1994
2019
2044