Nonlinear autoregressive time series models in R using `tsDyn` version 0.7

last revised 11/03/2008 by Antonio, Fabio Di Narzo

1 Introduction

`tsDyn` is an R package for the estimation of a number of nonlinear time series models. The package is at an early stage, and may presumably change significantly in the near future. However, it is quite usable in the current version.

Each function in the package has at least a minimal help page, with one or more working examples and detailed explanation of function arguments and returned values. In this document we try to give an overall guided tour of package contents, with some additional notes which are generally difficult to put in the context of a manual.

This guide is divided into 3 main sections:

- Explorative analysis tools
- Nonlinear autoregressive models
- A case study

2 Explorative analysis

2.1 Bivariate and trivariate relations

A first explorative analysis should include inspecting the distribution of (x_t, x_{t-l}) and that of $(x_t, x_{t-l_1}, x_{t-l_2})$ for some lags l, l_1, l_2. This can be done easily in R in a variety of ways. The `tsDyn` package provide functions `autopairs` and `autotriples` for this purpose.

The `autopairs` function displays, in essence, a scatterplot of time series x_t versus x_{t-lag}. The main arguments to the function are the time series and the desired lag. The scatterplot may be also processed to produce bivariate kernel density estimations, as well as nonparametric kernel autoregression estimations. The type of output is governed by the argument `type`. Possibile values, along with their meanings, are:

`lines`	directed lines
`points`	simple scatterplot
`levels`	iso-density levels
`persp`	density perspective plot
`image`	density image map
`regression`	kernel autoregression line superposed to scatterplot

For kernel density and regression estimation, you can specify also the kernel window `h`. A typical call to that function can be:

R code
```
autopairs(x, lag = , type = , h = )
```

All arguments (except the time series `x`) have default values.

By default, if running in an interactive environment, the function displays a simple experimental cross-platform GUI, where you can change function parameters and watch interactively how the plot changes.

Similar to `autopairs`, there is the `autotriples` function. This shows x_t versus (x_{t-lag1}, x_{t-lag2}), so that the user has to specify time series x and lags `lag1` and `lag2`. The scatterplot can be processed to produce kernel regression estimates. Plotting possibilities are:

`levels`	iso-values lines
`persp`	perspective plot
`image`	image map
`lines`	directed lines
`points`	simple scatterplot

2.2 Linearity

An interesting tool for inspecting possible nonlinearities in the time series is the *locally linear autoregressive fit* plot, proposed by Casdagli˜[1]. Suppose you think that the dynamical system underlying your time series is best reconstructed with embedding dimension m and time delay d. Then the locally linear autoregressive fit plot displays the relative error made by forecasting time series values with linear models of the form:

$$x_{t+s} = \phi_0 + \phi_1 x_t + \ldots + \phi_m x_{t-(m-1)d}$$

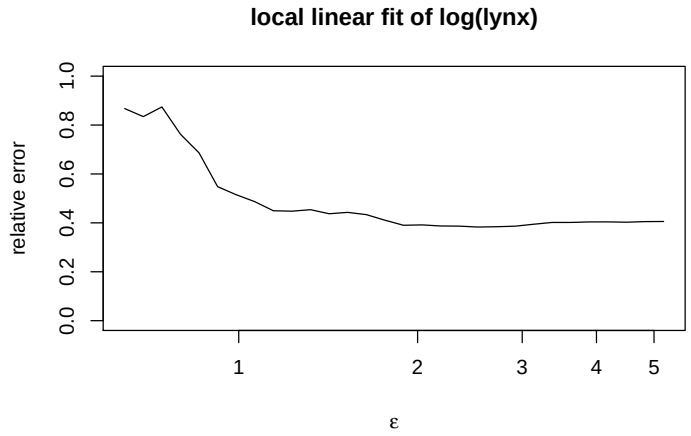
estimated on points in the sphere of radius ϵ around $\mathbf{x}_t^m$ for a range of values of ϵ. A minimum attained at relatively small values of ϵ may indicate that a global linear model would be inappropriate for the approximation of the time series dynamics.

For this analysis `tsDyn` proposes the function `llar` which accepts, among others, the following arguments:

`x`	time series
`m, d, steps`	embedding parameters (see the above model formulation)

The function returns a '`llar`' object, which can be plotted with the generic `plot` method. So, a typical usage would be:

R code
```
obj <- llar(log(lynx), m = 3)
plot(obj)
```



However, the `obj` object can be explicitely converted in an ordinary data.frame:

R code
```
obj <- data.frame(obj)
```

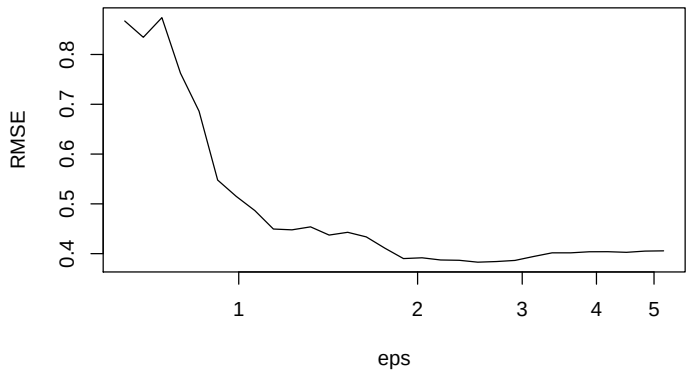
with variables:

R code
```
names(obj)
```

output
```
[1] "RMSE"    "eps"     "frac"    "avfound"
```

where '`RMSE`' stands for Relative Mean Square Error, and `eps` is enough self-explaining. You can explore this object with usual R commands dedicated to data.frames, such as:

R code
```
plot(RMSE ~ eps, data = obj, type = "l", log = "x")
```



2.3 Tests (experimental)

`tsDyn` implements conditional mutual independence and linearity tests as described in Manzan˜[7]. Function implementations are rather basic, and little tested. Use them carefully!

The `delta.test` function performs a bootstrap test of independence of x_t versus x_{t-md} conditional on intermediate observations $\{x_{t-d}, \ldots, x_{t-(m-1)d}\}$. The test statistic, available with the function `delta`, is based on the sample correlation integral, and calls internally the `d2` function provided by the `tseriesChaos` package. Among others things, the test requires the specification of a neighborhood window ϵ.
Function arguments are the time series `x`, a vector of embedding dimensions `m`, time delay `d`, a vector of neighborhood windows `eps`, the number of bootstrap replications `B`. However, default values are available for `m, d, eps` and `B`, so that a typical call can be:

R code

```
delta.test(x)
```

The return value is a matrix of p-values, labelled with their associated embedding dimensions and neighborhood windows (normally multiple values are tried simultaneously).

The `delta.lin.test` function performs a bootstrap test of linear dipendence of x_t versus x_{t-md} conditional on intermediate observations $\{x_{t-d}, \ldots, x_{t-(m-1)d}\}$. The test statistic is available with the function `delta.lin`. The function arguments and returned values are the same as those of `delta.test`.

3 Nonlinear autoregressive time series models

Consider the discrete-time univariate stochastic process $\{X_t\}_{t\in T}$. Suppose X_t is generated by the map:

$$X_{t+s} = f(X_t, X_{t-d}, \ldots, X_{t-(m-1)d}; \theta) + \epsilon_{t+s} \tag{1}$$

with $\{\epsilon_t\}_{t\in T}$ white noise, ϵ_{t+s} indipendent w.r.t. X_{t+s}, and with f a generic function from $\mathbf{R}^m$ to $\mathbf{R}$. This class of models is frequently referenced in the literature with the acronym NLAR(m), which stands for *NonLinear AutoRegressive* of order m.

In (1), we have implicitely defined the *embedding dimension* m, the *time delay* d and the *forecasting steps* s. The vector θ indicates a generic vector of parameters governing the shape of f, which we would estimate on the basis of some empirical evidence (i.e., an observed time series $\{x_1, x_2, \ldots, x_N\}$).

In `tsDyn` some specific NLAR models are implemented. For a list of currently available models, type:

R code
```
availableModels()
```

output
```
[1] "linear" "nnetTs" "setar"  "lstar"  "star"   "aar"
```

Each model can be estimated using a function which takes the name of the model as indicated by `availableModels`. I.e., use `linear` for fitting a linear model.

All those functions returns an object of base class `nlar`, from which informations can be extracted using some common methods. Among others:

```
print(obj) #prints basic infos on fitted model and estimated parameters
summary(obj) #if possible, shows more detailed infos and diagnostics on estimated model
plot(obj) #shows common diagnostic plots
```

Another method that can be useful for inspecting the estimated model properties is the `predict` method:

R code
```
x.new <- predict(obj, n.ahead = )
```

This function attempts to extend of `n.ahead` observations of the original time series used for estimating the model encapsulated in `obj` using the so called

skeleton of the fitted model. Assuming that from (1) we estimated f as $\hat{f} = f(\cdot;\hat{\theta})$, using the time series $\mathbf{x} = \{x_1, x_2, \ldots, x_N\}$, we have:

$$\begin{aligned}
\hat{x}_{N+1} &= \hat{f}(x_{N-s}, x_{N-s-d}, \ldots, x_{N-s-(m-1)d}) \\
\hat{x}_{N+2} &= \hat{f}(x_{N-s+1}, x_{N-s+1-d}, \ldots, x_{N-s+1-(m-1)d}) \\
&\ldots \\
\hat{x}_{N+S} &= \hat{f}(x_{N-s+(S-1)}, x_{N-s+(S-1)-d}, \ldots, x_{N-s+(S-1)-(m-1)d})
\end{aligned}$$

A detailed description of some actually implemented models follows.

3.1 Linear models

$$X_{t+s} = \phi + \phi_0 X_t + \phi_1 X_{t-d} + \ldots + \phi_m X_{t-(m-1)d} + \epsilon_{t+s} \tag{2}$$

It's a classical AR(m) model, and its specification doesn't require additional hyper-parameters. Estimation is done via CLS (Conditional Least Squares). The summary command returns asymptotics standard errors for the estimated coefficients, based on the normality assumption of residuals.

Note that in R there are plenty of functions for AR (and, more generally, ARMA) models estimation, with different estimation methods, such as ML, CLS, Yule-Walker, ... If you really need to fit linear models, use these methods directly.

A interesting reparametrization of the model can be done using differences, as in the ADF test for unit roots:

$$\Delta X_{t+s} = \phi + \rho X_t + \zeta_1 \Delta X_{t-d} + \ldots + \zeta_{m-1} \Delta X_{t-(m-2)d} + \epsilon_{t+s} \tag{3}$$

We have (see Hamilton˜[5]):

- $\rho = \phi_0 + \phi_1 + \ldots + \phi_m$
- $\zeta_i = -(\phi_{j+1} + \phi_{j+2} + \ldots + \phi_m) \qquad \text{for } j = 1, 2, \ldots, m-1$

It has the major advantage that the stationarity condition that all the roots of the polynomial in (2) lying in the unit root circle is in the ADF representation simply that $-1 < \rho < 0$.

This can be set in `tsDyn` using the argument `type` to `ADF` and using one lag less in `m`:

R code

```
usual <- linear(lynx, m = 3)
adf <- linear(lynx, m = 2, type = "ADF")
```

Note that it is only a reparametrization, the model estimated being the same:

```
R code
all.equal(deviance(adf), deviance(usual))
```

```
output
[1] TRUE
```

```
R code
all.equal(residuals(usual), residuals(adf))
```

```
output
[1] TRUE
```

3.2 SETAR models

$$X_{t+s} = \begin{cases} \phi_1 + \phi_{10} X_t + \phi_{11} X_{t-d} + \ldots + \phi_{1L} X_{t-(L-1)d} + \epsilon_{t+s} & Z_t \leq \text{th} \\ \\ \phi_2 + \phi_{20} X_t + \phi_{21} X_{t-d} + \ldots + \phi_{2H} X_{t-(H-1)d} + \epsilon_{t+s} & Z_t > \text{th} \end{cases} \tag{4}$$

with Z_t a threshold variable. How is one to define Z_t? Strictly speaking, in SETAR models Z_t should be one of $\{X_t, X_{t-d}, X_{t-(m-1)d}\}$. We can define the threshold variable Z_t via the *threshold delay* δ, such that

$$Z_t = X_{t-\delta d}$$

Using this formulation, you can specify SETAR models with:

```
R code
obj <- setar(x, m = , d = , steps = , thDelay = )
```

where `thDelay` stands for the above defined δ, and must be an integer number between 0 and $m - 1$.
For greater flexibility, you can also define the threshold variable as an arbitrary linear combination of lagged time series values:

$$Z_t = \beta_1 X_t + \beta_2 X_{t-1} + \ldots + \beta_m X_{t-(m-1)d}$$

In R this is implemented as follows:

```
R code
obj <- setar(x, m = , d = , steps = , mTh = )
```

where `mTh` stands for β, and takes the form of a vector of real coefficients of length m.
Finally, Z_t can be an external variable. This is obtained with the call:

```R code
obj <- setar(x, m = , d = , steps = , thVar = )
```

where `thVar` is the vector containing the threshold variable values.

Models with two thresholds and hence three regime are aslo availabe through the argument `nthresh`, its default value being 1.

```R code
obj <- setar(x, m = , d = , steps = , thDelay = , nthresh = 2)
```

Another hyper-parameter one can specify is the threshold value th, via the additional argument `th`. If not specified, this is estimated by fitting the model for a grid of different, by default all values of th, and taking the best fit as the final th estimate. This is done calling internally selectSETAR() with `criterion="SSR"`.

Note that, conditional on $\{Z_t \leq \text{th}\}$, the model is linear. So, for a fixed threshold value, the CLS estimation is straightforward.

The summary command for this model returns asymptotic standard errors for the estimated ϕ coefficients, based on the assumption that ϵ_t are normally distributed.

The threshold variable isn't the only additional parameter governing the TAR model. One can specify the *low* and *high* regime autoregressive orders L and H. These can be specified with the arguments `mL` and `mH`, respectively:

```R code
obj <- setar(x, m = , d = , steps = , thDelay = , mL = , mH = )
```

If not specified, `mL` and `mH` defaults to `m`. One can decide also only to select a few values between `1:mL` and `1:mH`. This is possible using `ML` and `MH`. Hence to have a first regime with lag 1 and 3, the second with all 3, would be: `ML=c(1,3)` and `MH=1:3`.

```R code
obj <- setar(x, m = , d = , steps = , thDelay = , ML = , MH = )
```

As suggested in Enders and Granger (1997)[3], the threshold variable can be in differences, leading to the so-called Momentum-TAR (M-TAR). In this case, the regime switching depends not on the position of the variable at time $t-1$ but on its signs at $t-1$. Hence, on can estimate whethera variable behaves differently whether it was previously increasing or decreasing. A M-TAR can be specified setting the argument `model="MTAR"`. Note that when the number of lags `m` is equal to the delay `d`, there is one less observation in the series[1].

[1] This is for now handled not so properly and may result in failures in the different methods

An interesting specification of the model in terms of another representation is possible with `type = c("level", "diff", "ADF")`. This will either set all variables in levels, in difference or as in the specification of the ADF test:

Note that using `type=level` or `ADF` results in the same model fit but is a convenient way to test for a unit root, as the value of the variable in levels should be smaller than one.

3.3 LSTAR models

The LSTAR model can be viewed as a generalization of the above defined SETAR model:

$$\begin{aligned} X_{t+s} = \quad & (\phi_1 + \phi_{10} X_t + \phi_{11} X_{t-d} + \ldots + \phi_{1L} X_{t-(L-1)d})(1 - G(Z_t, \gamma, \text{th})) \\ & + (\phi_2 + \phi_{20} X_t + \phi_{21} X_{t-d} + \ldots + \phi_{2H} X_{t-(H-1)d}) G(Z_t, \gamma, \text{th}) + \epsilon_{t+s} \end{aligned}$$

with G the logistic function, and Z_t the threshold variable. For Z_t, L and H specification, the same convention as that of SETAR models is followed. In addition, for LSTAR models one has to specify some starting values for all the parameters to be estimated: $(\phi, \gamma, \text{th})$.

Estimation is done by analytically determining ϕ_1 and ϕ_2 (through linear regression) and then minimizing residuals sum of squares with respect to th and γ. These two steps are repeated until convergence is achieved.

3.4 Neural Network models

A neural network model with linear output, D *hidden units* and activation function g, is represented as:

$$x_{t+s} = \beta_0 + \sum_{j=1}^{D} \beta_j g(\gamma_{0j} + \sum_{i=1}^{m} \gamma_{ij} x_{t-(i-1)d}) \tag{5}$$

For the implementation the `nnet` package is used, so please refer to the `nnet` package documentation for more details.

The only additional argument for specifying this model is the number of hidden units `size`, which stands for the above defined D:

R code

```
obj <- nnetTs(x, m = , d = , steps = , size = )
```

The estimation is done via CLS. No additional summary informations are available for this model.

3.5 Additive Autoregressive models

A non-parametric additive model (a GAM, Generalized Additive Model), of the form:

$$x_{t+s} = \mu + \sum_{i=1}^{m} s_i(x_{t-(i-1)d}) \tag{6}$$

where s_i are smooth functions represented by penalized cubic regression splines. They are estimated, along with their degree of smoothing, using the `mgcv` package˜[10].

No additional parameters are required for this model:

R code

```
obj <- aar(x, m = , d = , steps = )
```

Some diagnostic plots and summaries are provided for this model, adapted from those produced by `mgcv`.

3.6 Model selection

A common task in time series modelling is *fine tuning* of the hyper-parameters. R is a complete programming language, so the user can easily define his error criterion, fit a set of models and chose the best between them. However, the `tsDyn` package provides some helpful functions for this task.

For SETAR models, there is the `selectSETAR` function. The time series, the embedding parameters and a vector of values for each provided hyper-parameter is passed to this function. The routine then tries to fit the model for the full grid of hyper-parameter values, and gives as output a list of the best combinations found. So, for example:

R code

```
x <- log10(lynx)
selectSETAR(x, m = 3, mL = 1:3, mH = 1:3, thSteps = 5, thDelay = 0:2)
```

output

```
Searching on 73 possible threshold values within regimes with sufficient ( 15% ) number of observati
Searching on  219  combinations of thresholds ( 73 ), thDelay ( 3 ), mL ( 1 ) and MM ( 3 )
Results of the grid search for 1 threshold
   thDelay mL mH       th pooled-AIC
1        2  1  1 2.940018 -15.612619
2        2  1  1 2.907411 -12.252530
3        2  1  1 3.000000 -10.638946
4        2  1  1 2.894316  -9.038100
5        2  1  1 2.980912  -7.812850
```

```
6        2  1  1 2.879669  -6.462867
7        2  1  1 2.828660  -4.661813
8        2  1  1 2.820858  -4.382119
9        2  1  1 2.835056  -3.043105
10       2  1  1 2.866878  -2.127289
```

tries to fit $3 \times 3 \times 3 \times 5$ models, one for each combination of `mL`,`mH`,`thDelay` and `th`, and returns the best combinations w.r.t. the AIC criterion.

Totally analogous are the `selectLSTAR` and `selectNNET` functions, for which we refer to the online documentation.

4 Case study

We herein analyse the Canadian lynx data set. This consists of annual records of the numbers of the Canadian lynx trapped in the Mackenzie River district of North-west Canada for the period 1821-1934.

The time series, named `lynx`, is available in a default R installation, so one can type directly, in an R session:

R code

```
str(lynx)
```

output

```
Time-Series [1:114] from 1821 to 1934: 269 321 585 871 1475 ...
```

R code

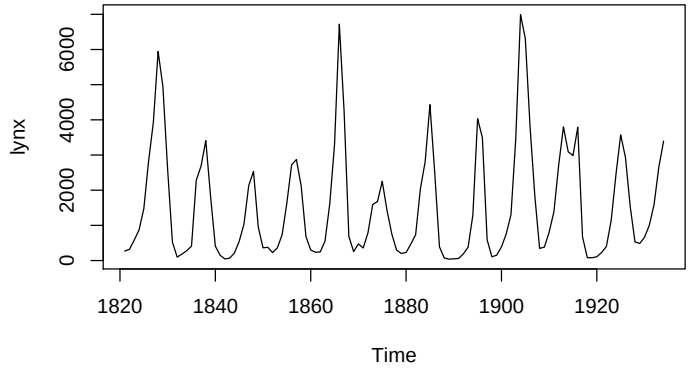
```
summary(lynx)
```

output

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   39.0   348.2   771.0  1538.0  2567.0  6991.0
```

R code

```
plot(lynx)
```



Here we will roughly follow the analysis in Tong˜[9].

4.1 Explorative analysis

First, we log transform the data:

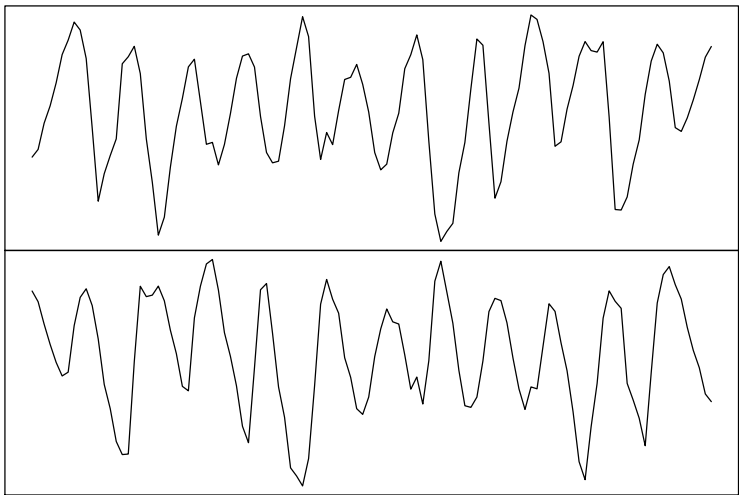
R code

```
x <- log10(lynx)
```

Plot of the time series and time-inverted time series:

R code

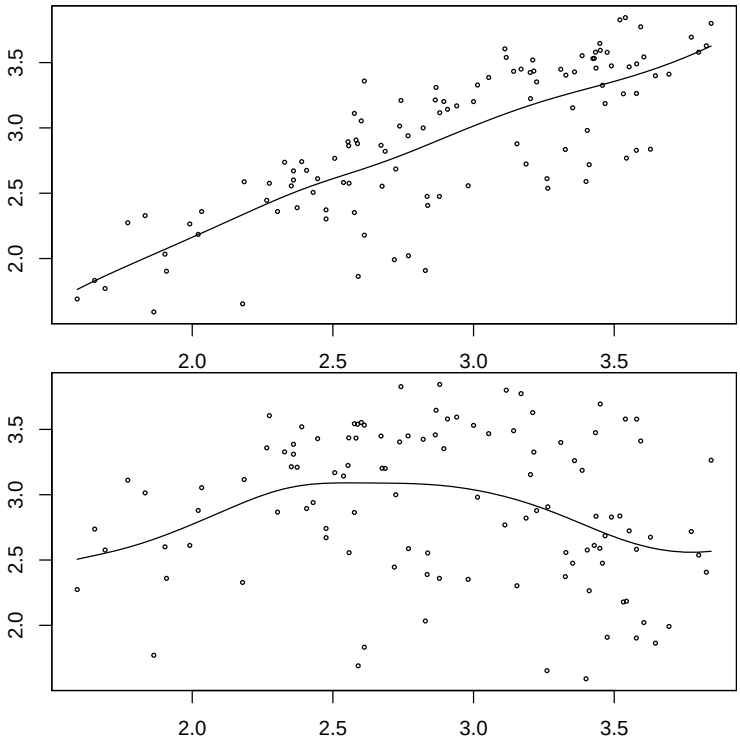
```
par(mfrow = c(2, 1), mar = c(0, 0, 0, 0))
plot(x, ax = F)
box()
plot(x[length(x):1], type = "l", ax = F)
box()
```



Nonparametric regression function of X_t versus X_{t-1} and of X_t versus X_{t-3} (kernel estimation):

R code

```
par(mfrow = c(2, 1), mar = c(2, 2, 0, 0))
autopairs(x, lag = 1, type = "regression", GUI = FALSE)
autopairs(x, lag = 3, type = "regression", GUI = FALSE)
```

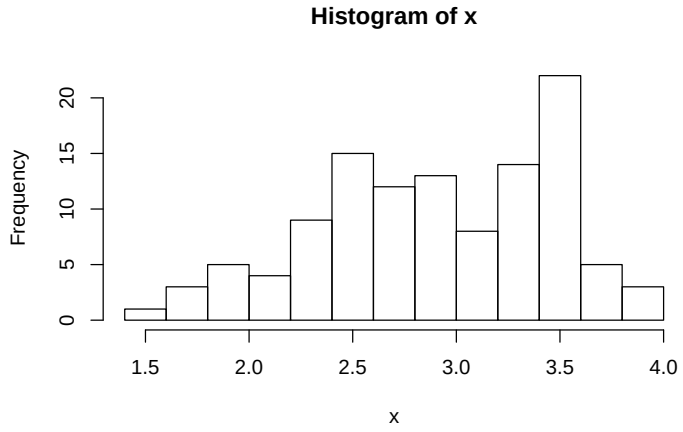


For lag 3 (bottom plot), a linear approximation for the regression function may be questionable.

The marginal histogram of data shows bimodality:

R code

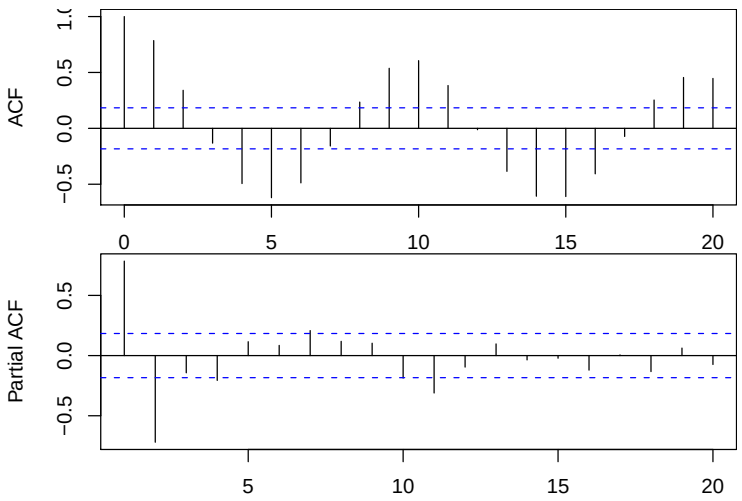
```
hist(x, br = 13)
```



Global and partial autocorrelation:

R code

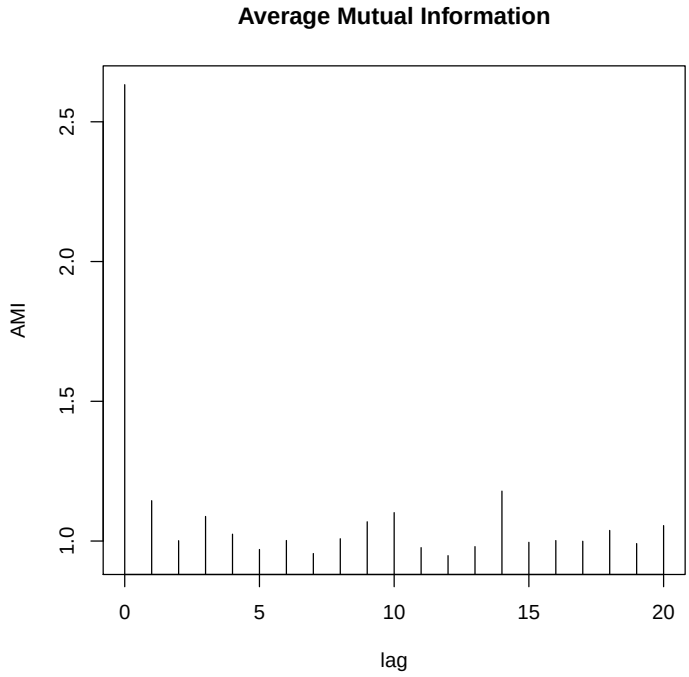
```
par(mfrow = c(2, 1), mar = c(2, 4, 0, 0))
acf(x)
pacf(x)
```



The `tseriesChaos` package offers some other explorative tools tipycal of nonlinear time series analysis. The Average Mutual Information (see online help for further explanation):

R code

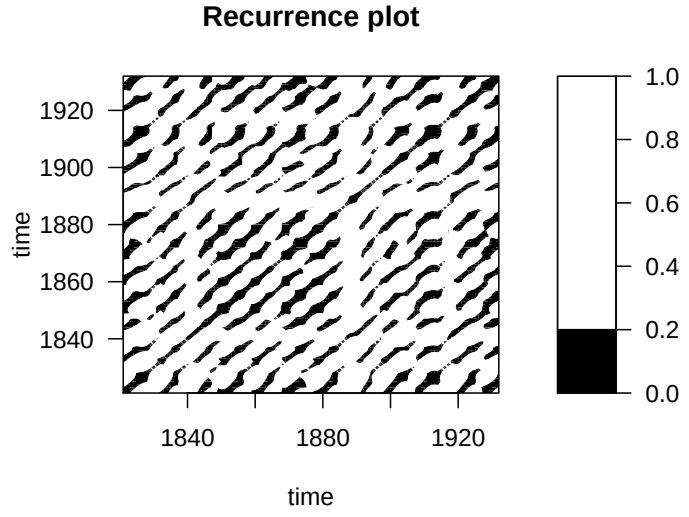
```
library(tseriesChaos)
mutual(x)
```



Recurrence plot (see online help):

R code

```
recurr(x, m = 3, d = 1, levels = c(0, 0.2, 1))
```

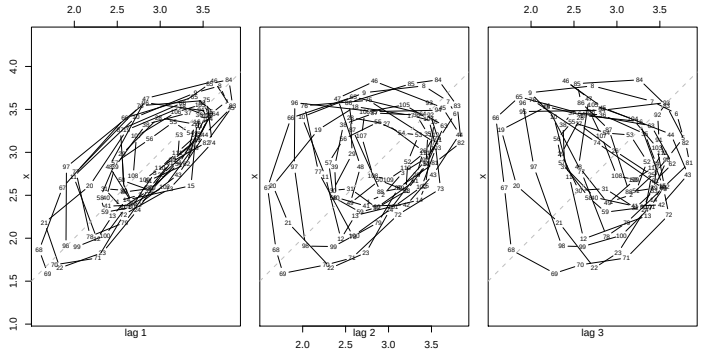


From this plot, deterministic cycles appears from the embedding-reconstructed underlying dynamics.

Directed lines are a tipycal tool for time series explorations. The `lag.plot` function in the base `stats` package does this well:

R code

```
lag.plot(x, lags = 3, layout = c(1, 3))
```



Especially for lag 2, a cycle is again evident. Moreover, the void space in the middle is a typical argument for rejecting the bivariate normality of (X_t, X_{t-l}).

What follows is the application of still-experimental code for testing the conditional mutual independence and linearity for lags 2 and 3:

R code

```
delta.test(x)
```

output

```
   eps
m   0.2792 0.5584 0.8376 1.1168
  2   0.02   0.02   0.02   0.02
  3   0.28   0.02   0.02   0.02
```

R code

```
delta.lin.test(x)
```

output

```
   eps
m   0.2792 0.5584 0.8376 1.1168
  2   0.36   0.12   0.52   0.66
  3   0.32   0.26   0.04   0.54
```

P-values are reported, labelled with their embedding dimensions m and *window* values ϵ. We reject conditional independence quite easily. There is some trouble instead for deciding to reject or not linearity. See Manzan˜[7] for a detailed discussion on these tests.

4.2 Model selection

The first model proposed in the literature for these data was an AR(2):

$$X_t = 1.05 + 1.41X_{t-1} - 0.77X_{t-2} + \epsilon_t$$

with $v(\epsilon_t) = \sigma^2 = 0.04591$.

This can be estimated with `tsDyn` using the command:

R code

```
mod.ar <- linear(x, m = 2)
mod.ar
```

output

```
Non linear autoregressive model

AR model
Coefficients:
     const      phi.1      phi.2
 1.0576005  1.3842377 -0.7477757
```

As an improvement to the AR model, we may consider applying a SETAR(2; 2,2) model with threshold delay $\delta = 1$. In R:

R code

```
mod.setar <- setar(x, m = 2, mL = 2, mH = 2, thDelay = 1)
mod.setar
```

output

```
Non linear autoregressive model

SETAR model ( 2 regimes)
Coefficients:
Low regime:
    phiL.1     phiL.2    const L
 1.2642793 -0.4284292  0.5884369

High regime:
   phiH.1    phiH.2   const H
 1.599254 -1.011575  1.165692

Threshold:
-Variable: Z(t) = + (0) X(t)+ (1)X(t-1)
-Value: 3.31
Proportion of points in low regime: 69.64%          High regime: 30.36%
```

So, the fitted model may be written as:

$$X_{t+1} = \begin{cases} 0.588 + 1.264X_t - 0.428X_{t-1} & X_{t-1} \leq 3.31 \\ 1.166 + 1.599X_t - 1.012X_{t-1} & X_{t-1} > 3.31 \end{cases}$$

For an automatic comparison, we may fit different linear and nonlinear models and directly compare some measures of their fit:

R code

```
mod <- list()
mod[["linear"]] <- linear(x, m = 2)
mod[["setar"]] <- setar(x, m = 2, thDelay = 1)
mod[["lstar"]] <- lstar(x, m = 2, thDelay = 1)
```

output

```
Using maximum autoregressive order for low regime: mL = 2
Using maximum autoregressive order for high regime: mH = 2
Performing grid search for starting values...
Starting values fixed: gamma =  11 , th =  3.338679 ; SSE =  4.337657
Optimization algorithm converged
Optimized values fixed for regime 2  : gamma =  11.00002 , th =  3.340093
```

R code

```
mod[["nnetTs"]] <- nnetTs(x, m = 2, size = 3)
mod[["aar"]] <- aar(x, m = 2)
```

Now the `mod` object contains a labelled list of fitted `nlar` models. As an example, we can compare them in term of the AIC and MAPE index:

R code

```
sapply(mod, AIC)
```

output

```
   linear     setar     lstar    nnetTs       aar
-333.8737 -358.3740 -356.6509 -344.4731 -328.0813
```

R code

```
sapply(mod, MAPE)
```

output

```
    linear      setar      lstar     nnetTs        aar
0.06801955 0.05648596 0.05580458 0.05709281 0.05951108
```

From this comparison, the SETAR model seems to be the best.

More detailed diagnostics can be extracted:

R code

```
summary(mod[["setar"]])
```

output

```
Non linear autoregressive model

SETAR model ( 2 regimes)
Coefficients:
Low regime:
    phiL.1     phiL.2    const L
 1.2642793 -0.4284292  0.5884369

High regime:
   phiH.1    phiH.2   const H
 1.599254 -1.011575  1.165692

Threshold:
-Variable: Z(t) = + (0) X(t)+ (1)X(t-1)
-Value: 3.31
Proportion of points in low regime: 69.64%          High regime: 30.36%

Residuals:
      Min        1Q    Median        3Q       Max
-0.571121 -0.109431  0.017641  0.116468  0.516270

Fit:
residuals variance = 0.03814,  AIC = -358, MAPE = 5.649%

Coefficient(s):

         Estimate  Std. Error  t value  Pr(>|t|)
const L  0.588437    0.143307   4.1061 7.844e-05 ***
phiL.1   1.264279    0.065256  19.3741 < 2.2e-16 ***
phiL.2  -0.428429    0.077487  -5.5291 2.260e-07 ***
const H  1.165692    0.876606   1.3298 0.1863928
phiH.1   1.599254    0.108966  14.6767 < 2.2e-16 ***
phiH.2  -1.011575    0.265011  -3.8171 0.0002255 ***
---
Signif. codes:  0

Threshold
Variable: Z(t) = + (0) X(t) + (1) X(t-1)
```

```
Value: 3.31
```

More diagnostic plots can be displayed using the command:

R code

```
plot(mod[["setar"]])
```

4.3 Out-of-sample forecasting

Fit models on first 104 observations:

R code

```
set.seed(10)
mod.test <- list()
x.train <- window(x, end = 1924)
x.test <- window(x, start = 1925)
mod.test[["linear"]] <- linear(x.train, m = 2)
mod.test[["setar"]] <- setar(x.train, m = 2, thDelay = 1)
mod.test[["lstar"]] <- lstar(x.train, m = 2, thDelay = 1, trace = FALSE,
+     control = list(maxit = 1e+05))
mod.test[["nnet"]] <- nnetTs(x.train, m = 2, size = 3, control = list(maxit = 1e+05))
```

output

```
# weights:  13
initial  value 1009.386247
iter  10 value 5.610020
iter  20 value 5.422071
iter  30 value 5.408916
iter  40 value 5.319344
iter  50 value 5.080819
iter  60 value 4.862858
iter  70 value 4.595347
iter  80 value 4.437773
iter  90 value 4.309204
iter 100 value 4.270208
iter 110 value 4.248650
iter 120 value 4.232032
iter 130 value 4.219283
iter 140 value 4.215564
iter 150 value 4.215542
iter 160 value 4.215540
final  value 4.215539
converged
```

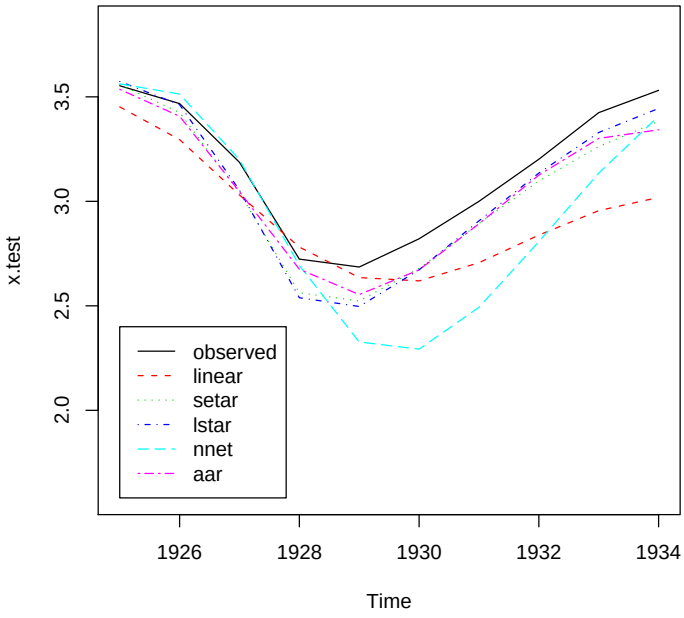
R code

```
mod.test[["aar"]] <- aar(x.train, m = 2)
```

Compare forecasts with real last 10 observed values:

R code

```
frc.test <- lapply(mod.test, predict, n.ahead = 10)
plot(x.test, ylim = range(x))
for (i in 1:length(frc.test)) lines(frc.test[[i]], lty = i +
+     1, col = i + 1)
legend(1925, 2.4, lty = 1:(length(frc.test) + 1), col = 1:(length(frc.test) +
+     1), legend = c("observed", names(frc.test)))
```



From this visual comparison, the SETAR(2; 2,2) model seems to be one of the bests.

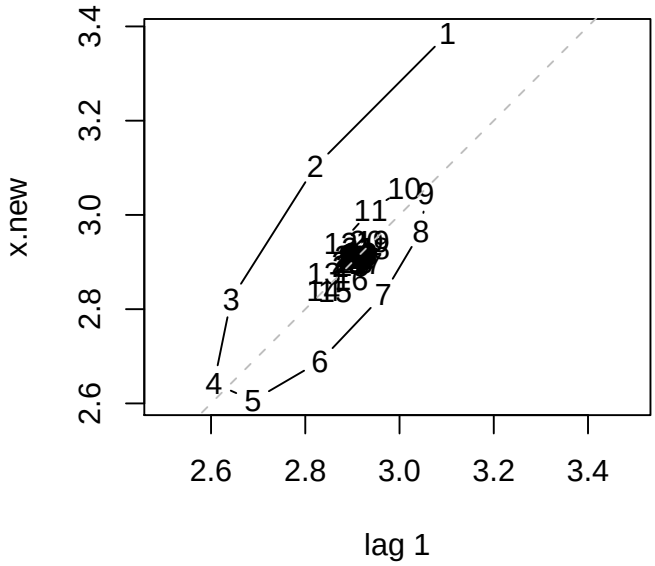
4.4 Inspecting model skeleton

An interesting task can be inspecting the fitted model skeleton.

This can be achieved by comparing the forecasting results under each model.

R code

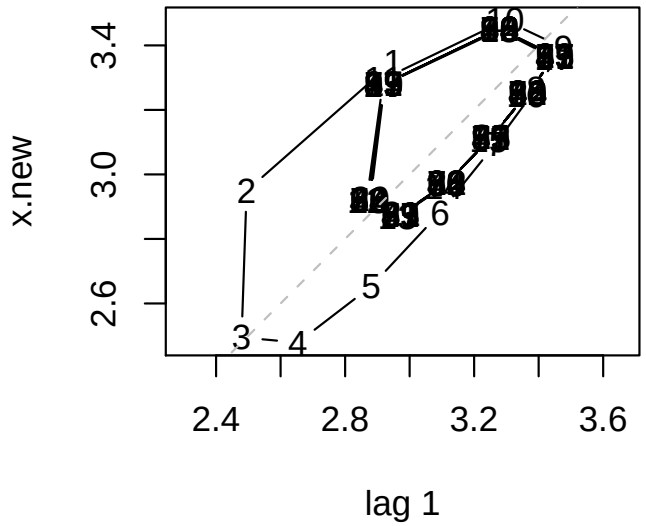
```
x.new <- predict(mod[["linear"]], n.ahead = 100)
lag.plot(x.new, 1)
```



A fixed point, i.e. the only possible stationary solution with a linear model.

R code

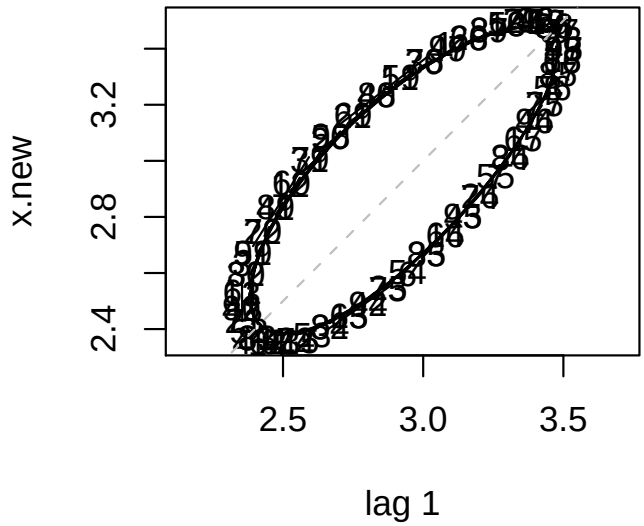
```
x.new <- predict(mod[["setar"]], n.ahead = 100)
lag.plot(x.new, 1)
```



A stable periodic cycle.

R code

```
x.new <- predict(mod[["nnetTs"]], n.ahead = 100)
lag.plot(x.new, 1)
```



Appears to be a quasiperiodic cycle lying on an invariant curve.

5 Sensitivity on initial conditions

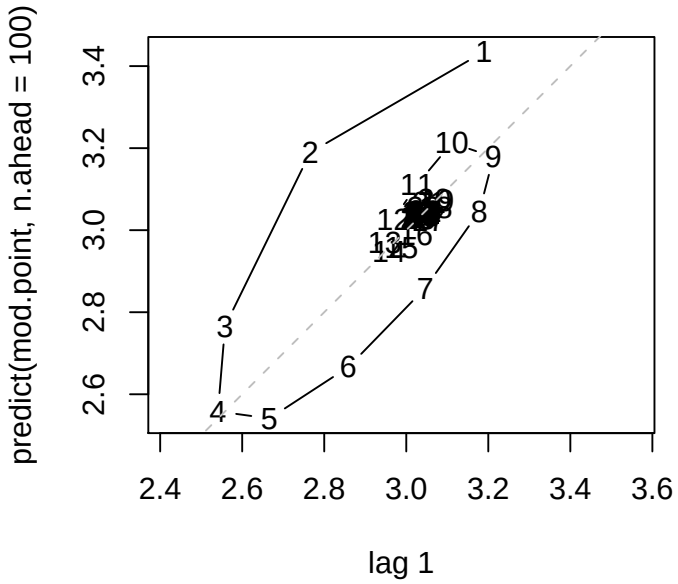
In the previous section we observed skeletons with cyclical or limit fixed point behaviour.

Neural networks and SETAR models can explain also different types of attractors. For this data set, Tong˜[9] showed that particular types of SETAR models can yeld to fixed limit points as well as unstable orbits and *possibly chaotic* systems.

For example, a fixed limit point:

R code

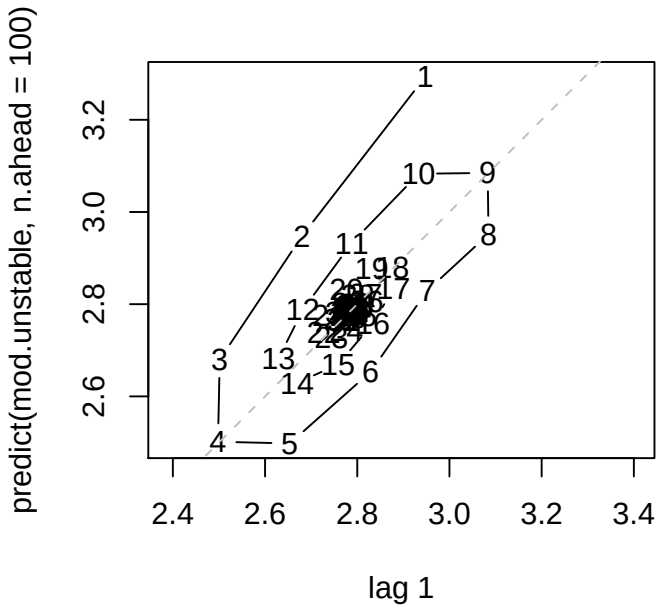
```
mod.point <- setar(x, m = 10, mL = 3, mH = 10, thDelay = 0, th = 3.12)
lag.plot(predict(mod.point, n.ahead = 100))
```



Unstable orbit:

R code

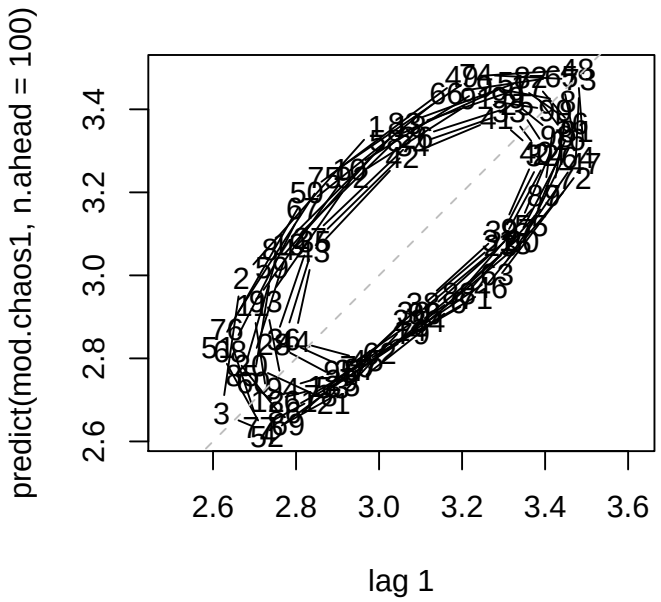
```
mod.unstable <- setar(x, m = 9, mL = 9, mH = 6, thDelay = 4,
+     th = 2.61)
lag.plot(predict(mod.unstable, n.ahead = 100))
```



Possibly chaotic systems:

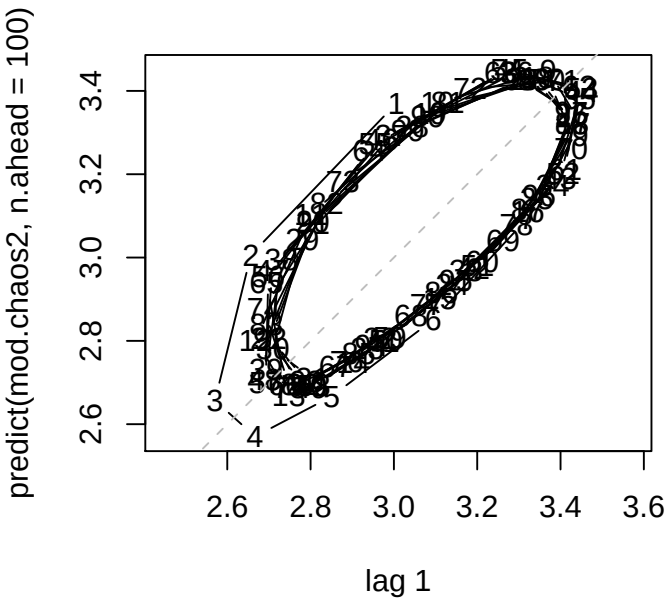
R code

```
mod.chaos1 <- setar(x, m = 5, mL = 5, mH = 3, thDelay = 1, th = 2.78)
lag.plot(predict(mod.chaos1, n.ahead = 100))
```



R code

```
mod.chaos2 <- setar(x, m = 5, mL = 5, mH = 3, thDelay = 1, th = 2.95)
lag.plot(predict(mod.chaos2, n.ahead = 100))
```



For a given fitted model, we can try estimating the maximal Lyapunov exponent with the Kantz algorithm using the `lyap_k` function in the `tseriesChaos` package˜[8, 6]. This function takes as input an observed time series, so we can procede as follows:

1. generate N observations from the model
2. add a little observational noise (otherwise the Kantz algorithm will fail)
3. apply the `lyap_k` function to the generated time series

Follows the R code for analysing the selected SETAR(2; 2,2) model of the previous paragraph and the possibly chaotic SETAR(2; 5,3) just seen above.

R code

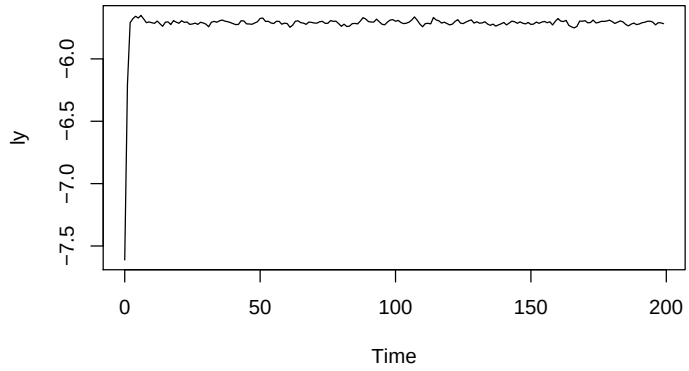
```
N <- 1000
x.new <- predict(mod[["setar"]], n.ahead = N)
x.new <- x.new + rnorm(N, sd = sd(x.new)/100)
ly <- lyap_k(x.new, m = 2, d = 1, t = 1, k = 2, ref = 750, s = 200,
+     eps = sd(x.new)/10)
```

output

```
Finding nearests
Keeping  741  reference points
Following points
```

R code

```
plot(ly)
```



There is no scaling region, so the maximal Lyapunov exponent can assumed to be ≤ 0.

R code

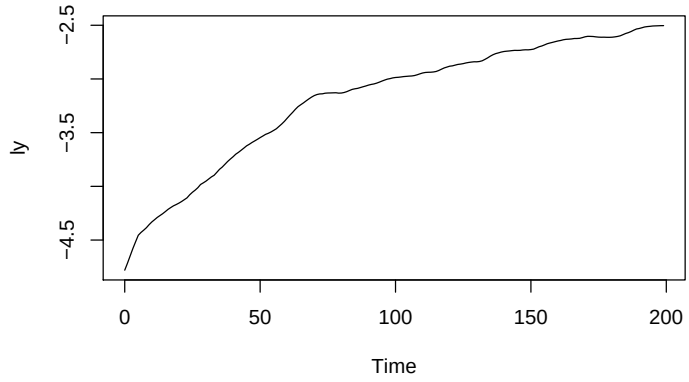
```
x.new <- predict(mod.chaos2, n.ahead = N)
x.new <- x.new + rnorm(N, sd = sd(x.new)/100)
ly <- lyap_k(x.new, m = 5, d = 1, t = 1, k = 2, ref = 750, s = 200,
+     eps = sd(x.new)/10)
```

output

```
Finding nearests
Keeping  732  reference points
Following points
```

R code

```
plot(ly)
```



Here there is a scaling region. The final λ estimate for this time series is the slope of the plotted curve in that region:

R code

```
lyap(ly, start = 6, end = 70)
```

output

```
(Intercept)      lambda
-4.53873652  0.01979557
```

At this point a natural question can be: *why not use directly the original time series as input to* `lyap_k` *instead of model-generated observations*? The answer here is that we have a too short time series for succesfully applying the Kantz algorithm, so a preliminary modelling for generating more observations is necessary.

6 Annex A: Implementation details

This section is devoted to document some points in the the implementation and is intended rather for developers. Users will hopefully not need it.

6.1 nlar.struct

Until Version 0.6 the building of data was organized as follows:

- `setar`, `star` and others all called `nlar.struct`
- `nlar.struct` calls `embedd` from package `tseriesChaos`, and stores it, returning xx, yy, m and d after some checks.

```
xxyy <- embedd(x, lags=c((0:(m-1))*(-d), steps) )

  extend(list(), "nlar.struct",
         x=x, m=m, d=d, steps=steps, series=series,
         xx=xxyy[,1:m,drop=FALSE], yy=xxyy[,m+1], n.used=length(x))
```

- But to use xx and yy the values are recomputed with new function `getXX(str)` used on str instead of using the returned value `str$xx`. It has been implemented as a class:

```
 getXXYY <- function(obj, ...) UseMethod("getXXYY")

getXXYY.nlar.struct <- function(obj, ...) {
x <- obj$x
m <- obj$m
d <- obj$d
steps <- obj$steps
embedd(x, lags=c((0:(m-1))*(-d), steps) )
}

getXX <- function(obj, ...)
getXXYY(obj,...)[ , 1:obj$m , drop=FALSE]

getYY <- function(obj, ...)
getXXYY(obj, ...)[ , obj$m+1]
```

Extension to MTAR models and ADF specification Extension to MTAR and ADF required use of both levels and lags in the same specification. That means, one needs also to obtain the differenced series. This will affect the end sample size.

In the usual case, end sample size t with be equal to $t = T - m$. It would be different (T-m-thDelay) with $thDelay > m$ but it is excluded. In some cases, the MTAR and ADF specification will lead to reduce the size of the end sample with 1 less observation, if

- MTAR: the lag of the transition delay in the MTAR is: $thDelay == (m - 1)$

- ADF and diff model: allways

To do this, some workarounds were needed, that are for now rather unsatisfactory. Similar functions as `getXX` and `getYY` have been created for the lags, see:

```
 getdXXYY <- function(obj, ...) UseMethod("getdXXYY")

getdXXYY.nlar.struct <- function(obj,same.dim=FALSE, ...) {
x <- obj$x
m<-if(same.dim) obj$m-1 else obj$m
d <- obj$d
steps <- obj$steps
embedd(x, lags=c((0:(m-1))*(-d), steps) )
}

getdXX <- function(obj, ...)
diff(getdXXYY(obj,...))[ , 1:obj$m , drop=FALSE]

getdYY <- function(obj, ...)
diff(getdXXYY(obj, ...))[ , obj$m+1]

getdX1 <- function(obj, ...)
getdXXYY(obj,...)[ -1, 1, drop=FALSE]
```

But when there is one less observation (see cases above), problems arise. So setar looks like:

```
   SeqmaxTh<-seq_len(maxTh+1)

if(model=="TAR"){
      if(type =="level")
z <- getXX(str)[,SeqmaxTh]
      else
z<-getXX(str)[-1,SeqmaxTh]
    }
    else{
      if(max(thDelay)==m-1){
if(type =="level"){
  z<-getdXX(str)[, SeqmaxTh]
  xx<-xx[-1,, drop=FALSE]
```

```
  yy<-yy[-1]
  str$xx<-xx
  str$yy<-yy
}
else
  z<-getdXX(str)[, SeqmaxTh]
      }
      else{
if(type =="level")
  z<-getdXX(str,same.dim=TRUE)[,SeqmaxTh]
else
  z<-getdXX(str)[,SeqmaxTh]
      }
    }
```

The biggest problem is that sometimes the xx vector need to be cut...

```
  z<-getXX(str)[-1,SeqmaxTh]
```

And so there are currently some bugs arising:

R code

```
plot(setar(lynx, m = 1, model = "MTAR"))
plot(setar(lynx, m = 3, type = "ADF"))
```

Adding arguments ML, MM Args `ML`, `MH` (and `MM` when `nthresh`= 2) have been added to allow to have holes in the lag structure. They are surely conflicting with arg `d`, that is, if arg `d` is not 1.

References

[1] M.˜Casdagli, *Chaos and deterministic versus stochastic nonlinear modelling*, J. Roy. Stat. Soc. 54, 303 **54** (1991).

[2] K.˜Chan and H.˜Tong, *Chaos: A Statistical Perspective*, Springer-Verlag, 2001.

[3] W.˜Enders and C.W.J. Granger, *Unit-root tests and asymmetric adjustment with an example using the term structure of interest rates.*, Journal of Business and Economic Statistics **16** (1998), no.˜3, 304–311.

[4] Philip˜Hans Franses and Dick van Dijk, *Non-linear time series models in empirical finance*, Cambridge University Press, 2000.

[5] J.˜Hamilton, *Time series analysis*, Princeton University Press, 1994.

[6] R.˜Hegger, H.˜Kantz, and T.˜Schreiber, *Practical implementation of nonlinear time series methods: The TISEAN package*, CHAOS **9** (1999), 413–435.

[7] Sebastiano Manzan, *Essays in nonlinear economic dynamics*, Thela Thesis, 2003.

[8] Antonio Fabio˜Di Narzo, *tseriesChaos: Analysis of nonlinear time series*, 2005, R package version 0.1-5.

[9] H.˜Tong, *Non-Linear Time Series: A Dynamical Systems Approach*, Oxford University Press, 1990.

[10] S.N. Wood, *Stable and efficient multiple smoothing parameter estimation for generalized additive models*, Journal of the American Statistical Association **99** (2004), 673–686.