Utility functions

1. Listing:

Commonly-used

```
fDrop[list_,k1_,k2_]
fTranspose[list_]
fRound[number_,dig_]
fShow[plot__, options___]
fNShow[plot__]
fListPlot[series_,options___]
fPartition[list_,sublength_]
fAugment[matrices__]
fMatrixPlots[matrix_,options_:{{},{},{}}]
fArgMinTensor[heap_,arg_List:{},llpos_Integer:0,func_Symbol:Min,rule_Rule:(AllSolutions→False)]
fAggregationOperator[list_,type_,prob_:0.5]
```

Diagnostic

```
fCorPlot[series_,k_]
fABHQIC[series,pmax]
fVABHQIC[series,pmax]
fELDEstimate[pDepData_,pIndepDataLagged_,pmax_,const_Integer:1,pIndepDataCurrent_List:{},options___]
fVABHQIC[pDepData_,pIndepDataLagged_,pmax_,const_Integer:1,pIndepDataCurrent_List:{}]
fSpectrumToPeriod[spec_,k_]
fGreatestPeriods[series_,k_]
fSpectrumPlot[series_,periods_,plotoptions__:{}]
fPeriodogram[series_,intervalP_,plotoptions___]
fChiSquarePValue[testst_,dof_]
fPortmanteauTest[residuals_,lag_]
fDurbinWatsonStatistic[res_?VectorQ]
```

Linear regression design

```
fSDummy[S_,n_]
fTrigVar[periods_,n_]
fΦ[from_,to_,mX_,Y_]
fDeterministicsRemoval[series_,c_,lt_,per_:{}]
f1stepForecast[tmin_,from_,to_,mX_,Y_]
```

Regime-switching

```
fTsayLTest[dMo_, p_, dEx_, q_, dTh_, d_,fAggF_:Last]
fLMtypeLTest[dMo_,dEx_,order_,dTh_,delay_,fAggF_:Last,type_Integer:1]
fLTestReport[{teststat_, dof_}, siglevel_]
fExplainCRSOrder[order_]
fResidualsTest[dMo_,dEx_,order_,dTh_,par_,Φ_,dRes_,orderRes_,startDelay_Integer:1,AggFu_:Last]
fThresholdLimits[z_,frac_]
fThresholdRange[dTh_,frac_,ndp_]
fThresholdRange[dTh_,frac_,andp_,rounding_]
fConditionalRegimeSwitching[dMo_,dEx_,order_,dTh_,regimes_,parameters_, specifications_:{110}]
```

Forecast processing

```
fUnNestAndCollectOutput[output_]
fMeanXPredictionErrorHitparade[ferrors_,g_]
fDieboldMArianoHitparade[ferrors_,horizon_,siglevel_,g_]
fCompareForecast[ferrors_,horizon_,siglevel_]....Modified Diebold-Mariano test
fPrintForecastErrors[a_,F_,k_]
fPlotForecast[orig_,ferrors_,determ_:0]
fPlotTheBestForecast[stseries_,outputfCRS_,detseries_List:{},lossfu_Function:(Dot[#,#]&),ratio_Real:1.1]
```

2. Definitions:

Commonly-used

```
fDrop[list_,k1_,k2_]
fTranspose[list_]
fRound[number_,dig_]
fShow[plot__, options___]
fNShow[plot__]
fListPlot[series_,options___]
fPartition[list_,sublength_]
fAugment[matrices__]
fMatrixPlots[matrix_,options_:{{},{},{}}]
fArgMinTensor[heap_,arg_List:{},llpos_Integer:0,func_Symbol:Min,rule_Rule:(AllSolutions→False)]
fAggregationOperator[list_,type_,prob_:0.5]
```

fDrop drops *k1* elements from the begining and *k2* from the end of *list*

```
fDrop[list_, k1_, k2_] := Drop[Drop[list, k1], -k2];
```

fDropTake[{1,2,3,4,5},2]→{{3,4,5},{1,2}}
fDropTakeNA[{{1,2,3,4,5},{11,12,13,14,15}},2]→{{{3,4,5},{13,14,15}},{{1,2},{11,12}}}

```
fDropTake[list_, k_] := {Drop[list, k], Take[list, k]};
fToNestedArray[list_] := If[ArrayDepth[list] == 1, {list}, list];
fDropTakeNA[list_, k_] := Transpose[{Drop[#, k], Take[#, k]} & /@ fToNestedArray[list]];
fToArray[x_] := If[ArrayQ[x], x, {x}];
```

fTranspose makes transposition even of one-dimensional matrix
{1,2,3} → {{1},{2},{3}}
{{1,2,3},{a,b,c}} → {{1,a},{2,b},{3,c}}

```
fTranspose[list_] := If[ArrayDepth[list] == 1, Transpose[{list}], Transpose[list]];
```

fRound returns the *number* rounded to the *dig* digits behind the floating point. The function is Listable.
fRound[123.4567, 2] → 123.46
fRound[{123.4567, 543.210},-1] → {120., 540.}

```
fRound[number_, dig_] := N[Round[number * (10^dig)] / (10^dig)];
```

fShow causes visibility of graphic objects, that are set to DisplayFunction→Identity. fNShow, on the contrary, sets this option to prevent visibility. These functions come useful when combining several graphic objects into fewer ones.

```
fShow[plot__, options___] := Show[plot, DisplayFunction :> $DisplayFunction, options];
fNShow[plot__] := Show[plot, DisplayFunction :> Identity];
```

fListPlot plots (lists in) *series* side by side with optional *options* common for all ListPlot objects.

```
fListPlot[series_, options___] :=
  Show[GraphicsArray[ListPlot[#, options, DisplayFunction → Identity] & /@ series], DisplayFunction → $DisplayFunction];
```

fPartition splits the *list* into (nonoverlaping) sublists of lengths defined in *sublength*. Following constraint on argument must be kept: Plus@@sublength ≤ Length[list],
Example: fPartition[{1,2,3,4,5,6,7},{2,3,1}] → {{1,2},{3,4,5},{6}}

```
fPartition[list_, sublength_] := Module[{i, b1 = {}, b2 = 1},
   For[i = 1, i ≤ Length[sublength], i++,
    b1 = Append[b1, Take[list, {b2, (b2 = b2 + sublength[[i]]) - 1}]];
   ];
   b1];
```

fAugment puts the sequence of matrices side by side. They must be of the same length (number of rows). Name adopted from Mathcad.

```
fAugment[matrices__] := (Thread[f[matrices]] /. f → Join);

(* ********************** *)
```

fMatrixPlots displays a *matrix* in 3 various `ListPlot`s side by side. *Options* add or respecify appearance parameters for corresponding graphic function, for example fMatrixPlots[matrix,{{Mesh→True},{},{}}] change mesh display to the first plot. Default options are {{Mesh→False},{},{ViewPoint→{-1.3,-2.4,1.5}}.

```
fMatrixPlots[matrix_, options_: {{}, {}, {}}] := fShow[GraphicsArray[{
       ListDensityPlot[matrix, DisplayFunction -> Identity, Sequence @@ #1, Mesh → False],
       ListContourPlot[matrix, DisplayFunction -> Identity, Sequence @@ #2],
       ListPlot3D[matrix, DisplayFunction → Identity, Sequence @@ #3, ViewPoint → {-1.3, -2.4, 1.5}]
      } & @@ options
   ]];
```

fArgMinTensor returns either position of minimum element in *heap* tensor, or corresponding values in *arg* if included.
llpos - optional argument, needed if the *heap* contains (on its lowest level) a list of heterogeneous elements. *llpos* then specifies which element in the list is the one of particular interest,
func - optional argument, specifies what function is applied to search the tensor; by default it is `Min`,
*Examples: fArgMinTensor[{{1,2,5,3},{4,1,0,2}}] → {2,3} ,
fArgMinTensor[{{1,2,5,3},{4,1,0,2}}, {{a1,a2},{b1,b2,b3,b4}}] → {a2,b3} ,
fArgMinTensor[{{1,2,5,3},{4,1,0,2}}, {{a1,a2}},4,Max] → {a1}.

*Note: If more than one solution is found and *AllSolutions→True* is included in arguments field, the function *fArgMinTensor* returns all solutions in a list. Otherwise it prints a warning a returns the first solution found.

```
fArgMinTensor[heap_, arg_List: {}, llpos_Integer: 0, func_Symbol: Min, rule_Rule: (AllSolutions → False)] :=
  Module[{aheap, apos, alength},
   aheap = If[llpos == 0,
     heap,
     Map[Flatten, Take[heap, Sequence @@ Table[All, {ArrayDepth[heap] - 1}], {llpos}], {-3}]];
   apos = If[arg == {}, #, MapThread[Part, {arg, #}] & /@ Position[aheap, func[aheap]];
   If[AllSolutions /. rule,
    apos,
    If[Length[apos] > 1,
     Print["Warning: ", Length[apos], " solutions found. Add \"AllSolutions→True\" to see them all."],]; apos[[1]] ]
  ];
```

fAggregationOperator is a set of the most frequent aggregation operators. The specific type is chosen by setting *type* to certain integer value, parameter *prob* serves in case of Sierpinsky carpet (type 7). Besides commonly understood operators as Last, Max, Min and Mean, there can be found other OWA operators, generally expressed as $A(a_1, a_2, \ldots a_d) = \sum_{i=1}^{d} w_i\, a_i$ where

type:	notation:	weights:
4	W_2	$w_i = (\frac{d-i+1}{d})^2 - (\frac{d-i}{d})^2$
5	W_3	$w_i = (\frac{d-i+1}{d})^3 - (\frac{d-i}{d})^3$
6	$W_{\text{Fibonacci}}$	$w_i = f_{d-i+1} / (f_{d+2} - 1)$ where f is Fibonnaci number ($f_i = f_{i-1} + f_{i-2}$ with $f_1 = f_2 = 1$)
7	$W_{\text{Sierpinski carpet}}$	$w_i = p\,(1-p)^{i-1}$ for $i < d$, $w_d = (1-p)^{d-1}$, p denotes initial probability

.....

```
fAggregationOperator[list_, type_, prob_: 0.5] := Module[{ak, av, aw2, aw3, awS, awF},
  ak = Length[list];
  av[i_, exp_] := (i / ak)^exp - ((i - 1) / ak)^exp;
  aw2 = Table[av[i, 2], {i, ak, 1, -1}];
  aw3 = Table[av[i, 3], {i, ak, 1, -1}];
  awF = Table[Fibonacci[i] / (Fibonacci[ak + 2] - 1), {i, ak, 1, -1}] // N;
  awS = Table[If[i < ak, prob (1 - prob)^(i - 1), (1 - prob)^(i - 1)], {i, ak}] // N;
  Switch[type,
   0, Last[list],
   1, Max[list],
   2, Min[list],
   3, Mean[list],
   4, Sum[aw2[[i]] * list[[i]], {i, ak}],
   5, Sum[aw3[[i]] * list[[i]], {i, ak}],
   6, Sum[awF[[i]] * list[[i]], {i, ak}],
   7, Sum[awS[[i]] * list[[i]], {i, ak}]
  ]
 ]
```

Diagnostic

```
fCorPlot[series_,k_]
fABHQIC[series,pmax]
fVABHQIC[series,pmax]
fELDEstimate[pDepData_,pIndepDataLagged_,pmax_,const_Integer:1,pIndepDataCurrent_List:{},options___]
fVABHQIC[pDepData_,pIndepDataLagged_,pmax_,const_Integer:1,pIndepDataCurrent_List:{}]
fSpectrumToPeriod[spec_,k_]
fGreatestPeriods[series_,k_]
fSpectrumPlot[series_,periods_,plotoptions__:{}]
fPeriodogram[series_,intervalP_,plotoptions___]
fChiSquarePValue[teststat_,dof_]
fPortmanteauTest[residuals_,lag_]
fDurbinWatsonStatistic[res_?VectorQ]
```

fCorPlot returns correlation function plotted for every variable included in *series* (side by side) up to lag *k*.
Requirements: TimeSeries (and affiliated "userfunc.m") package loaded.

```
fCorPlot[series_, k_] := Module[{n, cor},
   n = Length[series[[1]]];
   cor = CorrelationFunction[#, k] & /@ series;
   Show[GraphicsArray[ myplotcorr1[#, 2. / Sqrt[n], DisplayFunction → Identity] & /@ cor], DisplayFunction → $DisplayFunction]
  ];
```

fABHQIC returns GraphicsArray object containing three information criteria in one plot. In case of more time series included in *series*, the plots are given side by side. Parameter *pmax* stands for maximal order of AR(p).
Requirements: TimeSeries pack loaded.

```
fABHQIC[series_, pmax_] := Module[{seri, n, fAIC, fBIC, fHQIC, var},
  seri = If[VectorQ[series] == True, {series}, series];
  n = Length[seri[[1]]];
  fAIC[lvar_, p_] := Log[lvar[[p]] + 2 * (1 + p) / n;
  fBIC[lvar_, p_] := Log[lvar[[p]] + Log[n] * (1 + p) / n;
  fHQIC[lvar_, p_] := Log[lvar[[p]] + 2 * Log[Log[n]] * (1 + p) / n;
  var = #[[-1]] & /@ LevinsonDurbinEstimate[#, pmax] & /@ seri;
  Show[GraphicsArray[Map[Show, Transpose[
```

```
      Map[
       ListPlot[#, DisplayFunction → Identity] &,
       Outer[Table[#1[#2, p], {p, pmax}] &, {fAIC, fBIC, fHQIC}, var, 1],
       {2}]
     ], {1}]], DisplayFunction :→ $DisplayFunction]
  ]
```

fVABHQIC returns Graphics object containing three information criteria in one plot. Parameter *pmax* stands for maximal order of VAR(p).
Requirements: TimeSeries pack loaded.

```
fVABHQIC[series_, pmax_] := Module[{seri, m, n, fVAIC, fVBIC, fVHQIC, var, det},
  seri = If[VectorQ[series] == True, {series}, series];
  {m, n} = Dimensions[seri];
  fVAIC[ldet_, p_] := Log[ldet[[p]]] + 2 * (m + m^2 * p) / n;
  fVBIC[ldet_, p_] := Log[ldet[[p]]] + Log[n] * (m + m^2 * p) / n;
  fVHQIC[ldet_, p_] := Log[ldet[[p]]] + 2 * Log[Log[n]] * (m + m^2 * p) / n;
  var = #[[-1]] & /@ LevinsonDurbinEstimate[Transpose[seri], pmax];
  det = Table[Det[var[[i]]], {i, pmax}];
  Show[
   ListPlot[#, DisplayFunction → Identity] & /@ Transpose[Table[{fVAIC[det, p], fVBIC[det, p], fVHQIC[det, p] }, {p, pmax}]],
   DisplayFunction :→ $DisplayFunction]
  ]
```

Function fELDEestimate is an analogy and extention to LevinsonDurbinEstimate procedure using classical $y_t = \mu + \phi_1 x_{t-1} + ... + \phi_p x_{t-p} + \phi_0 z_t$ ordinary regression method (NOT the Yule-Walker method), where the variable y_t corresponds to *pDepData*, variable x_t to *pIndepDataLagged* and z_t to *pIndepDataCurrent* (this is optional). The constant term μ is included by default. If not desired, set *const* to zero. Variables x_t and z_t can be a list of variables in the form, e.g., {{x_{11}, x_{12}, ... x_{1n}}, {x_{21}, x_{22}, ... x_{2n}}, ...}.
The *options* and *output* is inherited from system build-in function DesignedRegress. The function results into *pmax*-dimensional list.

```
fELDEstimate[pDepData_, pIndepDataLagged_, pmax_, const_Integer: 1, pIndepDataCurrent_List: {}, options___] := Module[
   {con = If[const == 0, {}, const], tlag = fTranspose[pIndepDataLagged],
    tcur = If[Flatten[#] == {}, Table[{}, {Length[pDepData]}], fTranspose[#]] &[pIndepDataCurrent]}, Table[DesignedRegress[
     Table[Flatten[{con, tlag[[t - Range[p]]], tcur[[t]]}], {t, p + 1, Length[pDepData]}], Drop[pDepData, p], options], {p, 1, pmax}]
  ];
```

fABHQIC returns GraphicsArray object containing three information criteria in one plot. In case of more time series included in *pDepData*, the plots are given side by side. Arguments are inherited from function fELDEstimate (see details thereby). If *pDepData* contains several variables that are to be modeled with the same variables (either in *pIndepDataLagged* or *pIndepDataCurrent* or both), these need not be listed multiple times.
Requirements: fELDEstimate (and Statistics`LinearRegression` pack loaded)

```
fABHQIC[pDepData_, pIndepDataLagged_, pmax_, const_Integer: 1, pIndepDataCurrent_List: {}] :=
 Module[{dep, indlag, indcur, n, fAIC, fBIC, fHQIC, var},
  dep = If[VectorQ[pDepData] == True, {pDepData}, pDepData];
  {indlag, indcur} = Nest[List, #, Clip[1 + (Depth[dep] - Depth[#]), {0, 10}]] & /@ {pIndepDataLagged, pIndepDataCurrent};
  {indlag, indcur} =
   If[Length[#] ≠ Length[dep] ∧ Length[dep] ≠ 1 ∧ Depth[#] == 4, Table[#[[1]], {Length[dep]}], # ] & /@ {indlag, indcur};
  n = Length[dep[[1]]];
  fAIC[lvar_, p_] := Log[lvar[[p]]] + 2 * (1 + p) / n;
  fBIC[lvar_, p_] := Log[lvar[[p]]] + Log[n] * (1 + p) / n;
  fHQIC[lvar_, p_] := Log[lvar[[p]]] + 2 * Log[Log[n]] * (1 + p) / n;
  var = MapThread[
    EstimatedVariance /. fELDEstimate[#1, #2, pmax, const, #3, RegressionReport → EstimatedVariance] &, {dep, indlag, indcur}];
  Show[GraphicsArray[Map[Show, Transpose[
      Map[
       ListPlot[#, DisplayFunction → Identity] &,
       Outer[Table[#1[#2, p], {p, pmax}] &, {fAIC, fBIC, fHQIC}, var, 1],
       {2}]
     ], {1}]], DisplayFunction :→ $DisplayFunction]
  ]
```

fSpectrumToPeriod returns periods corresponding to *k* largest spectral densities in *spec*.
fGreatestPeriods returns the *k* above periods for every time series in *series*.
Requirements:TimeSeries package loaded.

```
fSpectrumToPeriod[spec_, k_] := Module[{half = Take[spec, Floor[Length[spec] / 2]]},
   N[Flatten[Length[spec] / (Position[half, _?(# ≥ Sort[half][[-k]] &)] - 1)]]
  ];
fGreatestPeriods[series_, k_] := fSpectrumToPeriod[#, k] & /@ (Spectrum /@ series)
```

fSpectrumPlot returns power spectrum plot for every variable included in *series* (side by side).
periods - list of periods T, for which (and also for their halves) the tics are drawn on x-axis in the form 2π / T (4π / T),
o*ptions* - redefine appearance of the plot.
Requirements:TimeSeries (and affiliated "userfunc.m") package loaded.

```
fSpectrumPlot[series_, periods_, plotoptions__: {}] :=
  Show[GraphicsArray[plotspectrum[#, plotoptions, Ticks → {Flatten[{2 * π / #, 2 * (2 * π / #)} & /@ periods], Automatic}] & /@
     (Spectrum /@ series)], DisplayFunction → $DisplayFunction];
```

fPeriodogram returns continuous plot of spectral densities corresponding to periods (given as interval), for every time series inlcuded in *series*.

```
fPeriodogram[series_, intervalP_, plotoptions___] := Module[{seri, n, p},
  seri = If[VectorQ[series] == True, {series}, series];
  n = Length[seri[[1]]];
  Show[GraphicsArray[f[1/(2 π n) ((Sum[#[[t]] Cos[2 * π/p t], {t, 1, n}])^2 + (Sum[#[[t]] Sin[2 * π/p t], {t, 1, n}])^2),
       Prepend[intervalP, p], DisplayFunction → Identity, plotoptions] & /@ seri] /. f → Plot, DisplayFunction → $DisplayFunction]]
```

fChiSquarePValue returns right-sided χ^2(dof) p-value for the *test st*atistic.

```
fChiSquarePValue[testst_, dof_] := 1 - CDF[ChiSquareDistribution[dof], testst];
```

fPortmanteauTest performes (multivariate, Ljung-Box) portmanteau test for serial independence in *k*×*n* residuals with respect to maximal lag and returns Q(*lag*) test statistic with ($k^2 * lag$) degrees of freedom

```
fPortmanteauTest[residuals_, lag_] := Module[{fLaggedDataMatrix, res, an, an1, aΓ, aμ, aΓ0inv, aLB},
  (* fLaggedDataMatrix returns Table[{Y_t, Y_{t-1}, ... Y_{t-lmax}}, {t, lmax+1, n}]//Transpose if cut = False,
   otherwise it cuts to (lmax+1)×(n-lmax) matrix, where n is the legth of Y *)
  fLaggedDataMatrix[Y_, lmax_, cut_: False] :=
   If[cut, Take[#, lmax - Length[Y]] & /@ (Drop[Y, -#] & /@ Range[0, lmax]), Drop[Y, -#] & /@ Range[0, lmax]];
  res = fTranspose[residuals];
  an = Length[res];
  aΓ = fLaggedDataMatrix[res, lag];
  aμ = (aμ = Mean[res]; Table[aμ, {an}]);
  aΓ = (an1 = Length[#]; Transpose[Take[aΓ[[1]] - aμ, -an1]].(# - Take[aμ, -an1]) / (an1 - 1)) & /@ aΓ;
  aΓ0inv = Inverse[aΓ[[1]]];
  aLB = an^2 Sum[Tr[Transpose[aΓ[[i + 1]]].aΓ0inv.aΓ[[i + 1]].aΓ0inv] / (an - i), {i, 1, lag}];
  {aLB, lag * Length[res // First]^2}
 ]
```

fDurbinWatsonStatistic returns D-W statistic for *res* (only univariate). It is ranged between 0 and 4 (perfect positive and perfect negative autocorelation), 2 means no correlations. Critical values are tabulated only.

```
fDurbinWatsonStatistic[res_?VectorQ] := Module[{be, bd}, be = Drop[res, 1]; bd = be - Drop[res, -1]; (bd.bd) / (be.be)];
```

Linear regression design

```
fSDummy[S_,n_]
fTrigVar[periods_,n_]
fΦ[from_,to_,mX_,Y_]
fDeterministicsRemoval[series_,c_,lt_,per_:{}]
f1stepForecast[tmin_,from_,to_,mX_,Y_]
```

fSDummy returns S×n matrix, where the s-th row represents seasonal dummy variable (1 if t=s and 0 otherwise) corresponding to season s,
/S - number of seasons, /n - number of time points

```
fSDummy[S_, n_] := Table[If[s == FractionalPart[(t - 1) / S] S + 1, 1, 0], {s, 1, S}, {t, 1, n}];
```

fTrigVar returns (n×2 Length[periods]) matrix of trigonometric polynomial elements to be used as determinstic variable in a regression.
*Symbolic example: `fTrigVar[{T1,T2},n]`→
`{{Cos[2π/T1], Sin[2π/T1], Cos[2π/T2], Sin[2π/T2]}, ... {Cos[2π/T1 n], Sin[2π/T1 n], Cos[2π/T2 n], Sin[2π/T2 n]}}`

```
fTrigVar[periods_, n_] := Module[{per},
   per = If[NumericQ[periods], {periods}, periods];
   Table[Flatten[{Cos[2 π t / #], Sin[2 π t / #]} & /@ per], {t, 1, n}]
  ];
```

fΦ uses OLS to estimate parameter matrix through specified range of time points (*from*, *to*, 1) given design matrix *mX* and response *Y*.

```
fΦ[from_, to_, mX_, Y_] := Inverse[Sum[Outer[Times, mX[[t]], mX[[t]]], {t, from, to}]].Sum[Outer[Times, mX[[t]], fTranspose[Y][[t]]], {t, from, to}];
```

fDeterministicsRemoval returns residuals from OLS regression of *series* on polynomial terms (x^0,x^1,... exponents are listed in *powers*) and trigonometric polynomial terms (Cos[$\frac{2\pi}{T}$], Sin[$\frac{2\pi}{T}$] ,.... with *T* listed in *periods*).

```
fDeterministicsRemoval[series_, powers_, periods_: {}] := Module[{an, at, apolyn, atrig, aX},
  an = Length[series];
  apolyn = at^powers;
  atrig = Flatten[{Cos[2 π at / #], Sin[2 π at / #]} & /@ If[Length[periods] == 0, {periods}, periods]];
  aX = N[Join[apolyn, atrig]]; aX = Table[aX, {at, an}];
  series - Flatten[aX.fΦ[1, an, aX, series]]
 ]
```

f1stepForecast returns 1-step-ahead forecasts for time points t∈{*from*,... *to*}. Parameters of the model are being reestimated every step from the data (design matrix *mX* and response *Y*) at time {*tmin*, ... t-1}.

```
f1stepForecast[tmin_, from_, to_, mX_, Y_] := Module[{bt, bForecast},
   bForecast = {};
   For[bt = from, bt ≤ to, bt++,
    bForecast = Append[ bForecast, mX[[bt]].fΦ[tmin, bt – 1, mX, Y] ]];
   bForecast
  ];
```

Regime-switching

```
fTsayLTest[dMo_, p_, dEx_, q_, dTh_, d_,fAggF_:Last]
fLMtypeLTest[dMo_,dEx_,order_,dTh_,delay_,fAggF_:Last,type_Integer:1]
fChiSquarePValue[testst_,dof_]
fLTestReport[{teststat_, dof_}, siglevel_]
fExplainCRSOrder[order_]
fResidualsTest[dMo_,dEx_,order_,dTh_,par_,Φ_,dRes_,orderRes_,startDelay_Integer:1,AggFu_:Last]
fPortmanteauTest[residuals_,lag_]
fThresholdLimits[z_,frac_]
fThresholdRange[dTh_,frac_,ndp_]
fThresholdRange[dTh_,frac_,andp_,rounding_]
fConditionalRegimeSwitching[dMo_,dEx_,order_,dTh_,regimes_,parameters_, specifications_:{110}]
```

■ Linearity test

fTsayLTest uses procedure of [Tsay1998] test of linearity (versus treshold regime-switching nonlinearity).

- Input: *dMo* = {y1,y2,...y*k*} - modeled data; *dEx* = {x1,x2,...x*v*} - exogenous data; *p* - order of AR model; *q* - order of exogenous dependency regression; *dTh* - threshold variable (single list); *d* - delay, fAggF - aggregation function.
- Output: {test statistic, degrees of freedom}
- Remark: If there is no exogenous variable to be included, then *dEx* = {}.

```
fTsayLTest[dMo_, p_, dEx_, q_, dTh_, d_, fAggF_: Last] :=
  Module[{an, ah, am0, ak, av, aTh, faDesignMatrix, aX, aY, aΦ, aV, ae, aη, aΨ, aw, aS0, aS1, aC, adf},
   an = Dimensions[dMo][[2]];
   ah = Max[p, q, d];
   am0 = Round[3 √an ];
   {ak, av} = Length /@ {dMo, dEx};
   aTh = Table[fAggF[dTh[[Range[t – 1, t – d, –1]]]], {t, ah + 1, an}];
   ai = Ordering[aTh] + ah;
   faDesignMatrix[endo_, pp_, exo_, qq_] := Module[{fbXen, fbXex},
     If[pp == 0, fbXen[i_] := {}, fbXen[i_] := endo[[ai[[i]] – Range[pp]]]];
     If[qq == 0, fbXex[i_] := {}, fbXex[i_] := exo[[ai[[i]] – Range[qq]]]];
     Table[Flatten[{1, fbXen[i], fbXex[i]}], {i, an – ah}]
    ];
   aY = fTranspose[dMo];
   aX = faDesignMatrix[aY, p, fTranspose[dEx], q];
   aY = Table[aY[[ ai[[i]] ]], {i, an – ah}];
   aΦ = PadLeft[Table[Inverse[Transpose[aX[[Range[1, m]]]].aX[[Range[1, m]]]].
        Transpose[aX[[Range[1, m]]]].aY[[Range[1, m]]], {m, am0, an – ah}], an – ah, {0}];
   aV = PadLeft[Table[
      Inverse[Transpose[Take[aX, m]].Take[aX, m]],
      {m, am0, an – ah}], an – ah, {0}];
   ae = PadLeft[Table[ aY[[i]] – aX[[i]].aΦ[[i – 1]] , {i, am0 + 1, an – ah}], an – ah, {0}];
   aη = PadLeft[Table[ ae[[i]] / (1 + aX[[i]].aV[[i – 1]].aX[[i]])^(1/2) , {i, am0 + 1, an – ah}], an – ah, {0}];
   aΨ = Inverse[Transpose[aX[[Range[am0 + 1, an – ah]]]].aX[[Range[am0 + 1, an – ah]]]].
     Transpose[aX[[Range[am0 + 1, an – ah]]]].aη[[Range[am0 + 1, an – ah]]];
   aw = PadLeft[Table[ aη[[i]] – aX[[i]].aΨ, {i, am0 + 1, an – ah}], an – ah, {0}];
   aS0 = 1/(an – ah – am0) Sum_{i=am0+1}^{an–ah} Outer[Times, aη[[i]], aη[[i]]];
   aS1 = 1/(an – ah – am0) Sum_{i=am0+1}^{an–ah} Outer[Times, aw[[i]], aw[[i]]];
   aC = (an – ah – am0 – (ak p + av q + 1)) (Log[Det[aS0]] – Log[Det[aS1]]);
   adf = ak (ak p + av q + 1);
   {aC, adf}
  ];
```

fLMtypeLTest uses procedure for linearity test (H_0: VAR) against smooth regime-switching nonlinearity proposed by [Luukkonen,Saikkonen and Teräsvirta(1988)] and [Escribano and Jordá(1999)] further extended to multivariate case. The input is similar to that for function *fConditionalRegimeSwitching* (Current version of fLMtypeLTest is temporary, the final one is meant to be included in fConditionalRegimeSwitching).

- Input: *dMo* = {y1,y2,...y*k*} - modeled data; *dEx* = {x1,x2,...x*v*} - exogenous data; *order* = {p,q} order of AR and exogenous dependency regression; *dTh* - threshold variable (single list); *delay* - delay in threshold variable; *fAggF* - aggregation function; *type* = 1 if H_1: LogisticSTAR otherwise H_1: ExponentialSTAR
- Output: {test statistic, degrees of freedom}
- Remark: If there is no exogenous variable to be included, then *dEx* = {}.

```
fLMtypeLTest[dMo_, dEx_, order_, dTh_, delay_, fAggF_: Last, type_Integer: 1] := Module[
  {fPaddedDesignMatrix, fThresVarAggMatrix, fTranspose, fAugment, f1sidedχ2pvalue, f2sidedχ2pvalue,
   ah, an,
   aX, aY, aΦ, aϵ, aXnc, aXext, aQ, aΣ0, aΣ1, aLM3, adf},
  (* ----- subroutine functions ----- *)
```

```
(* single-regime design matrix construction function - padded to be of full length
   - k × n modeled and exogenous data
   - orderr={p,q} of auto and exogen regression
    - const = 1 if constant term is desired in regression, otherwise const = {}  *)
fPaddedDesignMatrix[modeled_, exogenous_, orderr_, const_: 1] := Module[{bTmo, bTex, bXmo, bXex, bh, bn},
      {bTmo, bTex} = If[ArrayDepth[#] == 1, Transpose[{#}], Transpose[#]] & /@ {modeled, exogenous};
      bn = Max[Length /@ {bTmo, bTex}];
      bh = Max[orderr]; {bp, bq} = orderr;
      If[bp == 0 || Length[modeled] == 0, bXmo[i_] := {}, bXmo[i_] := Table[bTmo⟦i - j⟧, {j, bp}]];
      If[bq == 0 || Length[exogenous] == 0, bXex[i_] := {}, bXex[i_] := Table[bTex⟦i - j⟧, {j, bq}]];
  PadLeft[Table[Flatten[{const, bXmo[i], bXex[i]}], {i, bh + 1, bn}], bn, {{}}]];
(* fThresVarAggMatrix returns matrix of threshold variable values
  - threshold variable q_t = AggF[{dThV_{t-1},...,dThV_{t-d}}], where AggF is aggregation function of list of arguments
   - padded with {} to make a time-point and value index to coincide
   - requires order (parent module variable)
    Example: fThresVarAggMatrix1[{1,2,3,4,5,6,7},{2,4},Last] → { {} , {{},{},1,2,3,4,5}, {} , {{},{},{},{},1,2,3} } *)
fThresVarAggMatrix[dataThresVar_, valueDelay_, nameAggF_] := Module[{bn, btab1, btab2, bi},
      bn = Length[dataThresVar];
      btab1 =
   Outer[If[Max[order, #1] < #2, nameAggF[dataThresVar⟦ Reverse[Range[#2 - #1, #2 - 1]] ⟧], {}] &, valueDelay, Range[bn]];
      btab2 = Table[{}, {Max[valueDelay]}];
      For[bi = 1, bi ≤ Length[valueDelay], bi++,
       btab2⟦valueDelay⟦bi⟧⟧ = btab1⟦bi⟧ ];
  btab2];
(* fTranspose makes transposition even of one-dimensional matrix
 {1,2,3} → {{1},{2},{3}}
  {{1,2,3},{a,b,c}} → {{1,a},{2,b},{3,c}}
  Note: The If-condition with $VersionNumber is just for demonstration purpose,
 there is no difference in performance between ArrayDepth (Mathematica from v .5 up) and Length∘Dimensions *)
If[$VersionNumber ≥ 5,
 fTranspose[list_] := If[ArrayDepth[list] == 1, Transpose[{list}], Transpose[list]],
 fTranspose[list_] := If[Length[Dimensions[list]] == 1, Transpose[{list}], Transpose[list]]
];
(* augments the arg sequence of matrices - must be of the same dimensions;(name adopted from Mathcad) *)
fAugment[arg__] := (Thread[f[arg]] /. f → Join);

(* ---- variables definition ---- *)
ah = Max[order, delay];
an = Length[dTh];

(* ---- main routine ---- *)
(* -- linear regression of y_t on X_t -- *)
aX = Drop[fPaddedDesignMatrix[dMo, dEx, order], ah];
aY = Drop[fTranspose[dMo], ah];
aΦ = Inverse[Transpose[aX].aX].Transpose[aX].aY;
aϵ = aY - aX.aΦ;
aΣ0 = Transpose[aϵ].aϵ;
(* -- linear regression of y_t on  X_t,  X_t q_t,  X_t q_t^2, X_t q_t^3 -- *)
aXnc = Drop[fPaddedDesignMatrix[dMo, dEx, order, {}], ah];
aY = Drop[fTranspose[dMo], ah];
aQ = Drop[fThresVarAggMatrix[dTh, {delay}, fAggF]⟦delay⟧, ah]; (* it works iff delay is an integer, not a list *)
aXext =
 If[type == 1, fAugment[aX, aXnc × aQ, aXnc × aQ^2, aXnc × aQ^3], fAugment[aX, aXnc × aQ, aXnc × aQ^2, aXnc × aQ^3, aXnc × aQ^4]];
aΦ = If[aΦ = Transpose[aXext].aXext; MatrixRank[aΦ] < Length[aΦ],
   Print["Warning: Singular or badly conditioned matrix for delay=",
    delay, ". PseudoInverse enforced."]; PseudoInverse[aΦ],
   Inverse[aΦ]].Transpose[aXext].aY;
aϵ = aY - aXext.aΦ;
aΣ1 = Transpose[aϵ].aϵ;
(* -- test -- *)
aLM3 = (an - ah) (Log[Det[aΣ0]] - Log[Det[aΣ1]]); (* test statistics *)
adf = If[type == 1, 3, 4] * Length[dMo] * (1 + Map[Length, {dMo, dEx}].order); (* degrees of freedom *)

{aLM3, adf}];
```

fLTestReport returns summary report of linearity test.
- Input: {*test stat*istic, *d*egrees *o*f *f*reedom}, *sig*nificance *level* (e.g., 0.95)
- Output as table: test statistic, degrees of freedom, χ^2-quantile, "rejection of alternative hypothesis of nonlinearity (True or False)", one-sided p-value
Requirements: Statistics`HypothesisTests` package loaded.

```
fLTestReport[{teststat_, dof_}, siglevel_] := Module[{quantile}, TableForm[ {
    {"t.statistic", "dof", "quantile", "linear ?", "p-value"}, Join[{teststat, dof},
     {quantile = Quantile[ChiSquareDistribution[dof], siglevel], (teststat < quantile), fChiSquarePValue[teststat, dof]}]}]]
```

■ Residuals test

fExplainCRSOrder extracts all information from *order* (as defined by f*C*onditional*R*egime*S*witching), i.e.:
{number of regimes, No. of endog. variables per regime(p.r. onward), No. of exog. variables p.r., No. of lagged values of envars per variable(p.v.) p.r., No. of (not only lagged) values of exvars p.v. p.r., maximal lag in envars p.v. p.r., maximal lag in exvars p.v. p.r., length of regressor (1+kp+lq) p.r.}
Example: fExplainCRSOrder[{{{{1,2,4}},{{0}}},{{{1,2},{1,2,3}},{}}}] → {2,{1,2},{1,0},{{3},{2,3}},{{1},{}},{{4},{2,3}},{{0},{}},{5,6}}

```
fExplainCRSOrder[order_] := Module[{am, ak, al, ap, aq, apmax, aqmax, aK},
  am = Length[order];
  {ak, al} = (Apply[f, order, {2}] /. {f → (Length[{##}] &)})[[All, #]] & /@ {1, 2};
  {ap, aq} = (Apply[f, order, {3}] /. {f → (Length[{##}] &)})[[All, #]] & /@ {1, 2};
  {apmax, aqmax} = (Apply[f, order, {3}] /. {f → Max})[[All, #]] & /@ {1, 2};
  aK = MapThread[(1 + #1 + #2) &, {Plus @@@ ap, Plus @@@ aq}];
  (*?? aK=MapThread[(1+#1.#3+#2.#4)&,{Table[1,{#}]&/@ak,Table[1,{#}]&/@al,ap,aq}] ??*)
  {am, ak, al, ap, aq, apmax, aqmax, aK}
 ]
```

fResidualsTest returns table of p-values corresponding to test for residual serial correlations (H_0: serial independence) and nonlinearity (H_0: linearity).
dMo, *dEx*, *dTh* - *k*×*n*, *l*×*n*, *n*×1 endogenous, exogenous and threshold variable data,
order,*par*,*Φ*, *dRes* - order of regression in the full form, parameters (*d*, *r*, *γ*), 2 (1+*kp*+*lq*)×*k* parameter matrix, 2×(n-Max[*p*,*q*,*d*]) residuals. Output from *fConditionalRegime-Switching* (3rd and 2nd mode).
orderRes - order of AR for residuals in the test of residual autocorrelations,
startDelay - starting lag d_0 in AggFu[{dTh$_{t-d_0}$, ..., dTh$_{t-d}$}], where *AggFu* is the name of aggregation function.

```
fResidualsTest[dMo_, dEx_, order_, dTh_, par_, Φ_, dRes_, orderRes_, startDelay_Integer: 1, AggFu_: Last] :=
 Module[{fPaddedDesignMatrix, fThresVarAggMatrix,
   aK, ad, ac, aγ, an, ah, aϵ, az, aX, adFdΦ1, adFdΦ2, adFdγ, adFdr, aϵl, aXext, aϵd, au, aΣ0, aΣ1, aLMsi, aconst, aXs123, aLMlin},
  (* ---------------------------- subroutine functions ---------------------------- *)
  (* fDMRow creates ti^th row of singleregime design matrix,
   /mo, ex - k1,k2 × n modeled and exogenous data,
   /or - order of ARX = lag specification for every variable,
   /const = 1 if constant term is desired in regression, otherwise const = {},
   Example: if a={a1,a2,a3,a4,a5,a6},b={b1,b2,b3,b4,b5,b6}
      then fDMRow[6,{a},{a,b},{{{0,1,2}},{{},{0}}}]→{1,a6,a5,a4,b6}  *)
 fDMRow[ti_, mo_, ex_, or_, const_: 1] := Module[{bXmo, bXex},
   bXmo = MapThread[Part, {mo, ti - or[[1]]}];
   bXex = MapThread[Part, {ex, ti - or[[2]]}];
   Flatten[{const, bXmo, bXex}]
 ];
 (* fThresVarAggVectorRow returns value of threshold variable corresponding to time "ti".
      - threshold variable z_t = AggF[{ThresVar_{t-Delay[[1]]},...,ThresVar_{t-Delay[[2]]}}],
  Example: fThresVarAggVector[4,{1,2,3,4,5,6,7},{1,3},Last] → 1 *)
 fThresVarAggVectorRow[ti_, dataThresVar_, rangeDelay_, nameAggF_] :=
  If[Max[rangeDelay] < ti, nameAggF[dataThresVar[[ti - Range @@ rangeDelay]]], {}];
 (* fStackAndPad function encapsulates the Table and PadLeft functions *)
 fStackAndPad[function_, from_, to_, args__] := PadLeft[Table[function[i, args], {i, from, to}], to, {{}}];
 (* ---------------------------- variables definition ------------------------------- *)
 aK = fExplainCRSOrder[order][[8]];
 If[Equal @@ order, ,
  Print["Error: This procedure is designed only for the same order of regression in both regimes. Calculation aborted."];
  Abort[];];
 {ad, ar, aγ} = If[Length[par] == 3, par, Append[par, 20]];
 {aΦ1, aΦ2} = fPartition[Φ, aK];
 an = Dimensions[dMo][[2]];
 ah = an - Dimensions[dRes][[2]];
 aϵ = PadLeft[Transpose[dRes], an];
 az = fStackAndPad[fThresVarAggVectorRow, ad + 1, an, dTh, {startDelay, ad}, AggFu];
 aX = fStackAndPad[fDMRow, ah, an, dMo, dEx, #, 1] & /@ order;
 {adFdΦ1, adFdΦ2, adFdγ, adFdr} = Transpose[Table[aexp = Exp[-aγ (az[[t]] - ar)]; aXΦ2m1 = (aX[[2, t]].aΦ2 - aX[[1, t]].aΦ1);
    {aX[[1, t]] (1 - 1/(1 + aexp)), aX[[2, t]]/(1 + aexp), aXΦ2m1 (az[[t]] - ar) aexp/(1 + aexp)^2, -aXΦ2m1 aγ aexp/(1 + aexp)^2}, {t, ah + 1, an}]];
 (* ******** test for residual corelations ******** *)
 aϵl = Table[Flatten[aϵ[[Range[t - 1, t - orderRes, -1]]]], {t, ah + orderRes + 1, an}];
 aXext = fAugment @@ Append[Drop[#, orderRes] & /@ {adFdΦ1, adFdΦ2, adFdγ, adFdr}, aϵl];
 aϵd = Drop[aϵ, orderRes + ah];
 au = aϵd - aXext.(Inverse[Transpose[aXext].aXext].Transpose[aXext].aϵd);
 aΣ0 = Transpose[aϵd].aϵd;
 aΣ1 = Transpose[au].au;
 aLMsi = (an - ah - orderRes) (Log[Det[aΣ0]] - Log[Det[aΣ1]]);
 (* ******** test for residual nonlinearity -- not for different orders in regimes!! ******** *)
 aconst = (If[MemberQ[Flatten[Transpose[aX][[-1, 1]]], az[[-1]]], {}, 1]; {}); (* 2 check *)
 aX = fStackAndPad[fDMRow, ah, an, dMo, dEx, order[[1]], aconst];
 aXs123 = Transpose[Table[{aX[[t]] az[[t]], aX[[t]] az[[t]]^2, aX[[t]] az[[t]]^3}, {t, ah + 1, an}]];
 aXext = fAugment @@ Join[{adFdΦ1, adFdΦ2, adFdγ, adFdr}, aXs123];
 aϵd = Drop[aϵ, ah];
 au = aϵd - aXext.(PseudoInverse[Transpose[aXext].aXext].Transpose[aXext].aϵd);
 aΣ0 = Transpose[aϵd].aϵd;
 aΣ1 = Transpose[au].au;
 aLMlin = (an - ah) (Log[Det[aΣ0]] - Log[Det[aΣ1]]);

 {{"test for:", "p-value:"},
   {"serial independence    ", fChiSquarePValue[aLMsi, Length[dMo] * orderRes]},
   {"linearity    ", fChiSquarePValue[aLMlin, 3 * Length[dMo] * (aK[[1]] - 1 + UnitStep[Max[aconst]])]}
  } // TableForm
];
```

■ Conditional estimation

fThresholdLimits returns {min,max} values of threshold as *frac* % percentiles of *z*

```
fThresholdLimits[z_, frac_] := Sort[z][[Round[{Length[z] * frac, Length[z] * (1 - frac)} ]]]
```

fThresholdRange(3 arguments) returns {min,max, step} format for threshold values
dTh - threshold variable,
frac - fraction, of which *dTh* is being shortened from its begining as well as from its tail,
ndp - number of discrete points desired to get from the range,
Note: the function uses rounding at the expenses of frac exactitude.

```
fThresholdRange[dTh_, frac_, ndp_] := Module[{tmp, tmp1, tmp2, tmp3},
   tmp2 = fRound[(tmp = fThresholdLimits[dTh, frac]) / (tmp1 = -Subtract @@ tmp / (ndp - 1)), 1];
   tmp3 = 0 - RealDigits[tmp2 * tmp1 - tmp][[1, 2]];
   Append[tmp2 * #, #] &[fRound[tmp1, tmp3]]
  ];
```

fThresholdRange(4 arguments) returns {min,max, step} format for threshold values,
dTh - threshold variable,
frac - fraction, of which *dTh* is being shortened from its begining as well as from its tail,
andp - approximate number of discrete points desired to get from the range,
rounding - specifies number of digits to the right of decimal point; if rounding < 0 then the Real becomes Integer ending with as many number of zeros,
Note: the function uses rounding at the expenses of andp exactitude.

```
fThresholdRange[dTh_, frac_, andp_, rounding_] :=
 Append[#, fRound[-(Subtract @@ #) / andp, rounding] ] &[ fRound[fThresholdLimits[dTh, frac], rounding] ]

(* ********************** *)
```

fConditionalRegimeSwitching solves nonlinear regime-switching models like Threshold VAR, and SmoothTransition VAR for a set of modeled and exogenous time series. The procedure is based on Ordinary Least Squares estimation conditional on fixed values of the nonlinearity-causing "parameters". These are:
- delay "d" to be used for threshold variable,
- threshold "r" to be compared with threshold variable,
- smooth transition parameter "γ" determining the speed of transition between regimes,

Description of function input:

dMo - *k1* × *n* modeled data, where *k1* is number of variables and *n* length of time-series,

dEx - *k2* × *n* exogenous data, where *k2* is number of exogenous variables,

order - order of (Auto)Regressive models, either *p* (then *q* = *p*) or {*p*, *q*}, where *p* is order of AR and *q* exogenous regression order. In order to define the lags separately for every regime, lags must be fully listed at least for the first regime (for details, refer to function *fOrder*),

dTh - threshold variable data, must be of the same length as *dMo* and *dEx*, but univariate. If threshold variable is one of the modeled, it is possible to type only the position in *dMo*.

regimes - number of regimes,

parameters - range or values of parameters values in the following order: *d*, *r*, γ. A range is considered only if constituted by 3 values: {min,max,step}. *Example: {{2},{-1,1.35, 0.1}} where *d* has only one value (2) and *r* (-1,-0.9,-0.8,...1.3), where the step is 0.1. Parameter γ is not taken into account in this example setup, so the switching between regimes is automaticaly crisp $(y_t = \phi_1 x_t I[z_t \le c] + \phi_2 x_t I[z_t > c])$. Otherwise a smooth transition function is chosen according to specifications. *Another example: {{2,4},1.25,{18,0,1000,340}} where *d* = {2,3, 4}, r={1.25} (meaningful if *regimes* = 2), and γ ={18,0,1000, 340} (values can be specified if their number is either 1, 2 or 4, 5...),

specifications - list of switches and additional parameters for accessing various model extensions, symbolicaly {spec1, spec2, spec3, ...} where,
- spec1 = ***abcd***...,

a - main purpose - output:

a = 1 - sum of squares of residuals (trace of covariance matrix if Vector(S)TAR model estimated,

a = 2 - {Φ,Σ,res}, i.e. parameter and covariance matrix, residuals

a = 3 - {AIC,BIC} = Akaike and Schwarz(Bayesian) Information Criteria,

a = 4 - $\{y_{an-bnF+1} - \hat{y}_{an-bnF+1}, \ldots, y_{an} - \hat{y}_{an}\}'$ = forecast errors (evaluated at the last *bnF* time points),

a = 5 - $\{\hat{y}_{n+1}, \hat{y}_{n+2}, \ldots, \hat{y}_{n+h}\}'$ = h-steps-ahead forecast,

b - smooth transition function type (*b* = 1: logistic, ***b*** = 2: exponential),

c - threshold variable construction type,

c = 0 - $z_t = dTh_{t-d}$,

c = 1 - $z_t = AggF[\{dTh_{t-1}, \ldots, dTh_{t-d}\}]$,

c = 2 - $z_t = AggF[\{dTh_t, \ldots, dTh_{t-d}\}]$,

- spec2 - aggregation function name (the function must already be defined above), its argument must be a list of values. Function `Last` is set by default.
- spec3 - is purpose-dependent,

for *a* = 1, *a* = 2, there is no specification,

for *a* = 3, specification of how to obtain downsizing modifications of order, or / and additional orders. See details to function *fAicBic* (*fOrderAlternatives*).

for *a* = 4, *a* = 5, {size of sample-tail used for forecast evaluation, number of forecast steps, number of MonteCarlo iterations, out-of-sample{exogenous, threshold} variable values}. See details to function *fForecast*.

The default specification setup is {110} and need not be included at the end of arguments.

Note 1: Forecasting procedure uses fuction RandomArray which is the part of several Mathematica standard packages. It is required to pre load one of them.
Note 2: The outcome is stored in a system of successively nested lists according to $d \to r_1 \to \ldots \to r_{s-1}, \gamma_1 \to \ldots \to \gamma_{s-1}, \ldots$, that is, first level refers to delay, etc. The last levels depend on main purpose.

```
fConditionalRegimeSwitching[dMo_, dEx_, order_, dTh_, regimes_, parameters_, specifications_:{110}] := Module[
  {fThresholdConditions, fSmoothTransitionMultipliers, fLogisticSTF, fExponentialSTF, fRegimeMultiplier,
   fOrder, fDMRow, fMultiRegimeDesignMatrixRow, fStackAndPad, fParVal, fThresVarAggVectorRow, fiD,
   fOLS, fOLSres, fOrderAlternatives, fAicBic, fKStepForecast, fForecast, fParameterList, fOnError,
   an, ak, ah, aY, aiMoTh, adTh, aSpec, afAggF, aOrder, asName, aNameP, aValP, apRM, apX, aD, avXxD, aQxD, arr, aB2A, aOutput},
  (* ----- SHARED subroutine functions ----- *)
  (* fThresholdConditions returns list of all
      possible cases of threshold variable (name reserved to`z`) relation to threshold values
     E.g.: fThresholdConditions[{r1,r2}] → {z≤r1,z>r1&&z≤r2,z>r2} *)
  fThresholdConditions[r_] := Module[{arb1, arb2},
    Clear[z]; (* just in case... *)
    arb1 = Transpose[Outer[#1[z, #2] &, {LessEqual, Greater}, r]];
    arb2 = Join[#, Flatten[arb1], #] &[{True}];
    Apply[And, Partition[arb2, 2], 1]
   ];
  (* fSmoothTransitionMultipliers returns list of regime multipliers to be used for design matrix construction;
   - The name of threshold variable is reserved to `z`, name of transition function to `G`;
      E.g.: fSmoothTransitionMultipliers[{r1},{γ1}] → {1-G[z,r1,γ1],G[z,r1,γ1]}
         fSmoothTransitionMultipliers[{r1,r2},{γ1,γ2}] → {1-G[z,r1,γ1],G[z,r1,γ1]-G[z,r2,γ2],G[z,r2,γ2]} *)
  fSmoothTransitionMultipliers[r_, γ_] := Module[{arb1, arb2, arb3},
    Clear[z, G];
    arb1 = Inner[G[z, #1, #2] &, r, γ, List];
    arb2 = Outer[Times, arb1, {-1, 1}];
    arb3 = Join[#1, Flatten[arb2], #2] &[{1}, {0}];
    Apply[Plus, Partition[arb3, 2], 1]
   ];
  (* Logistic Smooth Transition Function with threshold variable `z`, threshold value `r` and smoothness parameter `γ` *)
  fLogisticSTF[z_, r_, γ_] := 1/(1 + Exp[-γ (z - r)]);
  (* Exponential Smooth Transition Function with threshold variable `z`,
   threshold value `r` and smoothness parameter `γ` *)
  fExponentialSTF[z_, r_, γ_] := 1 - Exp[-γ (z - r)^2];
  (* fRegimeMultiplier returns list of regime multipliers to be
    used for design matrix construction. The name of threshold variable is reserved to `z`;
       Examples: fRegimeMultiplier[{{r}}]→ {If[z≤r,1,0],If[z>r,1,0]};
           fRegimeMultiplier[{{r},{γ}}]= fRegimeMultiplier[{{r},{γ}},1]→ {1-1/(1+e^(-(-r+z) γ)), 1/(1+e^(-(-r+z) γ))};
           fRegimeMultiplier[{{r},{γ}},2]→ {e^(-(-r+z)^2 γ), 1-e^(-(-r+z)^2 γ)};
   Note: {r} can be {r1,r2,...}, and {γ}→ {γ1,γ2,...} *)
  fRegimeMultiplier[arg_, smoothTFtype_:1] := Module[{arb1},
    If[Length[arg] == 1,
     arb1 = (fThresholdConditions@@arg); If[#, 1, 0] & /@ arb1,
     arb1 = (fSmoothTransitionMultipliers@@arg); Switch[smoothTFtype,
      1, arb1 /. G → fLogisticSTF,
      2, arb1 /. G → fExponentialSTF]
    ]
   ];
  (* fOrder returns s×2 nested list of input variables lags used in regression in each regime,
   /ord - user input ARX order specification,
   /dimMo, dimEx - numbers of modeled and exogenous variables,
   /reg - number of regimes,
   Note: zero lag of modeled variables is considered as {}, i.e. variable is excluded from regression,
   unless set on the deepest level (which is then, however, a total nonsense),
   Levels structure: 1. level: regimes, 2. level: endo/exogeneity, 3. level: variables, 4. level: complete list of lags,
   Examples: fOrder[0,{1,2},2]→{{{{}},{{0},{0}}},{{{}},{{0},{0}}}}
   fOrder[{3,1},{1,2},2]→{{{{1,2,3}},{{1},{1}}},{{{1,2,3}},{{1},{1}}}}
   fOrder[{{3},{1,0}},{1,2},2]→{{{{1,2,3}},{{1},{0}}},{{{1,2,3}},{{1},{0}}}}
   fOrder[{{{3}},{{},0}},{1,2},2]→{{{{3}},{{},{0}}},{{{3}},{{},{0}}}}
   fOrder[{{{{3}},{{},0}},0},{1,2},2]→{{{{3}},{{},{0}}},{{{{}},{{0},{0}}}},
   Note: In order to define the lags separately for every regime, first regime must be specified at
    least to 3 rd level and the dimension of first level must be equal to reg (see the last example) *)
  fOrder[ord_, {dimMo_, dimEx_}, reg_] := Module[{bfLag},
    bfLag[lag_, {k1_, k2_}] := Module[{bTmp},
      bTmp = If[Length[lag] == 0, {lag, lag}, lag];
      bTmp = MapThread[If[Length[#1] == 0, Table[#1, {#2}], #1] &, {bTmp, {k1, k2}}];
      If[(Length /@ bTmp) ≠ {k1, k2}, Print["Lag misspecification"]; Abort[]];
      bTmp = MapAt[Replace[#, 0 → {0}, 2] &, bTmp, 2];
      Map[If[Length[#] == 0 ∧ ¬ VectorQ[#], Table[i, {i, 1, #}], #] &, bTmp, {2}]
     ];
    If[Length[ord] == reg ∧ Length[ ord⟦1⟧ ] == 2 ∧ (Length /@ ord⟦1⟧) == {dimMo, dimEx},
     bfLag[#, {dimMo, dimEx}] & /@ ord,
     Table[bfLag[ord, {dimMo, dimEx}], {reg}]
    ]
   ];
  (* fStackAndPad function encapsulates the Table and PadLeft functions *)
  fStackAndPad[function_, from_, to_, args__] := PadLeft[Table[function[i, args], {i, from, to}], to, {{}}];
  (* fDMRow creates ti^th row of singleregime design matrix,
   /mo, ex - k1,k2 × n modeled and exogenous data,
   /or - order of ARX = lag specification for every variable,
   /const = 1 if constant term is desired in regression, otherwise const = {},
   Example: if a={a1,a2,a3,a4,a5,a6},b={b1,b2,b3,b4,b5,b6}
      then fDMRow[6,{a},{a,b},{{{0,1,2}},{{},{0}}}]→{1,a6,a5,a4,b6}  *)
```

```
fDMRow[ti_, mo_, ex_, or_, const_: 1] := Module[{bXmo, bXex},
  bXmo = MapThread[Part, {mo, ti - or[[1]]}];
  bXex = MapThread[Part, {ex, ti - or[[2]]}];
  Flatten[{const, bXmo, bXex}]
 ];
(* fMultiRegimeDesignMatrixRow multiplies ti^th row of singleregime matrix by its corresponding regime multiplier,
 - fDMRow function required (shares most of the arguments),
 /or - list of order specifications for each regime,
 /regmultiplier - either vector of length=regimes, or time-ordered list of such vectors,
 Example: fMultiRegimeDesignMatrixRow[6,{a},{a,b},{{{{1,2}}},{{0},{1}}},{{{}}},{{0},{0}}}},{x,y}]→
  {x,a5 x,a4 x,a6 x,b5 x,y,a6 y,b6 y}, where x,y are some regime multiplers *)
fMultiRegimeDesignMatrixRow[ti_, mo_, ex_, or_, regmultiplier_] := Module[{bMultiDMatrix, bRMplier},
  bMultiDMatrix = fDMRow[ti, mo, ex, #] & /@ or;
  bRMplier = If[Length[regmultiplier] == regimes, regmultiplier, regmultiplier[[ti]]];
  Flatten[ MapThread[Times, {bMultiDMatrix, bRMplier}] ]
 ];
(* fParVal returns list of symbolic parameter names and their corresponding values.;
 fParVal transforms the list of parameter specification
  (either a single value, list of values or a range list) into list of  parameter values placed into
  separate sublist for every parameter in each regime transition. The first parameter (delay) is regime
  invariant. List of parameter values can also be specified directly for every regime transition.;
 Note: Any sublist in val must be of length 2 or 3 to be considered as a range definition.;
 Example: fParVal[3,{"d","r","γ"},{1,{{2,4,3,1},{6}},{1,3}}] →
  {{{d1},{r1,r2},{γ1,γ2}},{{{1}},{{2,4,3,1},{6}},{{1,2,3},{1,2,3}}}} *)
fParVal[regs_, name_, val_] := Module[{bVal, bReg, bNam},
  bReg = Join[{1}, Table[regs - 1, {Length[val] - 1}]];
  bNam = Inner[fParameterList[#1, #2, "", 0] &, name, bReg, List];
  bVal = If[Depth[#] > 1, #, {#}] & /@ val ;
  bReg = Join[{1}, Table[regs - 1, {Length[val] - 1}]];
  bVal = MapThread[
    Which[
      Depth[#1] == 2, Table[If[MemberQ[{3}, Length[#1]], Range @@ #1, #1], {#2}],
      Depth[#1] == 3 ∧ Length[#1] == (regs - 1) ∧ Min[Depth /@ #1] == 2, #1,
      1 == 1, Print["Parameter misspecification"]; Abort[];
     ] &, {bVal, bReg}
   ];
  {bNam, bVal}
 ];
(* fThresVarAggVectorRow returns value of threshold variable corresponding to time "ti"
  - threshold variable q_t = AggF[{dThV_{t-1(0)},...,dThV_{t-d}}], where AggF is aggregation function of list of arguments,
 and starting lag depends on specification (aSpec[[3]])
   - valueDelay must be a single value
   - requires aSpec (parent module variable)
    Example: (aSpec[[3]]=1) fThresVarAggVector[3,{1,2,3,4,5,6,7},2,Last] → 1 *)
fThresVarAggVectorRow[ti_, dataThresVar_, valueDelay_, nameAggF_] :=
 If[valueDelay < ti,
  nameAggF[dataThresVar[[ Reverse[Range[ ti - valueDelay, ti - If[aSpec[[3]] == 2, 0, Sign[valueDelay]] ]] ]]], {}];
(* fiD returns position of d in aD *)
fiD[d_] := Position[aD, d][[1, 1]];
(* fOLS returns - rule of input parameters, estimated param.matrix Φ and covariance matrix Σ,
 and residuals (if resQ is True) - ordinary least squares estimates from the data up to timepoint tmax *)
fOLS[tmax_, valp_, resQ_: False] := Module[{bname, bd, bh, brule, bvX, bΦ, bΣ, bres},
  bname = Flatten[aNameP];
  bd = valp[[1]];
  bh = Max[ah, bd];
  brule = Thread[Rule[bname, valp]];
  bvX = avXxD[[ fiD[bd] ]] /. brule ;
  bΦ = Inverse[ Sum[ Outer[Times, bvX[[ bt ]], bvX[[ bt ]]], {bt, bh+1, tmax}] ]. Sum[ Outer[Times, bvX[[ bt ]], aY[[ bt ]]], {bt, bh+1, tmax}];
  bres = Table[aY[[ bt ]] - bvX[[ bt ]].bΦ, {bt, bh + 1, tmax}];
  bΣ = (fTranspose[bres].bres) / (tmax - bh);

  If[resQ, {brule, bΦ, bΣ, Transpose[bres]}, {brule, bΦ, bΣ}]
 ];

(* ----- SPECIFIC subroutine functions ----- *)
(* fOrderAlternatives returns the list of all possible (restricted by spec) "subsets" of order given by "ord",
 (the first element in corresponding (lowest level) field is preserved, e.g. if the lag starts like {2,...},
  the most parsimonious version will be {2}, or if there's no element then {}→{} ),
 \ord - full specified order (system of lags),
 \spec = xyz - specification of how to perform downsizing in order "ord",
 neither part is compulsory (unless it is followed by others)- missing values are added from the default 110,
 x = 1 - decrease orders in all regimes simultaneously,
 x = 2 - --//-- per regimes,
 y = 1 - decrease orders for endo- and exogenous variables simultaneously,
 y = 2 - --//-- separately,
 z = 0 - decrease orders for both endo- and exogenous variables,
 z = 1 - --//-- only endogenous variables,
 z = 2 - --//-- only exogenous variables,
 Example: fOrderAlternatives[{{{{1,2,3}}},{{1,2},{}}},{{{1,2,3}},{{1,2},{}}}}, 120]→
  { {{{{1,2}}},{{1,2},{}}},{{{1,2}}},{{1,2},{}}}},
    {{{{1}}},{{1,2},{}}},{{{1}}},{{1,2},{}}}},
```

```
    {{{{1,2,3}},{{1},{}}},{{{1,2,3}},{{1},{}}}} } *)
fOrderAlternatives[ord_, spec_: 110] := Module[{f1, f2, f3, f4, f5, b1, b2, b3, breg, bpos},
  f1[a_] := If[ Length[a] == 0, a, Drop[a, -1] ];
  (* f2[{{1,2},{}}]→ {{1},{}} *)
  f2[a_] := If[ Length[a] == 0, a, f1[#] & /@ a ];
  (* f3[{{{1,2,3}},{{1,2},{}}}, {1}]→ {{{1,2}},{{1,2},{}}} *)
  f3[a_, pos_] := Table[ If[MemberQ[pos, j], f2[a[[j]]], a[[j]]], {j, Length[a]} ];
  (* tmp={ {{{1,2,3}},{{1,2},{}}}, {{{1,2,3}},{{1,2},{}}} } *)
  (* f4[tmp,{1},{2}]→ { {{{1,2,3}},{{1,2},{}}}, {{{1,2}},{{1,2},{}} } *)
  f4[a_, pos_, reg_] := Table[ If[MemberQ[reg, j], f3[a[[j]], pos], a[[j]]], {j, Length[a]} ];
  (* f5[tmp,{1},{2}]→ {{{{1,2,3}},{{1,2},{}}},{{{1,2}},{{1,2},{}}},{{{{1,2,3}},{{1,2},{}}},{{{1}},{{1,2},{}}}} *)
  f5[a_, pos_, reg_] := Module[{tmp},
    tmp = {a};
    While[ Max[Length /@ Flatten[Last[tmp][[reg, pos]], 2]] > 1, tmp = Append[tmp, f4[Last[tmp], pos, reg]]];
    Drop[tmp, 1]
   ];
  (* main *)
  {b1, b2, b3} = If[(b2 = Length[(b1 = IntegerDigits[spec])]) < 3, Join[b1, Reverse[{0, 1, 1}[[Range[3 - b2]]]], b1];
  breg = Switch[b1,
    1, {Range[Length[ord]]},
    2, Table[{j}, {j, Length[ord]}]];
  bpos = Switch[b2,
    1, DeleteCases[{{1, 2}}, 3 - b3],
    2, DeleteCases[{{1}, {2}}, {3 - b3}]];
  Flatten[Outer[f5[ord, #1, #2] &, bpos, breg, 1], 2]
 ];
(* fAicBic return a list {{AIC1,BIC1}, ...} of the information criteria for parameters in "par" and regime-
  variable-specific orders of ARX defined by aOrder and "spec".,
 \spec - specification of what orders are to be taken into account,
 it could be either integer digit interpreted by fOrderAlternatives or/and list of orders interpreted by fOrder.,
 Requires: aOrder, an, apRM, aQxD, dMo, dEx, regimes, ak;,
 Passes bOrdList={Ord1,Ord2,....} to parent module through aB2A variable. *)
fAicBic[par_, spec_: {110}] :=
 Module[{i, bname, bd, bh, brule, bOrdList, bvRMvec, bOrder, bvX, bΦ, bdimXj, bΦj, bXj, bΣj, bnj, bres, bresult},
  bname = Flatten[aNameP];
  bd = par[[1]];
  brule = Thread[Rule[bname, par]];
  bOrdList =
   Prepend[Flatten[If[IntegerQ[#], fOrderAlternatives[aOrder, #], fOrder[#, ak, regimes] & /@ #] & /@ spec, 1], aOrder];
  bvRMvec = Table[{}, {an}];
  For[i = bd + 1, i ≤ an, i++,
   bvRMvec[[i]] = apRM /. z → aQxD[[fiD[bd], i]]
  ];
  bvRMvec = bvRMvec /. brule;
  bresult = Table[{}, {Length[bOrdList]}];
  For[i = 1, i ≤ Length[bOrdList], i++,
   bOrder = bOrdList[[i]];
   bh = Max[bd, bOrder];
   bvX = fStackAndPad[fMultiRegimeDesignMatrixRow, bh + 1, an, dMo, dEx, bOrder, bvRMvec];
   bΦ = Inverse[ ∑_{bt=bh+1}^{an} Outer[Times, bvX[[bt]], bvX[[bt]]] ] . ∑_{bt=bh+1}^{an} Outer[Times, bvX[[bt]], aY[[bt]]];
   bdimXj = 1 + Length /@ (Flatten[#, 2] & /@ bOrder);
   bΦj = fPartition[bΦ, bdimXj];
   bXj = fStackAndPad[fDMRow, bh + 1, an, dMo, dEx, #] & /@ bOrder;
   bnj = Table[{}, {regimes}];
   bΣj = (Sum[ bres = aY[[bt]] - bXj[[#, bt]].bΦj[[#]]; Outer[ Times, bres, bres] * bvRMvec[[bt, #]], {bt, bh + 1, an} ] /
         (bnj[[#]] = Sum[bvRMvec[[bt, #]], {bt, bh + 1, an} ])) & /@ Range[regimes];
   bresult[[i]] = Plus @@ (bnj * Log[Det /@ bΣj]) + Apply[Plus, {2 ak[[1]] * bdimXj, Log[bnj] * bdimXj}, 1];
  ];
  aB2A = {bOrdList};
  bresult
 ];
(* fKStepForecast returns k predicted values {ŷ_{ti+1}, ŷ_{ti+2,...}, ŷ_{ti+kst}} given {y_{ti}, y_{ti-1,....}},
 /ti - starting (last given) point,
 /mo,ex - n×k modeled and exogenous data, ex can be {},
 /th -
  is either a threshold vector or an integer specifying which variable in mo is considered as the threshold variable,
 /kst - number of forecast steps (horizon),
 /kmc - number of MonteCarlo cycles,
 /model = {rule,Φ,σ} actual model parameters and results,
 - requires afAggF, aNameP, aOrder *)
fKStepForecast[ti_, mo_, ex_, th_, kst_, kmc_, model_] :=
 Module[{brule, bσ, bΦ, bd, bQ, bdMo, bdTh, bt, bRand, bXtp1, bdMotp1, bdMoR},
  {brule, bΦ, bσ} = model; bσ = √Tr[bσ, List];
  bd = aNameP[[1, 1]] /. brule;
  bQ = Take[aQxD[[ fiD[bd] ]], ti]; bdMo = Take[#, ti] & /@ mo; bdTh = If[ VectorQ[th], th, bdMo[[th]] ];
  For[bt = ti, bt ≤ ti + kst - 1, bt++,
   bQ = Append[bQ, fThresVarAggVectorRow[bt + 1, bdTh, bd, afAggF]];
   bRand = RandomArray[NormalDistribution[0, #], kmc] & /@ bσ;
   If[bt == ti
    ,
```

```
    bXtp1 = fMultiRegimeDesignMatrixRow[bt + 1, bdMo, ex, aOrder, apRM] /. z → bQ[[bt + 1]] /. brule;
    bdMotp1 = Transpose[{bXtp1.bΦ}];
    ,
    bdMotp1 = Sum[
        bdMoR = MapThread[ReplacePart[#1, #2, -1] &, {bdMo, Flatten[ bdMotp1 + Map[#[[i]] &, bRand] ]}];
        bXtp1 = fMultiRegimeDesignMatrixRow[bt + 1, bdMoR, ex, aOrder, apRM] /. z → bQ[[bt + 1]] /. brule;
        Transpose[{bXtp1.bΦ}], {i, kmc}] / kmc;
   ];
   bdMo = MapThread[Join, {bdMo, bdMotp1}];
   bdTh = If[ VectorQ[th], bdTh, Join[bdTh, bdMotp1[[th]]] ];
  ];
  Drop[#, ti] & /@ bdMo
 ];
(* fForecast returns {bnF×k {y_{an-bnF+1}-ŷ_{an-bnF+1},...,y_an-ŷ_an}, bkF×k {ŷ_{an+1},ŷ_{an+2},...,ŷ_{an+bkF}}},
 /par - list of values of d,r,γ parameters, e.g. {1, 0.54, 1000},
 /spec - specification for "forecast task" mode *)
fForecast[par_, spec_] := Module[{bnF, bkF, bkMC, bFEx, bFTh, bnY, bt, bOLS, bF, bFErr},
  {bnF, bkF, bkMC, {bFEx, bFTh}} = spec;
  (* /bnF - size of sample taken from the tail and used for forecast accuracy evaluation,
   /bkF - number of forecast steps (horizon, in theory denoted as "h"), /bkMC - number of MonteCarlo cycles,
   /bFEx,bFTh - pre-modelled further values of dEx and dTh (size ≥ bkF) *)
  bnY = an - bnF;
  For[bt = bnY + 1 - bkF; bFErr = {}, bt ≤ an - bkF, bt++,
   bOLS = fOLS[bt, par];
   bF = Last[Transpose[ fKStepForecast[bt, dMo, dEx, If[aiMoTh > 0, aiMoTh, adTh], bkF, bkMC, bOLS] ]] ;
   bFErr = Append[bFErr, bF];
  ];
  bFErr = Take[aY, -bnF] - bFErr;
  bOLS = fOLS[an, par];
  bF = fKStepForecast[an, dMo, MapThread[Join, {dEx, bFEx}], If[aiMoTh > 0, aiMoTh, Join[adTh, bFTh]], bkF, bkMC, bOLS] ;
  {Transpose[bFErr], bF}
 ];

(* ----- SUPLEMENTARY subroutine functions ----- *)
(* fParameterList[main_,rows_,secondary_,cols_] makes a list of names starting with "main1secondary1",
 it creates a list of lengt 'rows' with sublists of length 'cols':,
 - if cols=0, creates {main1, ..., main(rows)},
 - if cols is an integer,creates rows×cols matrix
  { {main1secondary1, ..., main1secondary(cols)}, ..., {main(rows) secondary1, ..., main(rows) secondary(cols)} },
 - if cols is a vector, every element represents number of elements in corresponding row,
 - if rows (or cols) > 9, the numbering begins with 01 (alt. 001, 0001, etc.),
 'main' and 'secondary' must be entered as string: "chainofanystrings";
 Example: fParameterList["r",3,"elem",{1,2,3}] → {{r1elem1},{r2elem1,r2elem2},{r3elem1,r3elem2,r3elem3}};
 fParameterList["r",3,"elem",0] → {r1,r2,r3}; *)
fParameterList[main_, rows_, secondary_, cols_] := Module[{fCharRange, singlerow, result},
  SetAttributes[StringJoin, Listable];
  fCharRange[max_] := Take[
    Flatten[Outer[StringJoin, Sequence @@ Table[CharacterRange["0", "9"], {Length[IntegerDigits[max]]}]]], {2, max + 1}];
  singlerow = StringJoin[main, fCharRange[rows]];
  result = Which[
    TrueQ[cols == 0], ToExpression /@ singlerow,
    NumberQ[cols] || Length[cols] == 1,
        Outer[StringJoin, singlerow, StringJoin[secondary, fCharRange[cols]]],
    VectorQ[cols] && (Length[cols] == rows),
        Table[StringJoin[singlerow[[i]], StringJoin[secondary, fCharRange[cols[[i]]]]], {i, rows}]
   ];
  ClearAttributes[StringJoin, Listable];
  ToExpression[result]
 ];
(* fPartition splits the list "a" into (nonoverlaping) sublists of lengths defined
  in "sublength". Following constraint on argument must be kept: Plus@@sublength ≤ Length[a],
 Example: fPartition[{1,2,3,4,5,6,7},{2,3,1}] → {{1,2},{3,4,5},{6}} *)
fPartition[a_, sublength_] := Module[{i, b1 = {}, b2 = 1},
  For[i = 1, i ≤ Length[sublength], i++,
   b1 = Append[b1, Take[a, {b2, (b2 = b2 + sublength[[i]]) - 1}]];
  ];
  b1];
(* fOnError checks for error mesages produced by MathKernel,
 if none occur it returns result of eval, otherwise prints message and aborts running procedure. *)
fOnError[eval_, message_] := Check[eval, Print[message]; Abort[];];

(* ---- variables definition ---- *)
an = Length[dMo[[1]]];
ak = Length /@ {dMo, dEx};
ah = Max[order];
aY = Transpose[dMo];
adTh = If[(aiMoTh = If[IntegerQ[dTh], dTh, Plus @@ Flatten[Position[dMo, dTh, 1]]]) > 0, dMo[[aiMoTh]], dTh];
(* dTh can be given as one of modelled variable by its position in dMo or explicitly as vector,
 this procedure also tests the (vector) dTh for presence in dMo (beware of duplication in dMo) *)
aSpec = IntegerDigits[ specifications[[1]] ];
afAggF = If[ aSpec[[3]] == 0, Last, specifications[[2]] ];
aOrder = fOrder[order, ak, regimes];
```

```
asName = Take[{"d", "r", "ϒ"}, Length[parameters]];
{aNameP, aValP} = fParVal[regimes, asName, parameters];
apRM = fRegimeMultiplier[ Drop[ aNameP, 1 ], aSpec[[2]] ];
apX = fStackAndPad[fMultiRegimeDesignMatrixRow, ah + 1, an, dMo, dEx, aOrder, apRM];
aD = aValP[[1, 1]];
(*
 If[Min[aD]==0∧aiMoTh>0,
   aD=DeleteCases[aD,0];
   aValP=ReplacePart[aValP,aD,{1,1}] ;
   Print["Zero delay removed from parameters list due to endogeneity of threshold variable."];
   If[Length[aD]==0,Abort[]];
  ];
 *)
avXxD = aQxD = Table[{}, {Length[aD]}, {an}];
For[i = 1, i ≤ Length[aD], i++,
 aQxD[[i]] = fStackAndPad[fThresVarAggVectorRow, aD[[i]] + 1, an, adTh, aD[[i]], afAggF ];
 For[j = Max[aD[[i]], ah] + 1, j ≤ an, j++,
  avXxD[[i, j]] = apX[[j]] /. z → aQxD[[i, j]];
 ];
];
arr = Range[regimes - 1];
aValP = Flatten[aValP, 1];
aB2A = {}; (* mediator of bOrdList from fAicBic to parent environment *)
(* ---- main routine ---- *)
aOutput = Switch[aSpec[[1]],
  1,
  Outer[If[Less @@ {##2}[[arr]], Tr[ fOLS[an, {##}][[3]] ], {}] &, Sequence @@ aValP],
  2,
  Outer[If[Less @@ {##2}[[arr]], fOLS[an, {##}, True][[{2, 3, 4}]], {}] &, Sequence @@ aValP],
  3,
  Outer[If[Less @@ {##2}[[arr]], fAicBic[{##}, specifications[[3]]], {}] &, Sequence @@ aValP] ,
  4,
  Outer[If[Less @@ {##2}[[arr]], fForecast[{##}, specifications[[3]][[1]], {}] &, Sequence @@ aValP] ,
  5,
  Outer[If[Less @@ {##2}[[arr]], fForecast[{##}, specifications[[3]][[2]], {}] &, Sequence @@ aValP] ,
  6,
  Outer[If[Less @@ {##2}[[arr]], fOLSres[an, {##}][[2]], {}] &, Sequence @@ aValP]
 ] ;
aOutput = {aOutput, Join[aValP, aB2A]};

Clear[fThresholdConditions, fSmoothTransitionMultipliers, fLogisticSTF, fExponentialSTF,
 fRegimeMultiplier, fOrder, fDMRow, fMultiRegimeDesignMatrixRow, fStackAndPad, fParVal, fThresVarAggVectorRow,
 fiD, fOLS, fOrderAlternatives, fAicBic, fKStepForecast, fForecast, fParameterList, fOnError,
 an, ak, ah, aY, aiMoTh, adTh, aSpec, afAggF, aOrder, asName, aNameP, aValP, apRM, apX, aD, avXxD, aQxD, arr, aB2A];

aOutput
];
```

Forecast processing

```
fUnNestAndCollectOutput[output_]
fMeanXPredictionErrorHitparade[ferrors_,g_]
fDieboldMArianoHitparade[ferrors_,horizon_,siglevel_,g_]
fCompareForecast[ferrors_,horizon_,siglevel_]....Modified Diebold-Mariano test
fPrintForecastErrors[a_,F_,k_]
fPlotForecast[orig_,ferrors_,determ_:0]
fPlotTheBestForecast[stseries_,outputfCRS_,detseries_List:{},lossfu_Function:(Dot[#,#]&),ratio_Real:1.1]
```

fUnNestAndCollectOutput returns unnested list of results with corresponding parameters appended (to each result item).
*Example: fUnNestAndCollectOutput[{{{{1.8},{1.9}},{{2.6},{2.7}}},{{1,2},{0,0.1},{10}}}] → {{1.8,{1,0,10}},{1.9,{1,0.1,10}},{2.6,{2,0,10}},{2.7,{2,0.1,10}}}.

```
fUnNestAndCollectOutput[output_] := MapThread[List,
   {Flatten[#[[1]], Length[#[[2]]] - 1], Flatten[Outer[List, Sequence @@ #[[2]]], Length[#[[2]]] - 1]} &[output]];
```

f....Hitparade functions evaluate forecast accuracy by comparing errors stacked in *ferrors*. Vector from each model is first transformed by function *g* at every time point (to get single value characterizing predictive accuracy at particular time, especially in multivariate case), then averaged and finally orderred increasingly (in case of *fMeanXPredictionErrorHitparade*), or tested for equality in *fCompareForecast* (in case of *fDieboldMarianoHitparade*).
The output of: "MXPE hitparade" - consist of enterig positions and of ordered means,
"DMtest hitparade" - consist of entering positions only. If some models are equally accurate, their positions are sublisted.
Example of function *g*: `Dot[#,#]&` means $g(\mathbf{e}_t) = \mathbf{e}_t'\mathbf{e}_t$ .... we compare (M)SPE (and note that $\mathbf{e}_t$ is multivariate),
`Abs` means $g(e_t) = |e_t|$.... we compare (M)APE

```
fMeanXPredictionErrorHitparade[ferrors_, g_] := {Ordering[#], Sort[#]} &[Mean /@ Map[g, Transpose /@ ferrors, {2}]];
fDieboldMarianoHitparade[ferrors_, horizon_, siglevel_, g_] :=
  fCompareForecast[Map[g, Transpose /@ ferrors, {2}], horizon, siglevel];
```

fCompareForecast returns hit parade of model competitors as the position of their corresponding forecast errors in *ferrors.*
ferrors - list of vectors of forecast errors from each individual model to be compared,
horizon - horizon of the foracasts (number of steps the forecast was made ahead),

siglevel - significance level.

Note: If some models are equally accurate, their entering position are sublisted, e.g., {{2,3},1,4} means that both 2^{nd} and 3^{rd} model are winers.

```
fCompareForecast[ferrors_, horizon_, siglevel_] := Module[{f2ModelTest, fPartition, tab, dim},
  (* f2ModelTest returns 1 if d is significantly
    negative (forecast errors of first model are smaller than of the second one),
   returns 0 if they are equally accurate and -1 if the second is better,
   /d = g(e1)-g(e2) loss differential, where ei are forecast errors and g some arbitrary function,
   /h - forecast horizon,
   /α - significance level; *)
  f2ModelTest[d_, h_, α_] := Module[{m, dmean, acov, i, DMstat, qα},
    m = Length[d];
    dmean = Mean[d];
    acov = Table[{}, {i, h}];
    For[i = 0, i ≤ h - 1, i++,
     acov[[i + 1]] = Sum[(d[[t]] - dmean) * (d[[t - i]] - dmean), {t, i + 1, m}] / m;
    ];
    DMstat = Sqrt[(m + 1 - 2 h + h (h - 1) / m)/m] dmean / Sqrt[(acov[[1]] + 2 If[h > 1, Sum[acov[[i]], {i, 2, h}], 0])/m];
    qα = Quantile[StudentTDistribution[m - 1], 1 - α / 2];
    Which[DMstat < -qα, 1, -qα ≤ DMstat ≤ qα, 0, True, -1]
   ];
  (* fPartition splits the list "a" into (nonoverlaping) sublists of lengths defined
    in "sublength". Following constraint on argument must be kept: Plus@@sublength ≤ Length[a],
   Example: fPartition[{1,2,3,4,5,6,7},{2,3,1}] → {{1,2},{3,4,5},{6}} *)
  fPartition[a_, sublength_] := Module[{i, b1 = {}, b2 = 1},
    For[i = 1, i ≤ Length[sublength], i++,
     b1 = Append[b1, Take[a, {b2, (b2 = b2 + sublength[[i]]) - 1}]];
    ];
    b1];
  dim = Length[ferrors];
  tab = Table[If[i < j, f2ModelTest[ferrors[[i]] - ferrors[[j]], horizon, siglevel], 0], {i, dim}, {j, dim}];
  tab = tab - Transpose[tab];
  Reverse[If[Length[#] == 1, Sequence @@ #, #] & /@ (fPartition[Ordering[#], Length /@ Split[Sort[#]]] &[Apply[Plus, tab, 2]])]
 ];


mae[a_, F_, k_] := N[Sum[Abs[a[[i]] - F[[i]]], {i, 1, k}]/k]

mse[a_, F_, k_] := N[Sum[(a[[i]] - F[[i]])^2, {i, 1, k}]/k]

rmse[a_, F_, k_] := N[Sqrt[Sum[(a[[i]] - F[[i]])^2, {i, 1, k}]/k]]

fPrintForecastErrors[a_, F_, k_] := TableForm[Transpose[{{"MSE", "RMSE", "MAE"}, {mse[a, F, k], rmse[a, F, k], mae[a, F, k]}}]]
```

fPlotForecast returns (multiple) plot of original and forecasted time series with optional deterministic component.

```
fPlotForecast[stseries_, ferrors_, determ_: 0, ratio_Real: 1.1] := Module[{as, af, ad, ar},
  {as, af} = If[Depth[#] < 3, {#}, #] & /@ {stseries, ferrors};
  ad = Switch[ Depth[determ], 1, Table[determ, {Length[as]}], 2, {determ}, 3, determ];
  Show[GraphicsArray[
    MapThread[Show[ListPlot[#1 + #3, DisplayFunction → Identity],
       ListPlot[#1 - #2 + #3, PlotStyle → {RGBColor[1, 0, 0]}, DisplayFunction → Identity],
       PlotRange → {ar = (Max[#1 + #3] - Min[#1 + #3]) * (ratio - 1); Min[#1 + #3] - ar, Max[#1 + #3] + ar}] &,
     {as, af, ad}]], DisplayFunction → $DisplayFunction];
  ]
```

fPlotTheBestForecast returns (multiple) plot of the *stseries* together with forecasts from the best model.

Further output: {parameters (d,r,γ), measure of fit} and forecast errors.

Input:

outputfCRS - output of the 4th mode of fConditionalRegimeSwitching,

detseries - deterministic component series to be added to stseries and forecasts,

lossfu - loss function,

ratio - vertical scale factor for the plots.

```
fPlotTheBestForecast[stseries_, outputfCRS_, detseries_List: {}, lossfu_Function: (Dot[#, #] &), ratio_Real: 1.1] :=
 Module[{a1, a2, a3, ak, an},
  a1 = fUnNestAndCollectOutput[outputfCRS];
  a2 = fMeanXPredictionErrorHitparade[a1[[All, 1]], lossfu];
  ak = Length[stseries];
  a3 = a1[[All, 1]][[a2[[1]]]][[1, #]] & /@ Range[ak];
  an = Length[a3[[1]]];
  fPlotForecast[Take[#, -an] & /@ stseries, a3, If[detseries == {}, 0, Take[#, -an] & /@ detseries], ratio];
  {{{"d,r,γ:  ", a1[[All, 2]][[a2[[1]]]][[1]]}, {"MSPE:  ", a2[[2, 1]]}} // MatrixForm, a3}
 ]
```