ARESLab

Adaptive Regression Splines toolbox for Matlab/Octave

ver. 1.13.0

Gints Jekabsons

E-mail: gints.jekabsons@rtu.lv
http://www.cs.rtu.lv/jekabsons/

User's manual

May, 2016

CONTENTS

1. INTRODUCTION

What is ARESLab

ARESLab is a Matlab/Octave toolbox for building piecewise-linear and piecewise-cubic regression models using the Multivariate Adaptive Regression Splines method (also known as MARS). (The term “MARS” is a registered trademark and thus not used in the name of the toolbox.) The author of the MARS method is Jerome Friedman (Friedman, 1991a; Friedman, 1993).

With this toolbox you can build MARS models (hereafter referred to as ARES models) for single-response and multi-response data, test them on separate test sets or using Cross-Validation, use the models for prediction, print their equations, perform ANOVA decomposition, assess input variable importance, as well as plot the models.

This user's manual provides overview of the functions available in the ARESLab.

ARESLab can be downloaded at http://www.cs.rtu.lv/jekabsons/.

The toolbox code is licensed under the GNU GPL ver. 3 or any later version. Some parts of functions `aresbuild` and `createList` were initially derived from ENTOOL toolbox (Merkwirth & Wichard, 2003) which also falls under the GPL licence.

Details

ARESLab toolbox is written entirely in Matlab/Octave. The MARS method is implemented according to the Friedman's original papers (Friedman, 1991a; Friedman, 1993). The knot placement algorithm is implemented very similarly to R package `earth` (Milborrow, 2016) (see description of `useMinSpan` and `useEndSpan` and remarks in Section 2.2).

One major difference is that the model building is not accelerated using the “fast least-squares update technique” (Friedman, 1991a). This difference however affects only the speed of the algorithm execution, not predictive performance of the built models.

The absence of the acceleration means that the code might be slow for large data sets (however, see description of `aresparams` on how to make the process faster by using the “Fast MARS” algorithm and/or setting more conservative values for algorithm parameters). An alternative is to use the open source package `earth` for R which is faster and in some aspects more sophisticated, however currently lacks the ability to create piecewise-cubic models. Yet another open source alternative is py-earth for Python (Rudy, 2016).

ARESLab does not automatically handle missing data or categorical input variables with more than two categories. Such categorical variables must be replaced with synthetic binary variables before using ARESLab, for example using function `dummyvar`.

Feedback

For any feedback on the toolbox including bug reports feel free to contact me via the email address given on the title page of this user's manual.

Citing the ARESLab toolbox

Jekabsons G., ARESLab: Adaptive Regression Splines toolbox for Matlab/Octave, 2016, available at http://www.cs.rtu.lv/jekabsons/

2. AVAILABLE FUNCTIONS

ARESLab toolbox provides the following list of functions:

- `aresbuild` – builds an ARES model;
- `aresparams, aresparams2` – creates a structure of ARES configuration parameters for further use with `aresbuild`, `arescv`, and `arescvc` functions;
- `arespredict` – makes predictions using ARES model;
- `arestest` – tests ARES model on a test data set;
- `arescv` – tests ARES performance using Cross-Validation; has additional built-in capabilities for finding the "best" number of basis functions for an ARES model;
- `arescvc` – finds the "best" value for penalty *c* of the Generalized Cross-Validation criterion from a set of candidate values using Cross-Validation;
- `aresplot` – plots ARES model, can visualize knot locations;
- `areseq` – prints equations of ARES model;
- `aresanova` – performs ANOVA decomposition;
- `aresanovareduce` – reduces ARES model according to ANOVA decomposition;
- `aresinfo` – lists basis functions of ARES model and tries to assess their relevance;
- `aresimp` – estimates input variable importance;
- `aresdel` – deletes basis functions from ARES model;
- `aresgetknots` – gets all knot locations of an ARES model for the specified input variable.

2.1. Function `aresbuild`

Purpose:

Builds a regression model using the Multivariate Adaptive Regression Splines method.

Call:

```
    [model, time, resultsEval] = aresbuild(Xtr, Ytr, trainParams, weights, keepX,
modelOld, dataEval, verbose)
```

All the input arguments, except the first two, are optional. Empty values are also accepted (the corresponding defaults will be used).

Input:

`Xtr, Ytr`	: `Xtr` is a matrix with rows corresponding to observations and columns corresponding to input variables. `Ytr` is either a column vector of response values or, for multi-response data, a matrix with columns corresponding to response variables. The structure of the output of this function changes depending on whether `Ytr` is a vector or a matrix (see below). `Xtr` type must be double. `Ytr` type must be double or logical (internally converted to double). Categorical variables in `Xtr` with more than two categories must be replaced with synthetic binary variables before using `aresbuild` (or any other ARESLab function), for example using function `dummyvar`. For multi-response data, each model will have the same set of basis functions but different coefficients. The models are built and pruned as usual but with the Residual Sum of Squares and GCVs summed across all responses. Since all the models are optimized simultaneously, the results for each model won't be as good as building the models independently.

However, the combined model may be better in other senses, depending on what you are trying to achieve. For example, it could be useful to select the set of basis functions that is best across all responses.

It is recommended to pre-scale `Xtr` values to [0,1] (Friedman, 1991a). This is because widely different locations and scales for the input variables can cause instabilities that could affect the quality of the final model. The MARS method is (except for numerics) invariant to the locations and scales of the input variables. It is therefore reasonable to perform a transformation that causes resulting locations and scales to be most favourable from the point of view of numeric stability (Friedman, 1991a).

For multi-response modelling, it is recommended to pre-scale `Ytr` values so that each response variable gets the appropriate weight during model building. A variable with higher variance will influence the results more than a variable with lower variance (Milborrow, 2016).

`trainParams`	: A structure of training parameters for the algorithm. If not provided, default values will be used (see function `aresparams` for details).
`weights`	: A vector of observation weights. The length of the vector must be the same as the number of observations in `Xtr` and `Ytr`. The weights must be nonnegative.
`keepX`	: Set to `true` to retain basis matrix `model.X` (see description of `model.X`). For multi-response modelling, the matrix will be replicated for each model. (default value = `false`)
`modelOld`	: If an already built ARES model is provided (whether pruned or not), no forward phase will be done. Instead the provided model will be taken directly to the backward phase and pruned. This is useful for fast tuning of parameters of the backward phase (`c`, `cubic`, `maxFinalFuncs`). Note that this is also a much faster way of changing a piecewise-linear model into a piecewise-cubic model or vice versa instead of building a new model from scratch. This argument is also used by function `arescvc` for fast selection of the "best" value for penalty `c` using Cross-Validation.
`dataEval`	: A structure containing test data in fields `X`, `Y`, and, optionally, `weights`. Used for getting evaluations for the best candidate models of each size in the backward pruning phase. For example, `arescv` uses it to help choosing a good value for the number of basis functions using Cross-Validation (see example of usage in Section 3.3). Results are saved in fields `R2test` and `MSEtest` of output argument `resultsEval`.
`verbose`	: Whether to output additional information to console (default value = `true`).

Output:

`model`	: A single ARES model for single-response `Ytr` or a cell array of ARES models for multi-response `Ytr`. A structure defining one model has the following fields:
`coefs`	: Coefficients vector of the regression model (first, for the intercept term, and then for all the rest of basis functions). Because of the coefficient for the intercept term, this vector is one row longer than the others.
`knotdims`	: Cell array of indices of used input variables for knots in each basis function.
`knotsites`	: Cell array of knot sites for each knot and used input variable in each basis function. `knotdims` and `knotsites` together contain all the information for locating the knots. If a variable entered a basis function linearly (i.e., without hinge function), the knot site for that variable is set to `minX`.

`knotdirs`	: Cell array of directions (-1 or 1) of the hinge functions for each used input variable in each basis function. If a variable entered a basis function linearly (i.e., without hinge function), the direction for that variable is set to 2.
`parents`	: Vector of indices of direct parents for each basis function (0 if there is no direct parent).
`trainParams`	: A structure of training parameters for the algorithm. The values are updated if chosen automatically. Except `useMinSpan`, because in automatic mode it is calculated for each parent basis function separately.
`MSE`	: Mean Squared Error of the model in the training data set.
`GCV`	: Generalized Cross-Validation of the model in the training data set. The value may also be `Inf` if model's effective number of parameters (see Eq. 1) is larger than or equal to the number of observations in the training data.
`t1, t2`	: For piecewise-cubic models only. Matrix of sites for the additional side knots on the left and on the right of the central knot.
`minX, maxX`	: Vectors defining the ranges of the input variables determined from the training data.
`isBinary`	: A vector indicating binary input variables. Determined automatically by counting unique values for each variable in training data. Therefore a variable can also be taken as binary by mistake if the data for some reason included only two values for the variable. Note that whether a variable is binary does not influence building of the model. This vector is further used in other functions to simplify printed equations.
`X`	: Basis matrix. Contains values of basis functions applied to `Xtr`. The number of columns in `X` is equal to the number of rows in `coefs`, i.e., the first column is for the intercept (all ones) and all the other columns correspond to the basis functions defined by `knotdims`, `knotsites`, `knotdirs`, `t1`, and `t2`. Each row corresponds to a row in `Xtr`. Multiplying `X` by `coefs` gives ARES prediction for `Ytr`. This variable is available only if argument `keepX` is set to `true`.
`time`	: Algorithm execution time (in seconds).
`resultsEval`	: Model evaluation results from the backward pruning phase. Fields `R2test` and `MSEtest` are available only if input argument `dataEval` is not empty. The structure has the following fields:
`MSE`	: MSE (Mean Squared Error) in training data for the best candidate model of each size.
`R2`	: R^2 (Coefficient of Determination) in training data for the best candidate model of each size.
`GCV`	: GCV (Generalized Cross-Validation) in training data for the best candidate model of each size. Contains `Inf` values for models with effective number of parameters larger than the number of observations in training data.
`R2GCV`	: R^2 estimated by GCV in training data for the best candidate model of each size. Contains `-Inf` values for models with effective number of parameters larger than the number of observations in training data.
`R2test`	: R^2 in `dataEval` test data for the best candidate model of each size.
`MSEtest`	: MSE in `dataEval` test data for the best candidate model of each size.
	Note that if `trainParams.cubic = true`, values of these fields are calculated using piecewise-linear models if `trainParams.cubicFastLevel = 2` and piecewise-cubic models if `trainParams.cubicFastLevel < 2`.

`usedVars` : Logical matrix showing which input variables were used in the best candidate model of each size.

Remarks:

The algorithm builds a model in two phases: forward selection and backward deletion. In the forward phase, the algorithm starts with a model consisting of just the intercept term and iteratively adds reflected pairs of basis functions giving the largest reduction of training error. The forward phase is executed until one of the following conditions is met:

1) reached maximum number of basis functions (`trainParams.maxFuncs`);
2) adding a new basis function changes R^2 by less than `trainParams.threshold`;
3) reached a R^2 of $1 -$ `trainParams.threshold` or more;
4) the number of coefficients in the model (i.e., the number of basis functions including the intercept term) has reached the number of data observations *n*;
5) optionally – model's effective number of parameters has reached the number of data observations *n* (see description of `trainParams.terminateWhenInfGCV` for details).

At the end of the forward phase we have a large model which typically overfits the data, and so a backward deletion phase is engaged. In the backward phase, the model is simplified by deleting one least important basis function (i.e., deletion of which reduces training error the least) at a time until the model has only the intercept term. At the end of the backward phase, from those "best" models of each size (except models larger than `trainParams.maxFinalFuncs`), the one with the lowest Generalized Cross-Validation (GCV) is selected and outputted as the final one.

GCV, as an estimator for prediction Mean Squared Error, for an ARES model is calculated as follows (Friedman, 1991a; Hastie et al., 2009; Milborrow, 2016):

$$GCV = MSE_{train} \Bigg/ \left(1 - \frac{enp}{n}\right)^2 , \tag{1}$$

where MSE_{train} is Mean Squared Error of the model in the training data, n is the number of observations in the training data, and enp is the effective number of parameters:

$$enp = k + c \times (k - 1)/2 , \tag{2}$$

where k is the number of basis functions in the model (including the intercept term) and c is `trainParams.c`. Note that $(k-1)/2$ is the number of hinge function knots, so the formula penalizes the model not only for its number of basis functions but also for its number of knots. Also note that in the situation when $enp \geq n$ the GCV value is set to `Inf` (the model is considered infinitely bad).

2.2. Function `aresparams`

Purpose:

Creates configuration for building ARES models. The output structure is for further use with `aresbuild`, `arescv`, and `arescvc` functions.

Call:

```
trainParams = aresparams(maxFuncs, c, cubic, cubicFastLevel, selfInteractions, maxInteractions, threshold, prune, fastK, fastBeta, fastH, useMinSpan, useEndSpan, maxFinalFuncs, endSpanAdjust, newVarPenalty, terminateWhenInfGCV, yesInteract, noInteract, allowLinear, forceLinear)
```

All the input arguments of this function are optional. Empty values are also accepted (the corresponding defaults will be used).

Parameters `prune` and `maxFinalFuncs` are used in the backward pruning phase. Parameters `c` and `cubic` may be used in both phases depending on `terminateWhenInfGCV`, `forceLinear`, and `cubicFastLevel`. All other parameters are used in the forward phase only.

For many applications, it can be expected that the most attention should be paid to the following parameters: `maxFuncs`, `maxInteractions`, `cubic`, `c`, and `maxFinalFuncs`. It is quite possible that the default values for `maxFuncs` and `maxInteractions` will be far from optimal for your data.

But note that, if you are prepared to use Cross-Validation, choosing a good value for `maxFinalFuncs` can sometimes release you from being too pedantic about parameters `maxFuncs` and `c`, because you can set large enough `maxFuncs` and not too large `c` and follow the example in Section 3.3.

If you have the necessary domain knowledge, it is recommended to also set `yesInteract`, `noInteract`, `allowLinear`, and `forceLinear`.

Input:

`maxFuncs`	: The maximum number of basis functions included in model in the forward building phase (before pruning in the backward phase). Includes the intercept term. The recommended value for this parameter is about two times the expected number of basis functions in the final model (Friedman, 1991a). Note that the algorithm may also not reach this number if some other termination condition happens first (see remarks on function `aresbuild` in Section 2.1). The default value for `maxFuncs` is -1 in which case it is calculated automatically using formula min(200, max(20, 2d)) + 1, where d is the number of input variables (Milborrow, 2016). This is fairly arbitrary but can be useful for first experiments. To enforce an upper bound on the final model size, use `maxFinalFuncs` instead. This is because the forward phase can see only one basis function ahead while the backward pruning phase can choose any of the built basis functions to include in the final model.
`c`	: Generalized Cross-Validation (GCV) penalty per knot. Larger values for `c` will lead to fewer knots (i.e., the final model will have fewer basis functions). A value of 0 penalizes only terms, not knots (can be useful, e.g., with lots of data, low or no noise, and highly structured underlying function of the data). Generally, the choice of the value for `c` should greatly depend on size of the dataset, how structured is the underlying function, and how high is the noise level, and mildly depend on the thoroughness of the optimization procedure, i.e., on the parameters `maxFuncs`, `maxInteractions`, and `useMinSpan` (Friedman, 1991a). Simulation studies suggest values for `c` in the range of about 2 to 4 (Friedman, 1991a). The default value for this parameter is -1 in which case `c` is chosen automatically using the following rule: if `maxInteractions` = 1 (additive modelling) `c` = 2, otherwise `c` = 3. These are the values recommended in Friedman, 1991a.
`cubic`	: Whether to use piecewise-cubic (`true`) or piecewise-linear (`false`) type of modelling. In general, it is expected that the piecewise-cubic modelling will give better predictive performance for smoother and less noisy data. (default value = `true`)
`cubicFastLevel`	: `aresbuild` implements three levels of piecewise-cubic modelling. In level 0, cubic modelling for each candidate model is done in both phases of the method (slow). In level 1, cubic modelling is done only in the backward phase (much faster). In level 2, cubic modelling is done after both phases, only for the final model (fastest). The default level is 2 (and it corresponds to the recommendations in Friedman, 1991a). Levels 0 and 1 may bring extra accuracy in the situations when, e.g., the underlying function of the

	data has sharp thresholds that for piecewise-cubic modelling require knot placements different that those required for piecewise-linear modelling.
`selfInteractions`	: This is experimental feature. The maximum degree of self interactions for any input variable. It can be set larger than 1 only for piecewise-linear modelling. The default, and recommended, value = 1, no self interactions.
`maxInteractions`	: The maximum degree of interactions between input variables. Set to 1 (default) for additive modelling (i.e., no interaction terms). For maximal interactivity between the variables, set the parameter to d×`selfInteractions`, where d is the number of input variables – this way the modelling procedure will have the most freedom building a complex model. Set to -1, so that `aresbuild` sets it automatically equal to d (maximal interactivity when self interactions are not used).
`threshold`	: One of the stopping criteria for the forward phase (see remarks section of function `aresbuild` for details). Default value = 1e-4. For noise-free data, the value may be lowered (e.g., to 1e-6) but setting it to 0 can cause numerical issues and instability.
`prune`	: Whether to perform model pruning (the backward phase). (default value = `true`)
`fastK`	: Parameter (integer) for Fast MARS algorithm (Friedman, 1993, Section 3.0). Maximum number of parent basis functions considered at each step of the forward phase. Typical values for `fastK` are 20, 10, 5 (default value = `Inf`, i.e., no Fast MARS). With lower `fastK` values model building is faster at the expense of some accuracy. Good starting values for exploratory work are `fastK` = 20, `fastBeta` = 1, `fastH` = 5 (Friedman, 1993). Friedman in his paper concluded that changing the values of `fastK` and `fastH` can have big effect on training computation times but predictive performance is largely unaffected over a wide range of their values (Friedman, 1993).
`fastBeta`	: Artificial ageing factor for Fast MARS algorithm (Friedman, 1993, Section 3.1). Typical value for `fastBeta` is 1 (default value = 0, i.e., no artificial ageing). The parameter is ignored if `fastK` = `Inf`.
`fastH`	: Parameter (integer) for Fast MARS algorithm (Friedman, 1993, Section 4.0). Number of iterations till next full optimization over all input variables for each parent basis function. Larger values make the search faster. Typical values for `fastH` are 1, 5, 10 (default value = 1, i.e., full optimization in every iteration). Computational reduction associated with increasing `fastH` is most pronounced for data sets with many input variables and when large `fastK` is used. There seems to be little gain in increasing `fastH` beyond 5 (Friedman, 1993). The parameter is ignored if `fastK` = `Inf`.
`useMinSpan`	: In order to lower the local variance of the estimates, a minimum span is imposed that makes the method resistant to runs of positive or negative error values between knots (by taking every `useMinSpan`-th observation for knot placement) (Friedman, 1991a). Setting `useMinSpan` to -1 (default), enables automatic mode (see remarks below). Setting `useMinSpan` to 0 or 1, disables the protection so that all the observations are considered for knot placement (except, see `useEndSpan`). Setting `useMinSpan` to > 1, enables manual tuning of the value. Disabling or lowering `useMinSpan` may allow creating a model which is more responsive to local variations in the data (can be especially useful if the number of data observations is small and noise is low) however this can also lead to overfitting. For further information and examples of usage, see Section 3.4.

`useEndSpan`	: In order to lower the local variance of the estimates near the ends of data intervals, a minimum span is imposed that makes the method resistant to runs of positive or negative error values between extreme knot locations and the corresponding ends of data intervals (by not allowing to place knots too near to the ends of the intervals) (Friedman, 1991a). Setting `useEndSpan` to -1 (default), enables automatic mode that chooses value for this parameter depending on the number of input variables (but never lower than 7). Setting `useEndSpan` to 0, disables the protection so that all the observations are considered for knot placement (except, see `useMinSpan`). Setting `useEndSpan` to > 1, enables manual tuning of the value. Disabling or lowering `useEndSpan` may allow creating a model which is more responsive to local variations near the edges of the data (can be especially useful if the number of data observations is small and noise is low) however this can also lead to overfitting. For further information and examples of usage, see remarks below and Section 3.4.
`maxFinalFuncs`	: Maximum number of basis functions (including the intercept term) in the final pruned model (default value = `Inf`). Use this (rather than the `maxFuncs` parameter) to enforce an upper bound on the final model size. See Section 3.3 for an example on how to choose value for this parameter using Cross-Validation.
`endSpanAdjust`	: For basis functions with variable interactions, `useEndSpan` gets multiplied by this value. This reduces probability of getting overfitted interaction terms supported by just a few observations on the boundaries of data intervals. Still, at least one knot will always be allowed in the middle, even if `endSpanAdjust` would prohibit it. Useful values range from 1 to 10. (default value = 1, i.e., no adjustment)
`newVarPenalty`	: Penalty for adding a new variable to a model in the forward phase. This is the gamma parameter of Eq. 74 in the original paper (Friedman, 1991a). The higher is the penalty, the more reluctant will be the forward phase to add a new variable to the model – it will rather try to use variables already in the model. This can be useful when some of the variables are highly collinear. As a result, the final model may be easier to interpret although usually the built models also will have worse predictive performance. Useful non-zero values typically range from 0.01 to 0.2 (Milborrow, 2016). (default value = 0, i.e., no penalty)
`terminateWhenInfGCV`	: Whether to check termination condition, terminating forward phase when the effective number of parameters of a model reaches the number of observations in training data, i.e., when GCV for such large models would be `Inf` (see remarks section in description of function `aresbuild` on GCV). In such cases it could be pointless to continue because larger models wouldn't be considered as candidates for final model anyway. On the other hand, some of the added basis functions could still turn out to be useful for inclusion in final model. Note that the effective number of parameters is not the same as the number of regression coefficients, except when `c` = 0 (in which case enabling `terminateWhenInfGCV` has no additional effect). (default value = `false`)
`yesInteract`	: A matrix indicating pairs of input variables that are allowed to interact with each other in the ARES model. The matrix must have two columns. Each row is a pair of indices for the input variables. Default value = `[]`. Cannot be used together with `noInteract`.
`noInteract`	: A matrix indicating pairs of input variables that are not allowed to interact with each other in the ARES model. The matrix must have two columns.

	Each row is a pair of indices for the input variables. Default value = `[]`. Cannot be used together with `yesInteract`.
`allowLinear`	: Whether to allow input variables to enter basis functions linearly, i.e., without hinge functions. Such basis functions are added to the model one at a time, as opposed to basis functions with new hinges that are added two at a time – one for each hinge of the reflected pair. Set `allowLinear` to 0 (default), to disallow variables entering linearly, i.e., consider hinge functions only (except see `forceLinear`). Set to 1, to allow, and treat error reduction associated with adding such basis function the same way as for a pair of basis functions with new hinges. Set to 2, to prefer variables entering basis functions linearly. This is done by calculating error reduction of such basis functions using GCV (instead of sum of squared error), resulting in preference of adding a single basis function instead of two even when this produces slightly smaller error reduction. Note that the R package `earth` (Milborrow, 2016) has options "0" and "1" while py-earth (Rudy, 2016) has options "0" and "2".
`forceLinear`	: A vector of indices of input variables that should be forced to enter the model only linearly, i.e., without hinge functions. This overrides `allowLinear` for the listed variables. Note that `forceLinear` does not say that a variable must enter the model; only that if it enters, it enters linearly. Also note that it has nothing to do with whether a variable is allowed to interact with other variables. Default value = `[]`.

Output:

`trainParams`	: A structure of parameters for further use with `aresbuild`, `arescv`, and `arescvc` functions containing the provided values (or defaults, if not provided).

Remarks:

The knot placement algorithm in `aresbuild` is implemented very similarly to R package `earth` (Milborrow, 2016). `useMinSpan` and `useEndSpan` are calculated using formulas given in Eq. 45 and Eq. 43 of the Friedman's original paper (Friedman, 1991a) with $alpha = 0.05$. For a fixed dimensionality of the data, `useEndSpan` always stays the same but `useMinSpan` is recalculated for each individual parent basis function used for generating new basis functions. The knots are placed symmetrically so that there are approximately equal number of skipped observations at each end of data intervals. For further information and examples of usage, see Section 3.4.

If more speed is required, try using the Fast MARS algorithm by setting `fastK` parameter to something other than `Inf`. Good starting values for exploratory work are `fastK` = 20, `fastBeta` = 1, `fastH` = 5 (Friedman, 1993). For more information, see descriptions of the mentioned parameters and the Friedman's paper.

Alternatively, for more speed you can try some of the following options (if they are adequate for your situation):

1) decreasing `maxFuncs` (less iterations in forward phase);
2) setting `cubicFastLevel` = 2 (faster computations; has effect on piecewise-cubic modelling only);
3) decreasing `selfInteractions` (less candidate models in forward phase);
4) decreasing `maxInteractions` (less candidate models in forward phase);
5) enabling `terminateWhenInfGCV` (sometimes less iterations in forward phase);
6) setting `yesInteract` or `noInteract` so that the algorithm doesn't waste it's time on needless interactions between input variables (less candidate models in forward phase);
7) setting `forceLinear` for input variables you are sure should enter the model only linearly, i.e., without hinge functions (less candidate models in forward phase);

8) manually increasing `useMinSpan` and/or `useEndSpan` (less candidate models in forward phase);
9) increasing `threshold` (not recommended; less iterations in forward phase).

Note that decreasing the number of iterations or candidate models may also result in worse final models.

2.3. Function `aresparams2`

Purpose:

Creates configuration for building ARES models. The output structure is for further use with `aresbuild`, `arescv`, and `arescvc` functions.

This function is an alternative to function `aresparams` for supplying parameters as name/value pairs.

Call:

```
trainParams = aresparams2(varargin)
```

Input:

`varargin`	: Name/value pairs for the parameters. For the list of the names, see description of function `aresparams`.

Output:

`trainParams`	: A structure of parameters for further use with `aresbuild`, `arescv`, and `arescvc` functions containing the provided values (or defaults, if not provided).

2.4. Function `arespredict`

Purpose:

Predicts response values for the given query points using ARES model.

Call:

```
[Yq, BX] = arespredict(model, Xq)
```

Input:

`model`	: ARES model or, for multi-response modelling, a cell array of ARES models.
`Xq`	: A matrix of query data points.

Output:

`Yq`	: Either a column vector of predicted response values or, for multi-response modelling, a matrix with columns corresponding to response variables.
`BX`	: Basis matrix. Contains values of basis functions applied to `Xq`.

2.5. Function `arestest`

Purpose:

Tests ARES model on a test data set (`Xtst`, `Ytst`).

Call:

```
results = arestest(model, Xtst, Ytst, weights)
```

Input:

`model`	: ARES model or, for multi-response modelling, a cell array of ARES models.
`Xtst, Ytst`	: `Xtst` is a matrix with rows corresponding to testing observations, and columns corresponding to input variables. `Ytst` is either a column vector of response values or, for multi-response data, a matrix with columns corresponding to response variables.
`weights`	: Optional. A vector of weights for observations. See description of function `aresbuild`.

Output:

`results`	: A structure of different error measures calculated on the test data set. For multi-response data, all error measures are given for each model separately in a row vector. The structure has the following fields:
`MAE`	: Mean Absolute Error.
`MSE`	: Mean Squared Error.
`RMSE`	: Root Mean Squared Error.
`RRMSE`	: Relative Root Mean Squared Error.
`R2`	: Coefficient of Determination.

2.6. Function `arescv`

Purpose:

Tests ARES performance using *k*-fold Cross-Validation.

The function has additional built-in capabilities for finding the "best" number of basis functions for the final ARES model (`maxFinalFuncs` for function `aresparams`). See example of usage in Section 3.3 for details.

Call:

```
[resultsTotal, resultsFolds, resultsPruning] = arescv(X, Y, trainParams, k,
shuffle, nCross, weights, testWithWeights, evalPruning, verbose)
```

All the input arguments, except the first three, are optional. Empty values are also accepted (the corresponding defaults will be used).

Note that, if argument `shuffle` is set to `true`, this function employs random number generator for which you can set seed before calling the function.

Input:

`X, Y`	: The data. See description of function `aresbuild`.
`trainParams`	: A structure of training parameters (see function `aresparams` for details).
`k`	: Value of *k* for *k*-fold Cross-Validation. The typical values are 5 or 10. For Leave-One-Out Cross-Validation set *k* equal to *n*. (default value = 10)
`shuffle`	: Whether to shuffle the order of observations before performing Cross-Validation. (default value = `true`)
`nCross`	: How many times to repeat Cross-Validation with different data partitioning. This can be used to get more stable results. Default value = 1, i.e., no repetition. Useless if `shuffle` = `false`.

`weights`	: A vector of weights for observations. See description of function `aresbuild`.
`testWithWeights`	: Set to `true` to use `weights` vector for both, training and testing. Set to `false` to use it for training only. This argument has any effect only when `weights` vector is provided. (default value = `true`)
`evalPruning`	: Whether to evaluate all the candidate models of the pruning phase. If set to `true`, the output argument `resultsPruning` contains the results. See example of usage in Section 3.3. (default value = `false`)
`verbose`	: Whether to output additional information to console. (default value = `true`)

Output:

`resultsTotal`	: A structure of Cross-Validation results. The results are averaged across Cross-Validation folds and, in case of multi-response data, also across multiple models.
`resultsFolds`	: A structure of vectors or matrices (in case of multi-response data) of results for each Cross-Validation fold. Columns correspond to Cross-Validation folds. Rows correspond to models.

Both structures have the following fields:

`MAE`	: Mean Absolute Error.
`MSE`	: Mean Squared Error.
`RMSE`	: Root Mean Squared Error.
`RRMSE`	: Relative Root Mean Squared Error. Not reported for Leave-One-Out Cross-Validation.
`R2`	: Coefficient of Determination. Not reported for Leave-One-Out Cross-Validation.
`nBasis`	: Number of basis functions in model (including the intercept term).
`nVars`	: Number of input variables included in model.
`maxDeg`	: Highest degree of variable interactions in model.

`resultsPruning`	: Available only if `evalPruning` = `true`. See example of usage in Section 3.3. The structure has the following fields:
`GCV`	: A matrix of GCV values for best candidate models of each size at each Cross-Validation fold. The number of rows is equal to `k`×`nCross`. Column index corresponds to the number of basis functions in a model.
`meanGCV`	: A vector of mean GCV values for each model size across all Cross-Validation folds.
`nBasisGCV`	: The number of basis functions (including the intercept term) for which the mean GCV is minimum.
`MSEoof`	: A matrix of out-of-fold MSE values for best candidate models of each size at each Cross-Validation fold. The number of rows for this matrix is equal to `k`×`nCross`. Column index corresponds to the number of basis functions in a model.
`meanMSEoof`	: A vector of mean out-of-fold MSE values for each model size across all Cross-Validation folds.
`nBasisMSEoof`	: The number of basis functions (including the intercept term) for which the mean out-of-fold MSE is minimum.
`R2GCV`	: A matrix of R^2_{GCV} (R^2 estimated by GCV in training data) values for best candidate models of each size at each Cross-Validation fold. The number of rows is equal to `k`×`nCross`. Column index corresponds to the number of basis functions in a model.

`meanR2GCV`	: A vector of mean R^2_{GCV} values for each model size across all Cross-Validation folds.
`nBasisR2GCV`	: The number of basis functions (including the intercept term) for which the mean R^2_{GCV} is maximum.
`R2oof`	: A matrix of out-of-fold R^2 values for best candidate models of each size at each Cross-Validation fold. The number of rows for this matrix is equal to `k`×`nCross`. Column index corresponds to the number of basis functions in a model.
`meanR2oof`	: A vector of mean out-of-fold R^2 values for each model size across all Cross-Validation folds.
`nBasisR2oof`	: The number of basis functions (including the intercept term) for which the mean out-of-fold R^2 is maximum.

2.7. Function `arescvc`

Purpose:

Finds the "best" value for penalty *c* of the Generalized Cross-Validation criterion from a set of candidate values using Cross-Validation assuming that all the other parameters of function `aresparams` would stay fixed. For a better alternative to using this function, see Section 3.3.

Call:

```
[cBest, results] = arescvc(X, Y, trainParams, cTry, k, shuffle, nCross,
weights, testWithWeights, verbose)
```

All the input arguments, except the first three, are optional. Empty values are also accepted (the corresponding defaults will be used).

Note that, if argument `shuffle` is set to `true`, this function employs random number generator for which you can set seed before calling the function.

Input:

`X, Y`	: The data. See description of function `aresbuild`.
`trainParams`	: A structure of training parameters (see function `aresparams` for details).
`cTry`	: A set of candidate values for *c*. (default value = `1:5`)
`k`	: Value of *k* for *k*-fold Cross-Validation. The typical values are 5 or 10. For Leave-One-Out Cross-Validation set *k* equal to *n*. (default value = 10)
`shuffle`	: Whether to shuffle the order of the observations before performing Cross-Validation. (default value = `true`)
`nCross`	: How many times to repeat Cross-Validation with different data partitioning. This can be used to get more stable results. Default value = 1, i.e., no repetition. Useless if `shuffle` = `false`.
`weights`	: A vector of weights for observations. See description of function `aresbuild`.
`testWithWeights`	: Set to `true` to use `weights` vector for both, training and testing. Set to `false` to use it for training only. This argument has any effect only when `weights` vector is provided. (default value = `true`)
`verbose`	: Whether to output additional information to console. (default value = `true`)

Output:

`cBest`	: The best found value for penalty *c*.

`results`	: A matrix with two columns. First column contains all values from `cTry`. Second column contains the calculated MSE values (averaged across all Cross-Validation folds) for the corresponding `cTry` values.

Remarks:

This function finds the "best" penalty *c* value in a clever way. In each Cross-Validation iteration, the forward phase in `aresbuild` is done only once while the backward phase is done separately for each `cTry` value. The results will be the same as if each time a full model building process would be performed because in the forward phase the GCV criterion is not used. Except if `aresparams` parameter `terminateWhenInfGCV` is set to `true` – in that case the results may sometimes slightly differ.

2.8. Function `aresplot`

Purpose:

Plots ARES model. For datasets with one input variable, plots the model together with its knot locations. For datasets with more than one input variable, plots 3D surface. If `idx` is not provided, checks if the model uses more than one variable and, if not, plots in 2D even if the dataset has more than one input variable.

For multi-response modelling, supply one submodel at a time.

Call:

```
fh = aresplot(model, idx, vals, minX, maxX, gridSize, showKnots, varargin)
```

All the input arguments, except the first one, are optional. Empty values are also accepted (the corresponding defaults will be used).

Input:

`model`	: ARES model.
`idx`	: Only used when the number of input variables is larger than two. This is a vector containing two indices for the two variables values of which are to be varied in the plot (default value = `[1 2]`).
`vals`	: Only used when the number of input variables is larger than two. This is a vector of fixed values for all the input variables (except that the values for the varied variables are not used). By default, continuous variables are fixed at (`minX` + `maxX`) / 2 but binary variables (according to `model.isBinary`) are fixed at `minX`.
`minX, maxX`	: Minimum and maximum values for each input variable (this is the same type of data as in `model.minX` and `model.maxX`). By default, those values are taken from `model.minX` and `model.maxX`.
`gridSize`	: Grid size for the plot. Default value is 400 for 2D plots and 50 for 3D plots.
`showKnots`	: Whether to show knots in the plot (default value is `true` for data with one input variable and `false` otherwise). Showing knot locations in 3D plots is experimental feature. In a 3D plot, knots for basis functions without interactions are represented as planes with white edges while knots for basis functions with interactions are represented as 90-degrees "broken planes" with black edges. The directions of the broken planes depend on directions of hinge functions in the corresponding basis functions. Planes for each new knot (or pair of knots) are shown using a new color and a new vertical offset so that they are easier to distinguish. Unfortunately,

	coincident planes flicker; though it helps seeing that there is more than one plane. Note that, for input variables entering linearly (i.e., without hinge functions), 2D plots don't show any knots but 3D plots show knots at `minX`.
`varargin`	: Name/value pairs of arguments passed to function `plot` (for 2D plots) and function `surfc` (for 3D plots). May include `'XLim'`, `'YLim'`, and `'ZLim'` which are separated and passed to axes.

Output:

`fh`	: Handle to the created figure.

2.9. Function `areseq`

Purpose:

Prints equations of ARES model.
For multi-response modelling, supply one submodel at a time.

Call:

```
eq = areseq(model, precision, varNames, binarySimple, expandParentBF, cubicSmoothing)
```

All the input arguments, except the first one, are optional. Empty values are also accepted (the corresponding defaults will be used).

Input:

`model`	: ARES model.
`precision`	: Number of digits in the model coefficients and knot sites. Default value = 15.
`varNames`	: A cell array of variable names to show instead of the generic ones.
`binarySimple`	: Whether to simplify basis functions that use binary input variables (default value = `false`). Note that whether a variable is binary is automatically determined during model building in `aresbuild` by counting unique values for each variable in training data. Therefore a variable can also be taken as binary by mistake if the data for some reason includes only two values for the variable. You can correct such mistakes by editing `model.isBinary`. Also note that whether a variable is binary does not influence building of models. It's just used here to simplify equations. The argument has no effect if the model was allowed to have input variables to enter linearly, because then all binary variables are handled using linear functions instead of hinge functions.
`expandParentBF`	: A basis function that involves multiplication of two or more hinge functions can be defined simply as a multiplication of an already existing basis function (parent) and a new hinge function. Alternatively, it can be defined as a multiplication of a number of hinge functions. Set `expandParentBF` to `false` (default) for the former behaviour and to `true` for the latter.
`CubicSmoothing`	: This is for piecewise-cubic models only. Set to `'short'` (default) to show piecewise-cubic basis functions in their short mathematical form (Equation 34 in Friedman, 1991a). Set to `'full'` to show all computations involved in calculating the response value. Set to `'hide'` to hide cubic smoothing and see the model as if it would be piecewise-linear. It's easier to

understand the equations if smoothing is hidden. Note that, while the model then looks like piecewise-linear, the coefficients are for the actual piecewise-cubic model.

Output:

`eq`	: A cell array of strings containing equations for individual basis functions and the main model.

2.10. Function `aresanova`

Purpose:

Performs ANOVA decomposition of given ARES model and reports the results. For details, see remarks below as well as Sections 3.5 and 4.3 in (Friedman, 1991a) and Sections 2.4 and 4.4 in (Friedman, 1991b).

For multi-response modelling, supply one submodel at a time.

Call:

```
aresanova(model, Xtr, Ytr, weights)
```

Input:

`model`	: ARES model.
`Xtr, Ytr`	: Training data observations. The same data that was used when the model was built.
`weights`	: Optional. A vector of weights for observations. The same weights that were used when the model was built.

Remarks:

To understand the outputted table, below is an excerpt from the original paper by Jerome Friedman (Friedman, 1991a) Section 4.3. Note that in the excerpt, starting from the mentioning of the fourth column, all the column numbers should be increased by one. This is because `aresanova` adds an additional column reporting GCV estimate of the Coefficient of Determination R^2 (called R2GCV) as suggested in (Friedman, 1991b). It estimates the proportion of variance explained when all the basis functions comprising the ANOVA function are excluded from the model. By comparing it to the GCV estimate of R^2 for the full model, one can see the amount of reduction the exclusion brings.

"The ANOVA decomposition is summarized by one row for each ANOVA function. The columns represent summary quantities for each one. The first column lists the function number. The second gives the standard deviation of the function. This gives one indication of its (relative) importance to the overall model and can be interpreted in a manner similar to a standardized regression coefficient in a linear model. The third column provides another indication of the importance of the corresponding ANOVA function, by listing the GCV, score for a model with all of the basis functions corresponding to that particular ANOVA function removed. This can be used to judge whether this ANOVA function is making an important contribution to the model, or whether it just slightly helps to improve the global GCV score. The fourth column gives the number of basis functions comprising the ANOVA function while the fifth column provides an estimate of the additional number of linear degrees-of-freedom used by including it. The last column gives the particular predictor variables associated with the ANOVA function."

If it is determined that by deleting a one specific ANOVA function GCV would decrease (i.e., model would get better) or stay about the same, you will see an exclamation mark next to the GCV value of that ANOVA function. See remarks on `aresinfo` for the same situation with basis functions.

2.11. Function `aresanovareduce`

Purpose:

Deletes all the basis functions from ARES model (without recalculating model's coefficients and relocating additional knots of piecewise-cubic models) in which at least one used variable is not in the given list of allowed variables. This can be used to perform ANOVA decomposition as well as for investigation of individual and joint contributions of variables in the model, i.e., the reduced model can then be plotted to visualize the contributions.

For multi-response modelling, supply one submodel at a time.

Call:

```
[model, usedBasis] = aresanovareduce(model, varsToStay, exact)
```

Input:

`model`	: ARES model.
`varsToStay`	: A vector of indices for input variables to stay in the model. The size of the vector should be between one and the total number of input variables.
`exact`	: Set this to true to get a model with only those basis functions where the exact combination of variables is present (default value = `false`). This is used from function `aresanova`.

Output:

`model`	: Reduced ARES model.
`usedBasis`	: Vector of original indices for basis functions still in use.

2.12. Function `aresinfo`

Purpose:

Takes an ARES model, prints each basis function together with MSE, GCV, and R2GCV (GCV estimate of the Coefficient of Determination, R^2) for a model from which the basis function was removed. By default, the functions are listed in the order of decreasing GCV – bigger is better. This can be used to judge whether, in the specific context of the given full model, a basis function is making an important contribution, or whether it just slightly helps to improve the global GCV score. See remarks below.

For multi-response modelling, supply one submodel at a time.

Call:

```
aresinfo(model, Xtr, Ytr, weights, showBF, sortByGCV, binarySimple, expandParentBF, cubicAsLinear)
```

All the input arguments, except the first three, are optional. Empty values are also accepted (the corresponding defaults will be used).

Input:

`model`	: ARES model.
`Xtr, Ytr`	: Training data observations. The same data that was used when the model was built.
`weights`	: A vector of weights for observations. The same weights that were used when the model was built.

`showBF`	: Whether to show equations of basis functions or just list input variables the basis functions are using (default value = `true`).
`sortByGCV`	: Whether to list basis functions in the order of decreasing GCV or in the order in which they were included in the model (default value = `true`).
`binarySimple`	: See description of input argument of the same name for function `areseq`. (default value = `false`).
`expandParentBF`	: See description of input argument of the same name for function `areseq`. (default value = `false`).
`cubicAsLinear`	: This is for piecewise-cubic models only. Set to `false` (default) to show piecewise-cubic basis functions in their own mathematical form (Equation 34 in Friedman, 1991a). Set to `true` to hide cubic smoothing – see the basis functions as if the model would be piecewise-linear. It's easier to understand the equations if smoothing is hidden. Note that, while the basis functions then look like from a piecewise-linear model, the coefficients are from the actual piecewise-cubic model.

Remarks:

1. If it is determined that by deleting a one specific basis function GCV would decrease (i.e., model would get better) or stay about the same, you will see an exclamation mark next to the GCV value of that basis function. This can happen either because the basis function is irrelevant or it's redundant with some other basis function(s) in the model. But note that if more than one basis function has such mark, it does not mean that all of them should be deleted at once or at all. Instead it means that you can try deleting them one after another (using function `aresdel`) starting from the least important one, each time recalculating this table, until all of the basis functions still left in model stop having that mark. This is similar to what the backward pruning phase does, except that it continues until model consists only of the intercept term and then selects the model with the best GCV from all tried sizes.
2. If you are using piecewise-cubic modelling with the default value for parameter `cubicFastLevel` you may sometimes see that a basis function has an exclamation mark even though you didn't disable the backward pruning phase and therefore all irrelevant and redundant basis functions should be already deleted. This is because by default models are pruned as piecewise-linear and only after pruning they become piecewise-cubic therefore it's possible that a basis function inclusion of which previously slightly reduced GCV, suddenly slightly increases it.
3. The column "hinges" shows types of functions that are multiplied to comprise the basis function. Hinge functions are shown as "_/" or "_". Linear functions for variables that entered linearly are shown as "/". The functions are showed in the same order as in the column "basis function".

2.13. Function `aresimp`

Purpose:

Performs input variable importance assessment and reports the results. For details, see remarks below.

For multi-response modelling, supply one submodel at a time.

Call:

```
varImp = aresimp(model, Xtr, Ytr, resultsEval, weights)
```

The first three input arguments are required.

Input:

`model`	: ARES model.
`Xtr, Ytr`	: Training data observations. The same data that was used when the model was built.
`resultsEval`	: `resultsEval` from function `aresbuild`. Do not use this argument if `model` was modified by any function other than `aresbuild` (i.e., `aresdel` or `aresanovareduce`).
`weights`	: A vector of weights for observations. The same weights that were used when the model was built.

Output:

`varImp`	: A matrix of estimated variable importance. Rows correspond to input variables, columns correspond to criterion used. If argument `resultsEval` is not supplied or the `model` was not pruned, then 2nd, 3rd, and 4th columns are `NaN`.

Remarks:

The output argument as well as the printed table reports estimated input variable importance in the order as they appear in `Xtr`.

First column of the printed table shows indices of the input variables. This column is omitted in output argument `varImp`.

Column "delGCV" reports variable importance estimations calculated according to (Friedman, 1991b) Section 4.4:

> "*The relative importance of a variable is defined as the square root of the GCV of the model with all basis functions involving that variable removed, minus square root of the GCV score of the corresponding full model, scaled so that the relative importance of the most important variable (using this definition) has a value of 100.*"

The next three columns use criteria from (Milborrow, 2016) – see Section 12.3 in "Notes on the `earth` package" document. These columns are available only if `resultsEval` argument is supplied and the `model` was pruned.

Column "nSubsets" (Milborrow, 2016):

> "*[The criterion] counts the number of model subsets that include the variable. Variables that are included in more subsets are considered more important. [..] By "subsets" we mean the subsets of terms generated by the pruning pass. There is one subset for each model size (from 1 to the size of the selected model) and the subset is the best set of terms for that model size. [..] Only subsets that are smaller than or equal in size to the final model are used for estimating variable importance.*"

Column "subsRSS" (Milborrow, 2016):

> "*[The criterion] first calculates the decrease in the RSS for each subset relative to the previous subset. (For multiple response models, RSS's are calculated over all responses.) Then for each variable it sums these decreases over all subsets that include the variable. Variables which cause larger net decreases in the RSS are considered more important.*"

Column "subsGCV" (Milborrow, 2016):

> "*[This criterion] is the same, but uses the GCV instead of the RSS. Note that adding a variable can sometimes increase the GCV. (Adding the variable has a deleterious effect on the model, as measured in terms of its estimated predictive power on unseen data.) If that happens often enough, the variable can have a negative total importance, and thus appear less important than unused variables.*"

For ease of interpretation, all columns, except "nSubsets", are scaled so that the largest summed decrease is 100.

Note that the variance of the variable importance estimates can be high – different realizations of the data can give different estimates. Another possible approach, independent from ARESLab, would be to employ Random Forests or Bagging with regression trees, for example using the

M5PrimeLab toolbox for Matlab/Octave (Jekabsons, 2016) or using TreeBagger class from Matlab's Statistics and Machine Learning Toolbox.

2.14. Function `aresdel`

Purpose:

Deletes basis functions from ARES model, recalculates model's coefficients and relocates additional knots for piecewise-cubic models (as opposed to `aresanovareduce` which does not recalculate and relocate anything).

Call:

```
model = aresdel(model, funcsToDel, Xtr, Ytr, weights)
```

Input:

`model`	: ARES model or, for multi-response modelling, a cell array of ARES models.
`funcsToDel`	: A vector of indices for basis functions to delete. Intercept term is not indexed, i.e., the numbering is the same as in `model.knotdims`, `model.knotsites`, and `model.knotdirs`.
`Xtr, Ytr`	: Training data observations. The same data that was used when the model was built.
`weights`	: Optional. A vector of weights for observations. The same weights that were used when the model was built.

Output:

`model`	: Reduced ARES model.

2.15. Function `aresgetknots`

Purpose:

Gets all knot locations of an ARES model for the specified input variable. A knot is added to the list only if the variable entered a basis function non-linearly, i.e., using a hinge function.

Call:

```
knots = aresgetknots(model, variable)
```

For datasets with one input variable, only the first input argument is used. For datasets with more than one input variable, both input arguments are required.

Input:

`model`	: ARES model or, for multi-response modelling, a cell array of ARES models.
`variable`	: Index of the input variable.

Output:

`knots`	: Column vector of knot locations.

3. EXAMPLES OF USAGE

3.1. Ten-dimensional function with noise

We start by creating a dataset using a ten-dimensional function with added i.i.d. noise. The dataset consists of 200 observations randomly uniformly distributed in a ten-dimensional unit hypercube. The function actually uses only the first five variables.

```
X = rand(200,10);
Y = 10*sin(pi*X(:,1).*X(:,2)) + 20*(X(:,3)-0.5).^2 + ...
    10*X(:,4) + 5*X(:,5) + 0.5*randn(200,1);
```

Let's try a piecewise-cubic model (the default). We set the maximum number of basis functions to 21 (including the intercept term) and limit maximum interaction level to 2 (only pairwise products of basis functions will be allowed), leaving all the other parameters to their defaults.

For most applications, it can be expected that the most attention should be paid to the following parameters: `maxFuncs`, `c`, `cubic`, `maxInteractions`, and `maxFinalFuncs`. It is quite possible that the default values for `maxFuncs` and `maxInteractions` will be far from optimal for your data.

But note that, if you are prepared to use Cross-Validation, choosing a good value for `maxFinalFuncs` can sometimes release you from being too pedantic about parameters `maxFuncs` and `c`, because you can set large enough `maxFuncs` and not too large `c` and follow the example in Section 3.3.

If you have the necessary domain knowledge, it is recommended to also set `yesInteract`, `noInteract`, `allowLinear`, and `forceLinear`.

```
params = aresparams2('maxFuncs', 21, 'maxInteractions', 2);
```

ARES model is built by calling `aresbuild`. The function has three output arguments: the final model (`model`), algorithm execution time (`time`), and evaluations of best models of each size in the backward pruning phase (`resultsEval`). As the model building finishes, we can examine the data structure of the final model. It has 18 basis functions including the intercept term.

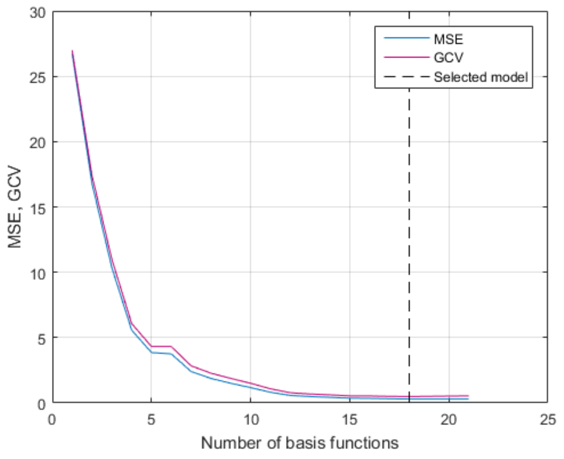
```
[model, ~, resultsEval] = aresbuild(X, Y, params)

model =
            MSE: 0.2741
            GCV: 0.4476
          coefs: [18x1 double]
       knotdims: {17x1 cell}
      knotsites: {17x1 cell}
       knotdirs: {17x1 cell}
        parents: [17x1 double]
    trainParams: [1x1 struct]
             t1: [17x10 double]
             t2: [17x10 double]
           minX: [1x10 double]
           maxX: [1x10 double]
       isBinary: [1x10 logical]
```

From the returned structure `model` we can find out what is its Mean Squared Error (`model.MSE`) and GCV (`model.GCV`) in the training data set. Field `model.coefs` gives us the list of all coefficients in the model (starting with the intercept). In `model.trainParams` we can see what were the training parameters for the method when the model was built, including the automatic choices. And if we want to extract knot locations, we can use `model.knotdims` and `model.knotsites` for that (or, alternatively, function `aresgetknots`).

Now let's plot, how MSE and GCV values changed during the iterations of the backward phase, i.e., evaluations of best models of each size. Typically, the two lines are close together at first, but, as the number of basis functions increases, GCV diverges from MSE and goes up. In our case, this is not so apparent because of small `maxFuncs`.

Alternatively, we could also create such plot for R^2 (Coefficient of Determination) and R^2_{GCV} (R^2 estimated by GCV) just by replacing `resultsEval.MSE` and `resultsEval.GCV` with `resultsEval.R2` and `resultsEval.R2GCV`.

```
figure;
hold on; grid on; box on;
h(1) = plot(resultsEval.MSE, 'Color', [0 0.447 0.741]);
h(2) = plot(resultsEval.GCV, 'Color', [0.741 0 0.447]);
numBF = numel(model.coefs);
h(3) = plot([numBF numBF], get(gca, 'ylim'), '--k');
xlabel('Number of basis functions');
ylabel('MSE, GCV');
legend(h, 'MSE', 'GCV', 'Selected model');
```



To assess input variable importance, use function `aresimp`. From the table, we can see that the 4th variable has the highest relative importance while the 3rd and 5th have the lowest (ignoring the last five variables not included in the model).

```
aresimp(model, X, Y, resultsEval);

Estimated input variable importance:
Variable    delGCV      nSubsets       subsRSS       subsGCV
1           74.975            15        62.407        63.789
2           71.245            14        38.002        39.637
3           32.772            12        13.393        14.395
4          100.000            16       100.000       100.000
5           34.466            13        19.900        21.046
6            0.000             0         0.000         0.000        unused
7            0.000             0         0.000         0.000        unused
8            0.000             0         0.000         0.000        unused
9            0.000             0         0.000         0.000        unused
10           0.000             0         0.000         0.000        unused
```

Let's take a look at ANOVA decomposition using function `aresanova`. (Note that ARESLab includes another function called `aresinfo` which is used for getting a table similar to ANOVA decomposition but with analysis of each separate basis function.)

From the table, we can see that the last ANOVA function (the one that is associated with the 1st and the 5th input variable) gives relatively small contribution (or even degrades performance) and maybe its basis functions should be deleted.

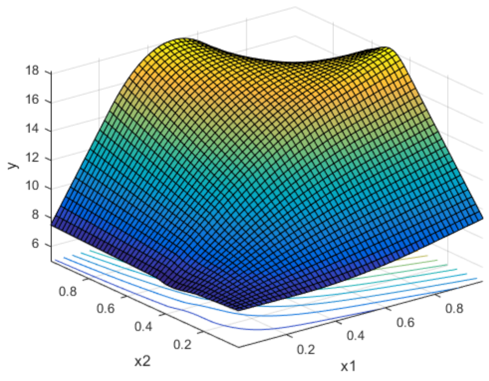
```
aresanova(model, X, Y);

Type: piecewise-cubic
GCV: 0.447615
R2GCV: 0.983416
Total number of basis functions (including intercept): 18
Total effective number of parameters: 43.5
ANOVA decomposition:
Function    STD          GCV        R2GCV     #basis     #params     variable(s)
1        3.1351       3.4082      0.87373        2          5.0      1
2        3.7505       3.3497      0.87590        2          5.0      2
3        1.2695       2.6547      0.90165        2          5.0      3
4        2.9888      12.9544      0.52005        2          5.0      4
5        1.5241       1.3505      0.94997        2          5.0      5
6        2.5585       2.5279      0.90634        5         12.5      1 2
7        0.2853       0.4463 !    0.98347        2          5.0      1 5
```

Let's say we want to delete the two basis functions comprising the 7th ANOVA function. Deletion of basis functions can be done using function `aresdel`. The function requires that we supply indices of the basis functions to delete. We can find those indices using `model.knotdims`. In our case they happen to be number 16 and number 17.

```
model = aresdel(model, [16 17], X, Y);
```

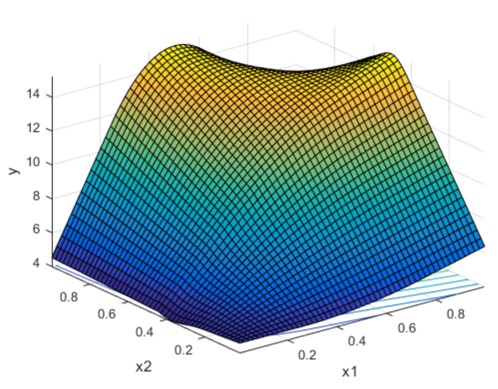
Function `aresplot` is used for plotting ARES models. Let's make a plot using the first two input variables. By default, `aresplot` fixes all other variables at the middle of their ranges (except for binary variables – they are fixed at their lowest values).

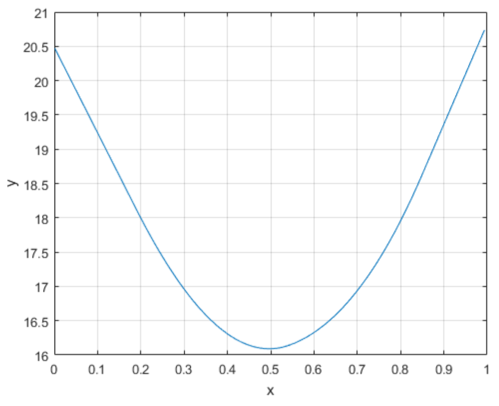
```
aresplot(model, [1 2]);
```

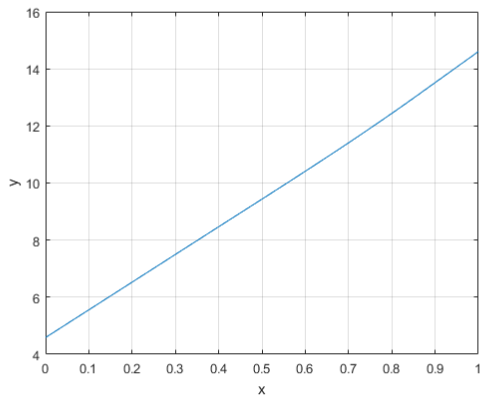


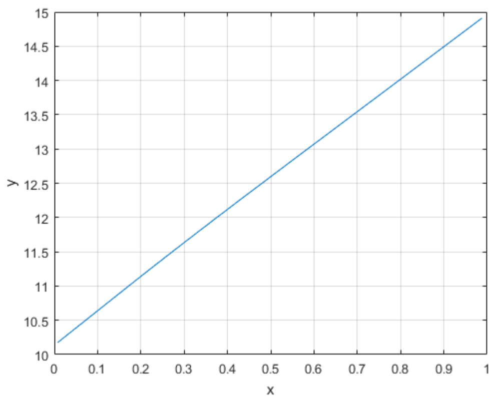
We can also plot ANOVA functions if they are associated with one or two input variables. This allows us to visualize the contributions of the ANOVA functions. Let's plot pair-wise (for variables x_1 and x_2; three ANOVA functions (one for each variable and one for the pair)) and individual (for variables x_3, x_4, and x_5; one ANOVA function each) contributions of variables.

```
modelReduced = aresanovareduce(model, [1 2]);
aresplot(modelReduced);

for i = 3 : 5
    modelReduced = aresanovareduce(model, i);
    aresplot(modelReduced, [], [], [], [], [], false);
end
```









In the last two plots, we can see that, although, in the model, variables x_4 and x_5 have two basis functions each, they actually may not need any knots, i.e., their true contributions could be completely linear. We could use parameters `allowLinear` or `forceLinear` of functions `aresparams` / `aresparams2` to encourage or force these variables to enter the model without hinge functions. This would result in simpler model with less possibilities to overfit the data.

To print the equation of the model with all its basis functions, use `areseq`. By default, basis functions of piecewise-cubic models are printed in their short form (see Equation 34 in Friedman, 1991a, for explanation) so that the output is more readable. But you can use input argument `cubicSmoothing` to make the function print full set of computations or print the model as piecewise-linear.

```
    areseq(model, 5);

    BF1 = C(x4|+1,0.36743,0.73434,0.86692)
    BF2 = C(x4|-1,0.36743,0.73434,0.86692)
    BF3 = C(x1|+1,0.24571,0.48679,0.74146)
    BF4 = C(x1|-1,0.24571,0.48679,0.74146)
    BF5 = C(x2|+1,0.43198,0.4357,0.46269)
    BF6 = C(x2|-1,0.43198,0.4357,0.46269)
    BF7 = C(x5|+1,0.16062,0.31343,0.65068)
    BF8 = C(x5|-1,0.16062,0.31343,0.65068)
    BF9 = C(x3|+1,0.18927,0.3774,0.54072)
    BF10 = BF3 * C(x2|+1,0.46269,0.48969,0.60606)
    BF11 = BF3 * C(x2|-1,0.46269,0.48969,0.60606)
    BF12 = BF4 * C(x2|+1,0.22187,0.42825,0.43198)
    BF13 = BF4 * C(x2|-1,0.22187,0.42825,0.43198)
    BF14 = C(x3|-1,0.54072,0.70405,0.84888)
    BF15 = BF3 * C(x2|+1,0.60606,0.72244,0.86076)
    y = 11.712 +10.887*BF1 -9.7031*BF2 +5.8423*BF3 -15.303*BF4 +11.774*BF5 -16.99*BF6
+4.7416*BF7 -5.0294*BF8 +14.642*BF9 -36.781*BF10 -4.8324*BF11 -23.387*BF12 +37.256*BF13
+12.46*BF14 -42.375*BF15
```

To evaluate performance of our ARES configuration (without the manual edits above, of course) using Cross-Validation, we'll use function `arescv`. By default, the function performs 10-fold Cross-Validation. Note that for more stable results one should consider repeating Cross-Validation several times (see description of the argument `nCross`).

```
rng(1);
resultsCV = arescv(X, Y, params)

resultsCV =
       MAE: 0.5089
       MSE: 0.4263
      RMSE: 0.6470
     RRMSE: 0.1304
        R2: 0.9826
    nBasis: 17.1000
     nVars: 5
    maxDeg: 2
```

Now let’s try piecewise-linear modelling.

```
params = aresparams2('maxFuncs', 21, 'maxInteractions', 2, 'cubic', false);
model = aresbuild(X, Y, params)

model =
            MSE: 0.2996
            GCV: 0.4893
          coefs: [18x1 double]
       knotdims: {17x1 cell}
      knotsites: {17x1 cell}
       knotdirs: {17x1 cell}
        parents: [17x1 double]
    trainParams: [1x1 struct]
           minX: [1x10 double]
           maxX: [1x10 double]
       isBinary: [1x10 logical]

rng(1);
resultsCV = arescv(X, Y, params)

resultsCV =
       MAE: 0.5763
       MSE: 0.5338
      RMSE: 0.7242
     RRMSE: 0.1459
        R2: 0.9782
    nBasis: 17.1000
     nVars: 5
    maxDeg: 2
```

Finally, we print the equation of the piecewise-linear model with all its basis functions (note that this time we did not delete basis functions 16 and 17).

```
areseq(model, 5);

BF1 = max(0, x4 -0.73434)
BF2 = max(0, 0.73434 -x4)
BF3 = max(0, x1 -0.48679)
BF4 = max(0, 0.48679 -x1)
BF5 = max(0, x2 -0.4357)
BF6 = max(0, 0.4357 -x2)
BF7 = max(0, x5 -0.31343)
BF8 = max(0, 0.31343 -x5)
BF9 = max(0, x3 -0.3774)
BF10 = BF3 * max(0, x2 -0.48969)
BF11 = BF3 * max(0, 0.48969 -x2)
BF12 = BF4 * max(0, x2 -0.42825)
BF13 = BF4 * max(0, 0.42825 -x2)
```

```
    BF14 = max(0, 0.70405 -x3)
    BF15 = BF3 * max(0, x2 -0.72244)
    BF16 = BF7 * max(0, x1 -0.86869)
    BF17 = BF7 * max(0, 0.86869 -x1)
    y  =  11.664  +10.758*BF1  -9.708*BF2  +8.5785*BF3  -13.385*BF4  +9.838*BF5  -15.279*BF6 
+6.2628*BF7  -4.3835*BF8  +13.418*BF9  -32.827*BF10  -13.016*BF11  -16.532*BF12  +32.427*BF13 
+11.22*BF14 -45.029*BF15 -46.087*BF16 -2.493*BF17
```

3.2. Two-dimensional function without noise

We start by creating training and test data using a two-dimensional noise-free function. The training data consists of 121 observations distributed in a regular 11×11 grid. The test data has 10000 observations distributed randomly.

```
clear
[X1,X2] = meshgrid(-1:0.2:1, -1:0.2:1);
X(:,1) = reshape(X1, numel(X1), 1);
X(:,2) = reshape(X2, numel(X2), 1);
clear X1 X2;
Y = sin(0.83*pi*X(:,1)) .* cos(1.25*pi*X(:,2));
Xt = rand(10000,2);
Yt = sin(0.83*pi*Xt(:,1)) .* cos(1.25*pi*Xt(:,2));
```

There is no noise and the data is plenty – we can hope for a very accurate model. We set the maximum number of basis functions to 101 (including the intercept term), no penalty for knots, and maximum interaction level equal to 2 (the number of input variables), leaving all the other parameters to their defaults. For noise-free data one could also consider decreasing `useMinSpan` and `useEndSpan` but for our dataset this would have little to no effect because each dimension in the dataset has only 11 distinct *x* values each repeated 11 times – by default, for datasets this small, the algorithm jumps over very few observations and therefore in most cases all of the available locations would be considered for knot placement anyway.

```
params = aresparams2('maxFuncs', 101, 'c', 0, 'maxInteractions', 2);
```

We build ARES model by calling `aresbuild`. Our model that has 48 basis functions including the intercept term.

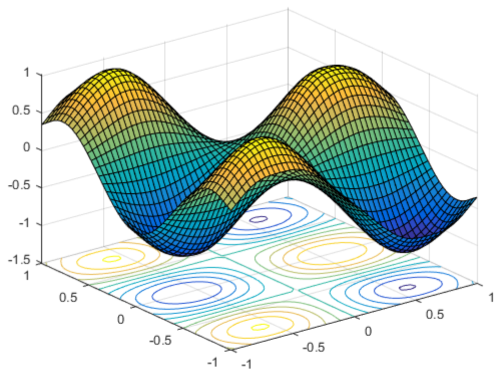
```
model = aresbuild(X, Y, params)

model = 
            MSE: 1.5734e-04
            GCV: 4.3227e-04
          coefs: [48x1 double]
       knotdims: {47x1 cell}
      knotsites: {47x1 cell}
       knotdirs: {47x1 cell}
        parents: [47x1 double]
    trainParams: [1x1 struct]
             t1: [47x2 double]
             t2: [47x2 double]
           minX: [-1 -1]
           maxX: [1 1]
       isBinary: [0 0]
```

Test the model using the test data.

```
results = arestest(model, Xt, Yt)

results =
      MAE: 0.0115
      MSE: 2.0000e-04
     RMSE: 0.0141
    RRMSE: 0.0253
       R2: 0.9994
```

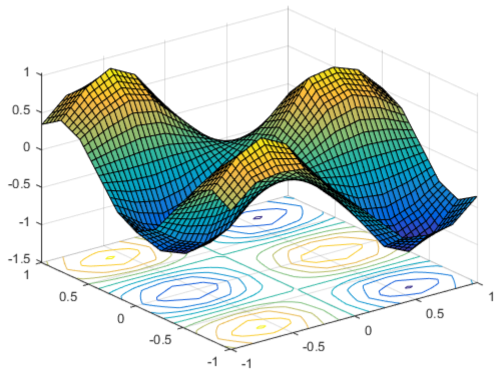
Plot the model.

```
aresplot(model);
```



Finally, let’s try doing the same but instead of piecewise-cubic modelling we'll do piecewise-linear.

```
params = aresparams2('maxFuncs', 101, 'c', 0, 'maxInteractions', 2, 'cubic', false);
model = aresbuild(X, Y, params);
results = arestest(model, Xt, Yt)

results =
      MAE: 0.0409
      MSE: 0.0023
     RMSE: 0.0482
    RRMSE: 0.0862
       R2: 0.9926

aresplot(model);
```



3.3. Using Cross-Validation to select the number of basis functions

One of the ways to select the “best” number of basis functions for final ARES model (i.e., argument `maxFinalFuncs` for functions `aresparams` / `aresparams2`) or to confirm that the GCV criterion is indeed making good choices, is to evaluate backward pruning phase's best candidate models of each size using Cross-Validation and compare which would be chosen by GCV and which by Cross-Validation.

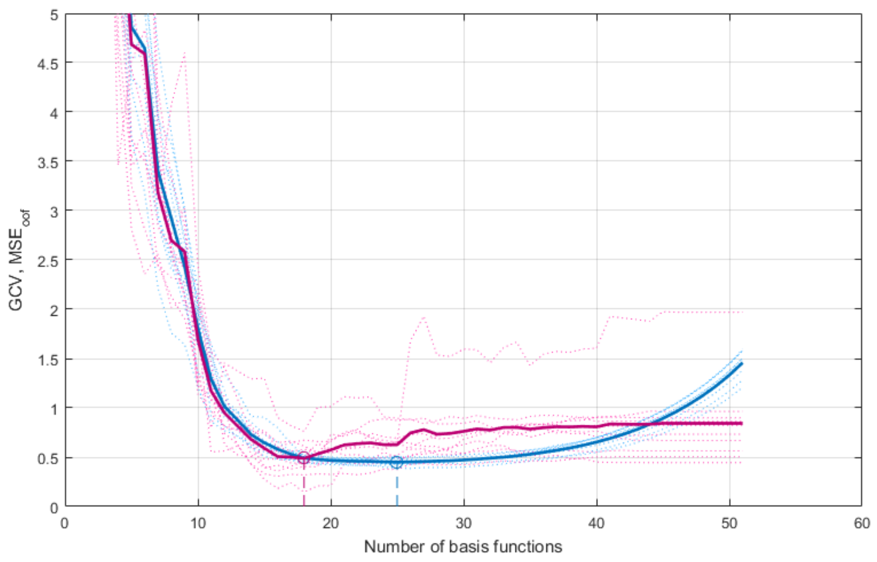
This can be done using function `arescv` by setting its argument `evalPruning` to `true`. In each Cross-Validation iteration, a new ARES model is built and pruned as usual using the GCV in the in-fold (training) data, but additionally, in the pruning phase, for the best candidate model of each size the function also calculates out-of-fold (test) data MSE (MSE_{oof}). Now for each model we have an estimate of prediction Mean Squared Error by both, in-fold GCV as well as out-of-fold MSE_{oof}.

Let's try this with the data from Section 3.1. We will use 10-fold Cross-Validation. Note that for more stable results one should consider repeating Cross-Validation several times (this can be set up using the argument `nCross`).

```
params = aresparams2('maxFuncs', 51, 'maxInteractions', 2);
rng(1);
[resultsTotal, resultsFolds, resultsPruning] = ...
    arescv(X, Y, params, [], [], [], [], [], true);
```

Here's code for plotting the results:

```
figure;
hold on; grid on; box on;
for i = 1 : size(resultsPruning.GCV,1)
plot(resultsPruning.GCV(i,:), ':', 'Color', [0.259 0.706 1]);
plot(resultsPruning.MSEoof(i,:), ':', 'Color', [1 0.259 0.706]);
end
plot(resultsPruning.meanGCV, 'Color', [0 0.447 0.741], 'LineWidth', 2);
plot(resultsPruning.meanMSEoof, 'Color', [0.741 0 0.447], 'LineWidth', 2);

ylim = get(gca, 'ylim');
posY = resultsPruning.meanGCV(resultsPruning.nBasisGCV);
plot([resultsPruning.nBasisGCV resultsPruning.nBasisGCV], [ylim(1) posY], '--', 'Color', [0 0.447 0.741]);
plot(resultsPruning.nBasisGCV, posY, 'o', 'MarkerSize', 8, 'Color', [0 0.447 0.741]);
posY = resultsPruning.meanMSEoof(resultsPruning.nBasisMSEoof);
plot([resultsPruning.nBasisMSEoof resultsPruning.nBasisMSEoof], [ylim(1) posY], '--', 'Color', [0.741 0 0.447]);
plot(resultsPruning.nBasisMSEoof, posY, 'o', 'MarkerSize', 8, 'Color', [0.741 0 0.447]);

xlabel('Number of basis functions');
ylabel('GCV, MSE_{oof}');
```



The ten blue dotted lines show the GCV for models of each fold. The blue solid line is the mean GCV for each model size (i.e., the average of the blue dotted lines). The ten pink dotted lines show the MSE_{oof} for models of each fold. The pink solid line is the mean MSE_{oof} for each model size (i.e., the average of the pink dotted lines).

The two vertical dashed lines are at the minimum of the two solid lines, i.e., they show the optimum number of basis functions estimated by GCV (blue) and Cross-Validation (pink). Ideally, the two vertical lines would coincide. In practice, they are usually close but not identical. In our case the two lines are at 25 (for GCV) and 18 (for MSE_{oof}). This information can be used to set the number of basis functions for final ARES model (`maxFinalFuncs`). But note that if the best number estimated by Cross-Validation is considerably larger than the best number estimated by GCV or if the best number is very near to the largest available, one should first consider allowing building more complex models, e.g., by decreasing GCV penalty per knot `c` and/or increasing `maxFuncs`.

Such statistics can also be generated for different values of `c` to see how the parameter influences the selection of the final model. Just call `arescv` once for each considered value of `c` and compare the graphs.

Finally, a note about piecewise-cubic models. Because all the aforementioned model evaluations are done in the actual backward pruning phase, by default they are calculated for piecewise-linear models even if in the end you are getting piecewise-cubic models. That is correct behaviour because by default all ARES models are first built as piecewise-linear and turned into piecewise-cubic only after the backward phase (Friedman, 1991a). Still, you can change this behaviour by setting `cubicFastLevel` for `aresparams` to 1 or 0.

3.4. Parameters `useMinSpan` and `useEndSpan`

In this section, we'll take a look at examples showing how important it can sometimes be to set your own values for `aresparams` parameters `useMinSpan` and `useEndSpan`.

The first dataset consists of 21 evenly distributed observations generated using sinus function and i.i.d. noise.

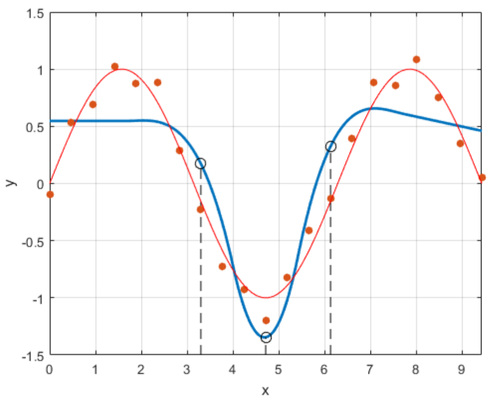
```
X = (0:0.05:1)' * pi * 3;
Y = sin(X) + randn(21,1) * 0.1;
Xsin = (0:0.01:1)' * pi * 3;
Ysin = sin(Xsin);
```

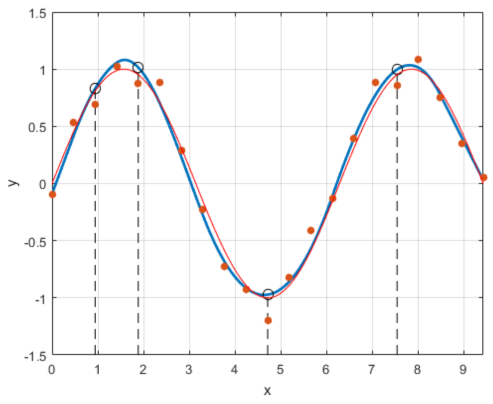
We'll build an ARES model using the default parameters. As can be seen in the first plot below, it does not model the data very well (red curve is the true function, blue curve is our model). Notice how all three knots are concentrated at the middle of the data range. In fact, with the default values for `useMinSpan` and `useEndSpan`, the three locations used by those knots are the only locations available to `aresbuild` for knot placement. This is because, for one-dimensional data of this size, the default values for those parameters are `useMinSpan` = 3 and `useEndSpan` = 7 meaning that for knot placement we have every 3rd location from 21 – 7 – 7 = 7.

```
params = aresparams2();
model = aresbuild(X, Y, params);
aresplot(model,[],[],[],[],[],[],'LineWidth',2,'XLim',[0,pi*3],'YLim',[-1.5,1.5]);
hold on; plot(X, Y, '.', 'MarkerSize', 20); plot(Xsin, Ysin, '-r');
```

Because of the noise in the data, it could be risky to turn `useMinSpan` and `useEndSpan` off completely (the bigger the noise, the bigger the risk in lowering those values). Let's set them both to 2. Now the algorithm will have 9 locations for knot placement (every 2nd location from 21 – 2 – 2 = 17). As can be seen in the second plot below, that made the model considerably better.

```
params = aresparams2('useMinSpan', 2, 'useEndSpan', 2);
model = aresbuild(X, Y, params);
aresplot(model,[],[],[],[],[],[],'LineWidth',2,'XLim',[0,pi*3],'YLim',[-1.5,1.5]);
hold on; plot(X, Y, '.', 'MarkerSize', 20); plot(Xsin, Ysin, '-r');
```





The second dataset consists of 21 evenly distributed observations generated using a step function without any noise. With accurate knot locations, we should be able to model it perfectly.

```
X = (0:0.05:1)';
Y = [ones(1,7)*3 ones(1,7) ones(1,7)*2]';
```

We'll build an ARES model using the default parameters, except that we don't need piecewise-cubic modelling. As can be seen in the first plot below, it uses just one knot location in the middle of data range and therefore can't model the data very well.

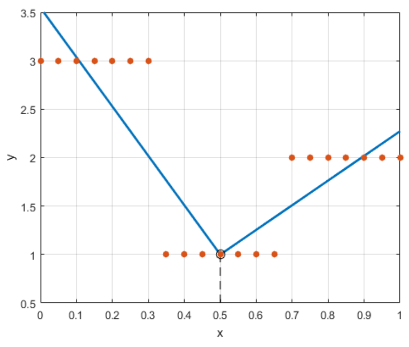
```
params = aresparams2('cubic', false);
model = aresbuild(X, Y, params);
aresplot(model, [], [], [], [], [], [], 'LineWidth', 2, 'YLim', [0.5, 3.5]);
hold on; plot(X, Y, '.', 'MarkerSize', 20);
```

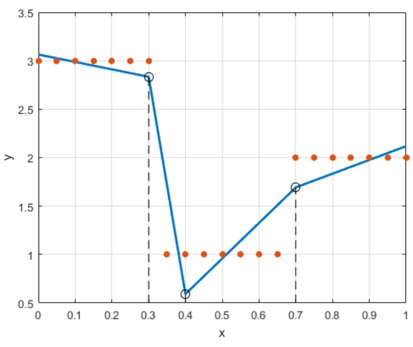
Setting `useMinSpan` and `useEndSpan` to 2, like for the previous dataset, is still not enough, because, while the first and the last knot is placed correctly, the algorithm jumps over the other needed knot locations. See the second plot below.

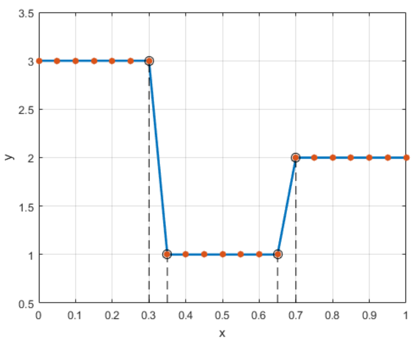
```
params = aresparams2('cubic', false, 'useMinSpan', 2, 'useEndSpan', 2);
model = aresbuild(X, Y, params);
aresplot(model, [], [], [], [], [], [], 'LineWidth', 2, 'YLim', [0.5, 3.5]);
hold on; plot(X, Y, '.', 'MarkerSize', 20);
```

Let's set `useMinSpan` to 1 (effectively turning it off) so that every location can be considered for knot placement. Now the data is modelled perfectly (see the third plot below). Note that actually it is enough to set `useEndSpan` to 6 because the first and the last six observations can be safely ignored.

```
params = aresparams2('cubic', false, 'useMinSpan', 1, 'useEndSpan', 6);
model = aresbuild(X, Y, params);
aresplot(model, [], [], [], [], [], [], 'LineWidth', 2, 'YLim', [0.5, 3.5]);
hold on; plot(X, Y, '.', 'MarkerSize', 20);
```







4. REFERENCES

1. Friedman J.H. Multivariate Adaptive Regression Splines (with discussion), The Annals of Statistics, Vol. 19, No. 1, 1991a
2. Friedman J.H. Estimating functions of mixed ordinal and categorical variables using adaptive splines, Technical Report No. 108, Laboratory for Computational Statistics, Department of Statistics, Stanford University, 1991b
3. Friedman J.H. Fast MARS, Department of Statistics, Stanford University, Tech. Report LCS110, 1993
4. Hastie T., Tibshirani R., Friedman J. The elements of statistical learning: Data mining, inference and prediction, 2nd edition, Springer, 2009
5. Jekabsons G., M5PrimeLab: M5' regression tree, model tree, and tree ensemble toolbox for Matlab/Octave, 2016, available at http://www.cs.rtu.lv/jekabsons/
6. Merkwirth C. and Wichard J. A Matlab toolbox for ensemble modelling, 2003, available at http://www.j-wichard.de
7. Milborrow S., Earth: Multivariate Adaptive Regression Spline Models (derived from code by T. Hastie and R. Tibshriani), 2016, R package available at http://cran.r-project.org/src/contrib/Descriptions/earth.html
8. Rudy J., py-earth: A Python implementation of Jerome Friedman's Multivariate Adaptive Regression Splines, 2016, available at https://github.com/jcrudy/py-earth