MATLAB Library Documentation

Multidimensional Time Series Toolbox

Institute for Automation

Department Product Engineering

Montanuniversität Leoben

Thomas Grandl,
Roland Ritt

Version: 1.2
September 25, 2019

Contents

1 Introduction

1.1 Download

This toolbox can be downloaded from the IA-GitLab Server.

- Web-Url: `https://git.unileoben.ac.at/Chair-Of-Automation/IA-toolboxes/mdtsToolbox`
- Git-Link: `git@git.unileoben.ac.at:Chair-Of-Automation/IA-toolboxes/mdtsToolbox.git`

1.2 Purpose of this Toolbox

This toolbox provides tools for handling time series in multi dimensions. It implements a data storage for these time series together with the according metadata. Various functions and methods provide tools for the extraction and extension, as well as the initialization, analysis and manipulation of the data.

1.3 Toolbox Structure

The structure of the toolbox is illustrated in figure 1.1.

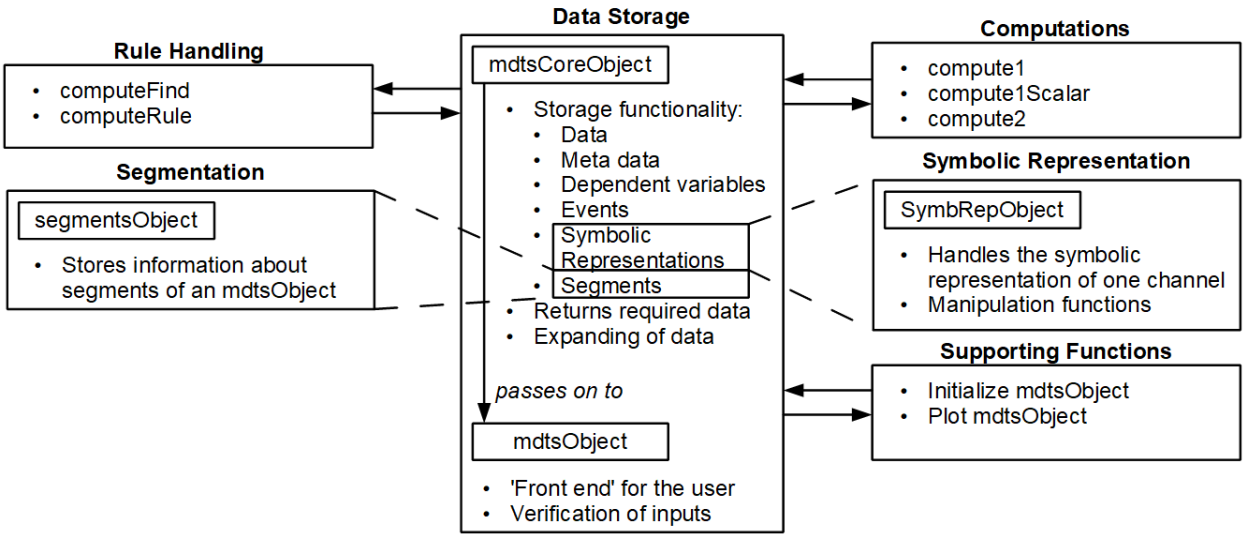

Figure 1.1: Structure of the toolbox

1.3.1 Data Storage

The data storage is implemented as two classes, the *mdtsCoreObject* Class and the *mdtsObject* Class.

The mdtsCoreObject represents the actual data storage. It holds all time series (data) as well as additional information about these series (metadata). It also provides functions to extract specific parts of the data as well as to extend the data set by a new time series. Furthermore, events and symbolic representations of channels can be added to the mdtsCoreObject.

The mdtsObject is a "wrapper class" which inherits from the mdtsCoreObject. It represents the interface between the data storage and the user or computational functions, respectively. It has the same functions as the core object. This way it validates the inputs and passes them to the actual functions of the mdtsCoreObject.

The user is expected to only use the mdtsObject, direct usage of the mdtsCoreObject is restricted to internal processes for faster execution!

1.3.2 Symbolic Representation

The data of every channel can be discretized and represented through symbols. Such a symbolic representation is implemented as separate object. If a channel is represented symbolically, a *SymbRepObject* is generated and assigned to this channel. The mdtsCoreObject holds the handles to every available SymbRepObject and the assignment to the according channel. Beside the representation of the data, the SymbRepObject provides a number of functions to analyse and manipulate the symbolic representation of a channel.

1.3.3 Computations

A set of functions is available to execute various computations.

1.3.4 Rule Handling

A set of functions to find specific states within the data and to combine or superpose states with rules.

1.3.5 Segmentation

The data of a mdtsObject can be devided into segments. Such a segment can for example represent the occurrence of certain events. One instance of such a segment is represented by its start time and the duration together with the name of the segment. For example, if a mdtsObject contains data of a certain machine, one segmentation could represent the running times of this machine, where one instance of this segmentation represents the start when the machine is turned on and the time it is running. The segmentation and the *segmentsObject*, respectively, can followingly include multiple of such instances with different start times and durations.

1.3.6 Supporting Functions

Further functions are available to initialize and inspect (plot) the data.

1.4 Dependencies

The mdtsToolbox requires the following IA-toolboxes in the MALTAB-path:

1. IAToolboxes/DOPbox: Git-Link `git@git.unileoben.ac.at:Chair-Of-Automation/IA-toolboxes/DopBox.git`
2. IAToolboxes/graphics: Git-Link `git@git.unileoben.ac.at:Chair-Of-Automation/IA-toolboxes/graphics.git`

3. IAToolboxes/general: Git-Link `git@git.unileoben.ac.at:Chair-Of-Automation/IA-toolboxes/general.git`
4. IAToolboxes/figureManager: Git-Link `git@git.unileoben.ac.at:Chair-Of-Automation/IA-toolboxes/figureManager.git`
5. IAToolboxes/summaryTable: Git-Link `git@git.unileoben.ac.at:Chair-Of-Automation/IA-toolboxes/summarytable.git`

2 Class and Function Description

2.1 Data Storage

2.1.1 mdtsCoreObject

Properties

1. Core data:
   a) **time:** n x 1 vector of time stamps. Can be given as datenum, datetime, duration or any arbitrary numerical vector
   b) **data:** n x m matrix of values
   c) **tags:** 1 x m vector of series names as strings (character vectors)
   d) **tsEvents:** Map which holds all events. Keys are event IDs as character vector (string) and values are structs, where value.eventTime represents an array of all time stamps when the event starts and value.eventDuration represents an array of all durations of the event as number of time stamps. Furthermore, the user can add specific data to every event in the field value.userData.
   e) **symbReps:** Cell array which holds all symbolic representations. The representation of a channel is stored in the cell array as the element with the index according to the channel or tag index. Further descriptions in section 2.2.1.
   f) **segments:** Variable which holds a segmentsObject, compare section 2.5.1.
   g) **aliasTable:** A table which holds the aliases for tags. The row names are the aliases, and the column *OrigTag* holds the name of the original tag. If you request a tag, it first searches through this table, and after this, it searches in the tags itself.
2. Meta data:
   a) **name:** name of the object/time series
   b) **who:** creator of the object/time series
   c) **when:** creation date of the object/time series as string
   d) **description:** description to the object/time series
   e) **comment:** comment to the object/time series
   f) **units:** units of the tags
   g) **uniform:** ture if the time steps are spaced uniformly
   h) **ts:** sampling time
   i) **isSubset:** true if the object was derived by extraction from another mdtsObject
3. Dependent data:
   a) **fs:** sampling frequency
   b) **timeRelative:** relative time stamps in which the first stamp starts at zero

c) **timeDateTime:** time stamps in datetime format

d) **timeDateTimeRelative:** relative time stamps in which the first stamp starts at zero in datetime format

e) **nChannels:** number of Channels in the mdts-Object (number of Columns)

f) **nDataPoints:** number of DataPoints (time-stamps); this is the number of rows

The structure of these properties within the mdtsCoreObject is illustrated in figure 2.1.

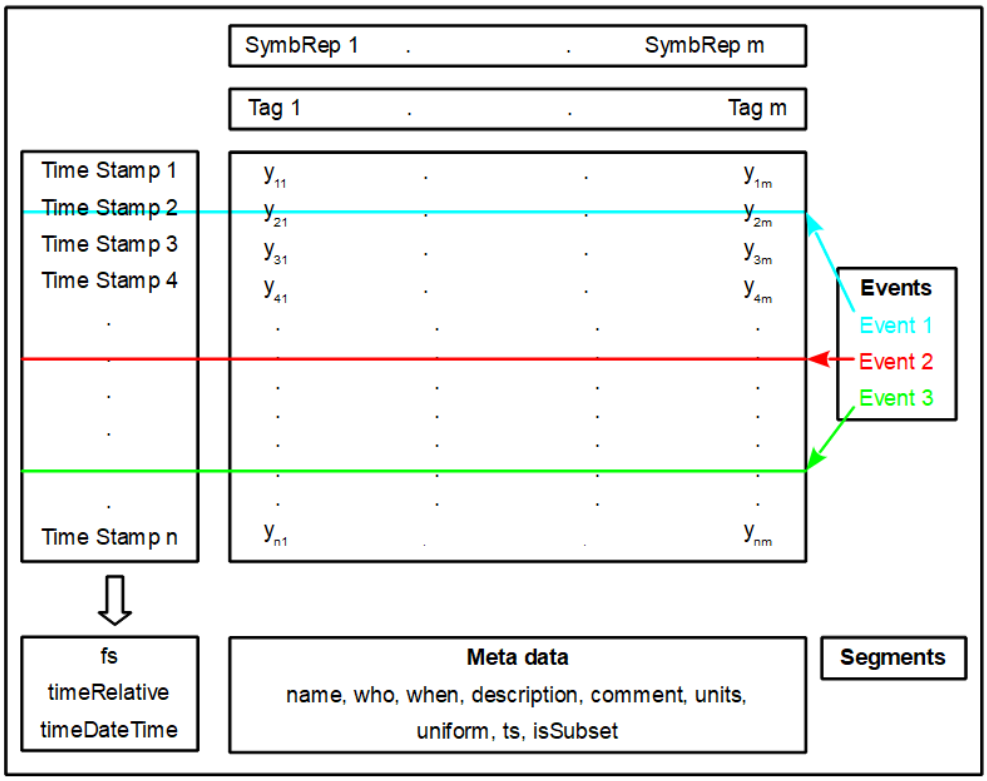

Figure 2.1: Structure of the properties within the mdtsCoreObject

Constructor

obj = mdtsCoreObject(time, data, tags, units, ts, name,
who, when, description, comment, tsEvents, symbReps, segments, aliasTable)

Inputs must be given according to the properties of the object.

Methods

1. **returnObject = getData**
   Returns a new object with extracted data from the original object. The following function calls are available:

   returnObject = getData(tags)
   returnObject = getData(tags, timeInterval)

where *tags* is a string which represents the name of the time series or a 1 x m cell array of strings which represent the names of multiple time series and *timeInterval* is a 2 x 1 array of time stamps as datenum which define start and end of the time interval, e.g.

```
tags = {'Channel1', 'Channel2', 'Channel3'};
timeInterval = [datenum(datetime(2017, 7, 31, 14, 3, 3, 123));
                datenum(datetime(2017, 7, 31, 14, 3, 3, 123 + 5))];
returnObject = originalObject.getData(tags, timeInterval);
```

2. **dataMat = getRawData**
   Returns the data from the mdtsObject as specified in the input parameters. The following function calls are available:

   dataMat = getRawData(tags)
   dataMat = getRawData(tags, timeInterval)

   where *tags* is a string which represents the name of the time series or a 1 x m cell array of strings which represent the names of multiple time series, *timeInterval* is a 2 x 1 array of time stamps as datenum which define start and end of the time interval and *dataMat* is the extracted matrix from the data of the mdtsObject, e.g.

```
tags = {'Channel1', 'Channel2', 'Channel3'};
timeInterval = [datenum(datetime(2017, 7, 31, 14, 3, 3, 123));
datenum(datetime(2017, 7, 31, 14, 3, 3, 123 + 5))];
dataMat = originalObject.getRawData(tags, timeInterval);
```

3. **tagIndices = getTagIndices(tagList)**
   Returns indices of the tags given in *tagList*. tagList must be given as string which represents the name of the time series or as 1 x m cell array of strings wich represent the names of multiple time series, e.g.

```
tagList = {'Channel1', 'Channel2', 'Channel3'};
tagIndices = originalObject.getTagIndices(tagList);
```

4. **intervalIndices = getIntervalIndices(timeInterval)**
   Returns indices of the time interval given in *timeInterval*. timeInterval must be given as 2 x 1 array of time stamps as datenum which define start and end of the time interval, e.g.

```
timeInterval = [datenum(datetime(2017, 7, 31, 14, 3, 3, 123));
                datenum(datetime(2017, 7, 31, 14, 3, 3, 123 + 5))];
intervalIndices = originalObject.getIntervalindices(timeInterval);
```

5. **correctTags = checkTags(tagList)**
   Checks if the tags given in *tagList* are available within the data set. Returns a boolean which is true when tags are available, otherwise false. tagList must be given as string which represents the name of the time series or as 1 x m cell array of strings which represent the names of multiple time series, e.g.

```
tagList = {'Channel1', 'Channel2', 'Channel3'};
correctTags = originalObject.checkTags(tagList);
```

6. **[startDateNum, endDateNum] = startEndOfDate(dateString)**
   Returns the first and the last available time stamp of a time span as datenum. The time span must be given in *dateString* as string (see example below). For example, if the data contains one time stamp every day from January 1st, 2015 to December 31st, 2017 and *dateString* is "2016", then *startDateNum* is the datenum of January 1st, 2016 and endDateNum is the datenum of December 31st, 2016. Possible values for dateString are given in the following example:

```
dateString1 = '25-Jul-2017 14:03'; % 14:03:00 to 14:03:59 on 7/25/2017
dateString2 = '25-Jul-2017 2pm'; % 14:00:00 to 14:59:59 on 7/25/2017
dateString3 = '25-Jul-2017'; % 00:00:00 to 24:59:59 on 7/25/2017
dateString4 = 'Aug-2017'; % 00:00:00 on 8/1/2017 to 24:59:59 on 8/31/2017

[startDateNum1, endDateNum1] = originalObject.startEndOfDate(dateString1);
[startDateNum2, endDateNum2] = originalObject.startEndOfDate(dateString2);
[startDateNum3, endDateNum3] = originalObject.startEndOfDate(dateString3);
[startDateNum4, endDateNum4] = originalObject.startEndOfDate(dateString4);
```

7. **obj = expandDataSet(addData, addTags)**
   Expands the data set of the object by the (multiple) time series given in *addData* and assigns these new series the tags given in *addTags*. addData must be given as n x m matrix of data values and addTags as string which represents the name of the time series or as 1 x m cell array of strings which represent the names of multiple time series, e.g.

```
addTags1 = 'Channel1';
addData1 = [3;
            4;
            5];
addTags2 = {'Channel1', 'Channel2'};
addData2 = [3, 6;
            4, 7;
            5, 8;
            6, 9];
addTags3 = {'Channel1', 'Channel2', 'Channel3'};
addData3 = [3, 6, 10;
            4, 7, 11;
            5, 8, 12;
            6, 9, 13];
            7, 10, 14];

originalObject1 = originalObject1.expandDataSet(addData1,addTags1);
originalObject2 = originalObject2.expandDataSet(addData2,addTags2);
originalObject3 = originalObject3.expandDataSet(addData3,addTags3);
```

```
originalObject4 = originalObject4.expandDataSet(addData4,addTags4);
```

8. **Direct Indexing**
   It is also possible to extract a part from the mdtsCoreObject as

$$\text{newmdtsCoreObject} = \text{originalmdtsCoreObject}(i:j, k:l)$$

   where i, j are the time indices and k, l are the tag indices, e.g.

```
newObject = originalObject(2:4, 1:3); % time stamps 2 - 4, tags 1 - 3
newObject = originalObject(1:4, 2:3); % time stamps 1 - 4, tags 2 - 3
newObject = originalObject(4, 1:3); % time stamp 4, tags 1 - 3
newObject = originalObject(2:5, 2); % time stamps 2 - 5, tags 3
```

9. **obj = addEvent(eventID, eventTime, eventDuration)**
   Adds an event to *tsEvents* such that $tsEvents(key) = value$ with the key *eventID* and a struct as value with the two entries *eventTime* as value.eventTime, as well as *eventDuration* as value.eventDuration.

10. **obj = addSymbRepToChannel(channelNumber, symbObj)**
    Adds the symbRepObject *symbObj* to *symbReps* as element with index *channelNumber*.

11. **obj = addSymbRepToAllChannels(symbObj, keepExistentSymbReps)**
    Adds the symbRepObject *symbObj* to all elements of *symbReps*. If *keepExistentSymbReps* is true, already existent symbRepObjects within *symbReps* will be preserved, if set to false, all elements in *symbReps* will be overwritten.

12. **obj = addSegments(segmentsObj)**
    Stores the segmentsObject *segmentsObj* in the *segments* property of the mdtsObject.

13. **obj = addSegmentsToAllChannels(segmentsObj, keepExistentSegReps)**
    Assigns the segmentsObject *segmentsObj* to all channels in the *segments* property of the mdtsObject. If *keepExistentSegReps* is true, already existent segmentsObjs within *segments* will be preserved, if set to false, all elements in *segments* will be overwritten.

14. **obj = addSegmentsToChannels(segmentsObj, channelNumber)**
    Assigns the segmentsObject *segmentsObj* to *segments* the property of the mdtsObject for only the given channelNumber(s).

Methods (Protected)

These functions can only be called within the class or a subclass.

1. **obj = keepRowsOfData(intervalIndices)**
   Delete all data (rows) of time stamps which are not given in *intervalIndices*. *intervalIndices* are the numeric indices of the rows to be kept.

2. **obj = keepTagsOfData(tagsIndices)**
   Delete all data (columns) oftags which are not given in *tagsIndices*. *tagsIndices* are the numeric indices of the columns to be kept.

3. **extractRowsOfSymbReps(intervalIndices)**
   Extract all symbols and druations from the symbRepObjects of all channels, according to the input *intervalIndices*. *intervalIndices* are the numeric indices of the rows to be kept. The symbReps are overwritten with the trimmed version.

4. **extractRowsOfSegsReps(intervalIndices)**
   Extract all segments and druations from the segmentsObject of all channels, according to the input *intervalIndices*. *intervalIndices* are the numeric indices of the rows to be kept. The segments are overwritten with the trimmed version.

5. **extractIntervalOfEvents(intervalIndices)**
   Extract all events from the eventsStruct of all channels, according to the input *intervalIndices*. *intervalIndices* are the numeric indices of the rows to be kept. The events are overwritten with the trimmed version. ATTENTION: This method is not yet implemented.

6. **tags = alias2tag(tags)**
   returns the tag names which is assigned to the *aliases*. *tags* are the original tag names which are assigned to the aliases.

2.1.2 mdtsObject

As mentioned in section 1.3.1, the mdtsObject inherits from the mdtsCoreObject. Thus, this object has the same properties available as the mdtsCoreObject in addition to the properties stated here.

Properties

1. Internal data:
   Note: Internal data is only for the usage in one object and for the communication between different objects. These are no properties the user should deal with!

   a) **timeType:** Scalar which holds the information about the original time format which was used to initialize the object. Possible variants are datetime, duration, datenum or other numerical representations. This property will be set by the constructor of the mdtsObject automatically, according to the type of the given time vector.

      i. **timeType == 0, timeType == 1**: the time was given as numeric values; and are treated like this → no datetime representation is used for plotting etc.

      ii. **timeType == 2**: the time was given as datetime values; and are treated like this in plotting

      iii. **timeType == 3**: the time was given as duration values; and are treated like this in plotting

2. Dependent data:

   a) **timeInFormat:** Time stamps of the object in the format according to the original format which was used to initialize the object.

Constructor

obj = mdtsObject(time, data, tags, varargin)

where *time*, *data* and *tags* are required inputs and *varargin* represents all optional inputs. Required inputs must be given according to the porperties of the mdtsCoreObject (compare section

2.1.1). Optional inputs can be given as tuples. The available optional inputs are summarized in table 2.1.

Table 2.1: Available optional inputs for mdtsObject constructor

Input	Description	Default Values
'units'	Units of the corresponding tags	'-'
'ts'	Sampling time	[]
'name'	Name of the data set	'Time Series'
'who'	Creator of the object	'Author'
'when'	Date when the object was created	'Now'
'description'	Description to the object	'No description available'
'comment'	Comment to the object	'No comment available'
'tsEvents'	Events within the data	Empty map
'symbReps'	Symbolic representations	Empty cell array
'segments'	Segments of the mdtsObject	Empty cell array
'aliasTable'	aliases for the tags	Emty table with column 'OrigTag'

Methods

1. **returnObject = getData**
   Validates the inputs and passes these inputs to the *getData* function of the mdtsCoreObject, compare section 2.1.1.

2. **dataMat = getRawData**
   Validates the inputs and passes these inputs to the *getRawData* function of the mdtsCore-Object, compare section 2.1.1.

3. **tagIndices = getTagIndices(tagList)**
   Validates the inputs in *tagList* and passes them to the *getTagIndices* function of the mdtsCore-Object, compare section 2.1.1.

4. **intervalIndices = getIntervalIndices(timeInterval)**
   Checks if the given time interval lies within the bounds of the time stamps of the object and passes them to the *getIntervalIndices* function of the mdtsCoreObject, compare section 2.1.1.

5. **obj = expandDataSet(addData, addTags)**
   Validates the inputs and passes them to the *expandDataSet* function of the mdtsCoreObject, compare section 2.1.1.

6. **Direct Indexing**
   Furthermore its possible to extract data from the mdtsObject as

$$\text{newObject} = \text{originalObject}(i : j, k : l)$$

   where i, j are the time indices and k, l are the tag indices, e.g.

```
newObject = originalObject(2 : 4, 1 : 3); % time stamps 2 - 4, tags
    1 - 3
newObject = originalObject(1 : 4, 2 : 3); % time stamps 1 - 4, tags
    2 - 3
newObject = originalObject(4, 1 : 3); % time stamp 4, tags 1 - 3
```

```
newObject = originalObject(2 : 5, 2); % time stamps 2 - 5, tag 2
newObject = originalObject(:, 2); % all time stamps , tag 2
newObject = originalObject(2 : 5, :); % time stamps 2-5 , all tags
```

additional tags/alias names can be used for indexing

```
newObject = originalObject(:,{'tag1', 'tag3'}); % all data
    extracted for channels named with 'tag1' and 'tag3'
newObject = originalObject(:,{'tag1', 'alias3'}); % all data
    extracted for channels named with 'tag1' and the alias 'alias3'
```

In case a datetime vector was used for generating the mdtsObject, datetimes can be used for row-indexing

```
newObject = originalObject({datetime(2019,8,12), datetime
    (2019,9,12) },:); % all data extracted between 2019-08-12 and
    2019-09-12 (including start and end date),
```

7. **obj = addEvent(eventID, eventTime, eventDuration)**
   Validates if *eventID* is a character vector (string), *eventTime* is an array of time stamps available in *time* and *eventDuration* is an array of integers or durations and of the same size as *eventTime*. Valid values will be passed to the *addEvent* function of the mdtsCoreObject, compare section 2.1.1.

8. **obj = addSymbRepToChannel(channelNumber, symbObj)**
   Validates if *symbObj* is a symbRepObject and *channelNumber* is numeric and passes valid parameters to the *addSymbRepToChannel* function of the mdtsCoreObject, compare section 2.1.1.

9. **obj = addSymbRepToAllChannels(symbObj, varargin)**
   Validates if *symbObj* is a symbRepObject and passes valid parameters to the *addSymbRepToAllChannels* function of the mdtsCoreObject, compare section 2.1.1. Optional parameter *keepExistentSymbReps* can be passed as logical scalar through a name-value-pair.

10. **obj = addSegmentsToAllChannels(segmentsObj, varargin)**
    Validates if *segmentsObj* is a segmentsObj and *nTimestamps* in *segmentsObj* is the same as the number of elements in *time* of the mdtsObject and passes valid parameters to the *addSegmentsToAllChannels* function of the mdtsCoreObject, compare section 2.1.1.

11. **obj = addSegments(segmentsObj)**
    Validates if *segmentsObj* is a segmentsObj and *nTimestamps* in *segmentsObj* is the same as the number of elements in *time* of the mdtsObject and passes valid parameters to the *addSegments* function of the mdtsCoreObject, compare section 2.1.1.

12. **obj = addSegmentsToChannels(segmentsObj, channelNames)**
    Validates if *segmentsObj* is a segmentsObj and *nTimestamps* in *segmentsObj* is the same as the number of elements in *time* of the mdtsObject and passes valid parameters to the *addSegmentsToChannels* function of the mdtsCoreObject, compare section 2.1.1. *channelNames* is a cell array of tags or aliases to which the segments should be assigned.

13. **timeDateNum = convert2Datenum(timeInput)**
    Converts a time in *timeInput* from datetime or duration to datenum (if *timeInput* is given as datenum, then *timeDateNum* = *timeInput*).

14. **bValid = isValidAliasTable(tIn)**
    This function checks, if a table *tIn* is a valid aliasTable with respect to the tags of the mdtsObject *obj.tags*. It checks if the table has exactly one column *OrigTag* and if the entries in this column can be found in *obj.tags*.

15. **isTagList = isTag(tags)**
    This function checks, if the given *tags* can be found within the taglist of the object or within the aliasTable. It returns a logical array of the same size as *isTagList* indicating if a tag is can be found and used for indexing.

16. **obj = setAliasTable(tIn)**
    This function assignes a new aliasTable *tIn* to the mdtsObject. All aliases defined before are deleted in this manner.

17. **isInList = isAlias(tags)**
    This function checks if the given *tags* are aliases defined in the aliasTable of the instance. A logical array *isInList* of the same size as *tags* is returned, indicating which tags are aliases.

18. **obj = addAliases(aliases, tags)**
    This function adds new aliases to the aliasTable. aliases is a list of new names which can be used to index the mdtsObject. Tags is the original tag name which is used to index the mdtsObject.

19. **obj = markRangeOnAxes_givenIndex(axes_in, startInds, stopInds, colorSpec, varargin)**
    This function marks a given range as a semitransparent patch. It forwards the input parameters to the Static-Method *markRangeOnAxes_givenIndexStatic* and takes the time values from the mdtsObject. (see *mdtsOjbect* 2.1.2 Methods (Static))

20. **arrayOfmdtsObjects = extractSegments(segmentTag, tagName, varargin)**
    This functions extracts segments from the segments structure and returns them as mdtsObjects (cell array of mdtsObjects *arrayOfmdtsObjects*). *segmentTag* is the name of the Segment which should be extracted. *tagName* is the name of the channel to which the *segmentTag* is assigned. *frameLength* (optional key-value pair; default value = 0) defining how much data points should be additionally returned to the beginning and at the end of each segment.

21. **summaryTable = getSummaryTable**
    This functions returns a *summaryTable* for the given mdtsObject. Therefore, the according toolbox (*summaryTable*) need to be on the matlab path.

22. **[axesH, fM, ph] = plot(varargin)**
    This functions plots the mdtsObject (stacked plot with Events and Symbolic Time Series represented as Semi-Transparent patches). The optional input arguments (key-value pairs) are:

    a) **Size**: the size (height, width) of the plot im centimeters. Default Value: [8.8cm, 11.7cm]

    b) **FontSize**: the font Size in pt used for figure. Default value: 10pt

    c) **plotSymbolName**: boolean flag indicating if the symbol should be plotted on the semi transparent patches (if a SymbolicRepresentation is defined). Default: false

    d) **plotSymbolDuration**: boolean flag indicating if the symbol duration should be plotted on the semi transparent patches (if a SymbolicRepresentation is defined). Default: false

e) **plotSymbolNameMinLengthRelative**: the minimum length of a symbolic representation to show the symbol name and duration defined as a relative value of the entire length of the time series. Default: 0.05; this can be used to avoid clattering of the plot in case of short symbols.

f) **figureH**: a handle to a figure which should be used for plotting. Default: a new figure is created with *figureGen* of the graphics-toolbox.

g) **additional key-value pairs**: are forwarded to the plotMulti function of the graphics-toolbox.

returned are:

a) **axesH**: the handles to the axes

b) **fM**: the figureManager used on the figure to handle large data sets (figureManager-toolbox)

c) **ph**: the handles to the plotted timeseries (handles to the line objects).

23. **ph = plotEventsOnAxes(axesIn)**
This function plots the event assigned to the mdtsObject on given axes (*axesIn* as a dashed line at the point of occurence. *axesIn* are the axes on which the Events should be plotted. *ph* are the handles to the event-lines (line-objects).

24. **[axesOut, fM, ph] = plotmdtsObject(inputObject, varargin)**
is a deprecated method which currently forwards *varargin* to the *plot* function of *mdtsObject*.

25. **[axesOut, fM, ph] = plotSegments(segmentTag, varargin)**
Will be deprecated in future. This function generates a new plot of the multidimensional time series where the assigned segments are marked as semi-transparent patches. The input arguments are:

a) **segmentTag**: the name of the segment which should be plotted

b) **Size**: optional; the size (height, width) of the plot im centimeters. Default Value: [8.8cm, 11.7cm]

c) **FontSize**: optional; the font Size in pt used for figure. Default value: 10pt

d) **other key-value pairs**: are forwarded to the plotMulti function of the graphics-toolbox.

returned are:

a) **axesH**: the handles to the axes

b) **fM**: the figureManager used on the figure to handle large data sets (figureManager-toolbox)

c) **ph**: the handles to the plotted segments (handles to the patches objects objects).

26. **ph = plotSegmentsOnAxes(axesIn, segmentTag, varargin)**
Will be deprecated in future. This function plots the assigned segments on the given *axesIn* as semi-transparent patches. The input arguments are:

a) **axesIn**: the axes on which the segments should be plotted

b) **segmentTag**: the name of the segment which should be plotted

c) **other key-value pairs**: are forwarded to segmentsObject.plotOnAxes.

returned are:

a) **ph**: the handles to the plotted segments (handles to the patches objects objects).

Methods (Static)

1. **bValid = isValidAliasTableTags(tIn, tags)**
   This function checks, if a table *tIn* is a valid aliasTable with respect to the given *tags*. It checks if the table has exactly one column *OrigTag* and if the entries in this column can be found in *tags*.

2. **isInList = isTagWithinTagList(tags, taglist)**
   This function checks, if the *tags* (cell array) are within the cell-array *taglist*. It returns a boolean list *isInList* with the same size as *tags* where the value 1 indicates if the tags are available in the given *taglist*.

3. **bValid = isTagWithinAliasTable(tags, aliasTab)**
   This function checks, if the *tags* (cell array) are within the alias table *aliasTab*. It returns a boolean list with the same size as *tags* where the value 1 indicates if the tags are available in the given *aliasTab*.

4. **[pa, tHandleAll] = markRangeOnAxes_givenIndexStatic(xVals, axes_in, startInds, stopInds, colorSpec, varargin)**
   This function marks a given range as a semitransparent patch. *xVals* is a vector of the xValues used to plot the patches. axes_in are the axes, on which the patches should be places. *startInds* and *stopInds* are arrays of the same size containing the indexes to *xVals* where the patches should start and end. *colorSpec* are the color specification for the patch. Optional the key-value pair *textToShow* and a string or cell array of strings of the text which should be shown within the patches. Additional all other key-value pairs are passed on to the fill command of Matlab. Returned are the handles to the patches and the handles to the text objects.

2.2 Symbolic Representation

2.2.1 SymbRepObject

Properties

1. **symbols:** Complete data of the according channel as sequence of symbols in a n x 1 categorical array. Repetitive symbols are merged to one symbol and the number of repretitions is stored in 'durations' (e.g. three repetitions {'a', 'a', 'a'} of the symbol 'a' will be merged to one symbol 'a' with a duration of three).

2. **durations:** Number of repetitions of every symbol as n x 1 array. $durations(i)$ represents the number of repetitions of the symbol $symbols(i)$. The sum of this array correlates with the number of time stamps in the channel data.

Constructor

obj = SymbRepObject(durations, symbols)

symbols must be given as categorical, *durations* must be given as numeric vector of the same dimension as *symbols*.

Methods

1. **symbRepVec = symbRepVec**
   Returns the symbolic representation as vector of the same size as the time stamps. All repetitive symbols are included in this vector, e.g.

```
>> symbols = categorical({'a'; 'b'; 'c'});
>> durations = [2; 4; 3];
>> symbolicObject = SymbRepObject(durations, symbols);
>> fullVector = symbolicObject.symbRepVec
fullVector =

  9x1 categorical array

     a
     a
     b
     b
     b
     b
     c
     c
     c
```

2. **[startInds, durations] = findSymbol(symbol)**
   Returns all indices of the first element of repetitive *symbol* in *symbRepVec* together with the according durations, e.g.

```
>> symbols = categorical({'a'; 'b'; 'a'; 'b'; 'a'});
>> durations = [2; 4; 3; 5; 2];
>> symbolicObject = SymbRepObject(durations, symbols);
>> [startInds, durations] = symbolicObject.findSymbol('a')

startInds =

   1
   7
   15

durations =

   2
   3
   2
```

3. **symbInd = findSymbolVec(symbol)**
   Returns a boolean array of the size of *symbRepVec* with "true" elements on all positions of *symbol*, e.g.

```
>> symbols = categorical({'a'; 'b'; 'c'});
```

```
>> durations = [2; 3; 1];
>> symbolicObject = SymbRepObject(durations, symbols);
>> symbInd = symbolicObject.findSymbolVec('a')

symbInd =

  6x1 logical array

   1
   1
   0
   0
   0
   0
```

4. **obj = mergeSequence(symbSequence)**
   Searches for all incidences of *symbSequence* sequences of symbols in *symbols*. Replaces the symbols of such a sequence by one new symbol, which name is assembled of the symbols of this sequence. Original symbols in such a sequence are identified by enclosing '{}', new symbols which represent a merged sequence are identified by enclosing '[]'. Merged symbols can be merged again, thus the function is ready for nested use. *symbSequence* must be given as 1×2 cell array of strings, representing one symbol of the sequence in every element of the cell array. The durations of the replaced symbols are summed over the complete sequence and set as duration of the new symbol. Example:

```
>> symbols = categorical({'a'; 'b'; 'c'; 'a'; 'b'; 'a'; 'b'});
>> durations = [2; 3; 1; 2; 1; 3; 2];
>> symbolicObject = SymbRepObject(durations, symbols);
>> symbolicObject = symbolicObject.mergeSequence({'a'; 'b'});
>> symbolicObject.symbols

ans =

  3x1 categorical array

     [{a}{b}]
     c
     [{a}{b}]

>> symbolicObject.durations

ans =

     5
     1
     8
```

5. **obj = renameSymbol(oldSymbol, newSymbol)**
   Renames all *oldSymbol* symbols to *newSymbol*.

6. **obj = mergeSymbols(oldSymbols, newSymbol)**
   Merges all *oldSymbols* symbols (given as cell array of strings) to one new *newSymbol*.

7. **obj = setSymbolsInRange(newSymbol, range)**
   Sets all symbols in *symbRepVec* in the range *range* to *newSymbol*, which is given as character array. Range must be given as array of the form [startIndex, endIndex], where *startIndex* represent the index of the first element of the range and *endIndex* represents the last element of the range, e.g.

```
>> symbols = categorical({'a'; 'b'; 'c'});
>> durations = [1; 3; 4];
>> symbolicObject = SymbRepObject(durations, symbols);
>> symbolicObject = symbolicObject.setSymbolsInRange('x', [4, 6]);
>> symbolicObject.symbols

ans =

  4x1 categorical array

     a
     b
     x
     c

>> symbolicObject.durations

ans =

     1
     2
     3
     2
```

8. **obj = removeShortSymbols(varargin)**
   Removes short symbols (symbols with a small duration). The duration of the short symbol is distributed to the enclosing symbols. Multiple successive short symbols will be treated as one short symbol and the sum of the durations will be distributed to the enclosing symbols. The following arguments are available:
   a) **shortSymbolLength:** Definition of a short symbol. All symbols with a length smaller or equal to *shortSymbolLength* will be removed or splitted, respecitvely. Default is 1.
   b) **maxNumberShortSymbols:** If the number of a sequence of multiple successive short symbols exceeds *maxNumberShortSymbols*, the sequence of these symbols will not be split, but merged to one larger symbol without any label ('<undefined>' in Matlab). Default is 5.
   c) **maxShortSymbolSequenceLength:** If the sum of all durations of multiple successive short symbols exceeds *maxShortSymbolSequenceLength*, the sequence of these symbols will not be splitted, but merged to one larger symbol without any label ('<undefined>' in Matlab). Default is 10.

d) **SplittingMode:** Name-Value-Pair. Defines how the duration of short symbols is distributed. Available options:

i. **'equal':** durations will be distributed 1 : 1 to the enclosing symbols

ii. **'weighted':** durations will be distributed due to the length of the two enclosing symbols

Default is 'equal'.

Note: If a sequence does not exceed the limits in *maxNumberShortSymbols* or *maxShortSymbolSequenceLength* but contains an unlabelled symbol ('<undefined>' in Matlab), this symbol will be treated as normal symbol and the complete sequence will be distributed to the enclosing symbols.

Example:

```
>> symbols = categorical({'a'; 'b'; 'c'; 'd'; 'e';...
>> 'f'; 'g'; 'h'; 'i'; 'j'});
>> durations = [4; 3; 4; 2; 2; 12; 2; 1; 2; 6];
>> symbolicObject = SymbRepObject(durations, symbols);
>> symbolicObject = symbolicObject.removeShortSymbols('shortSymbolLength',...
>> 3, 'maxNumberShortSymbols', 4, 'maxShortSymbolSequenceLength', 4,...
>> 'SplittingMode', 'weighted');
>> symbolicObject.symbols

ans =

  5x1 categorical array

     a
     c
     f
     <undefined>
     j

>> symbolicObject.durations

ans =

     6
     6
     15
     5
     6
```

9. **symbMarkov = genSymbMarkov(varargin)**
Generates a markov matrix for the transition from any symbol to any other symbol. The following arguments are available:

a) **Absolute:** If set to true, the number of transitions will not be normalized to a transition probability. Instead, the total number of transitions will be returned. Default is false.

10. **newObj = getTimeInterval(intervalIndices)**
    Extracts *durations* and *symbols* within the time interval given in *intervalIndices* as indices and returns the extracted data in a new symbRepObject *newObj*.

11. **symbMarkov3D = genTrigramMatrix**
    Only for development purposes!

12. **markovM = genLengthWeightedMatrix(varargin)**
    Only for development purposes!

13. **markovM = genWeightedMatrixChangedLength(varargin)**
    Only for development purposes!

14. **[allSymbRepObjects, imageMatrix, compressionData, evalRecord] = genHierarchicalCompounding(SymbObj, varargin)**
    Only for development purposes!

2.2.2 symbRepChannel

Function Signature:

returnObject = symbRepChannel(input, edges, alphabet)

Purpose

Generate the symbolic representation of a given channel of an mdtsObject.

Input Parameters:

Parameter	Description
input	Input for the computation as struct which holds the handle to the mdtsObject as input.object and the required tag as input.tag
edges	Array of length(alphabet) + 1, which contains the edges for the quantization
alphabet	Array containing the symbols to assign to the values

Return Parameters:

Parameter	Description
returnObject	symbRepObject with the symbolized channel

Description

Function to extract the data from the according channel of the mdtsObject specified in *input*, assign every data point of this channel a symbol given in *alphabet* according to the limits given in *edges*.

2.2.3 applyMCLA

Function Signature:

mclaSymbRepObject = applyMCLA(symbRepObjectsList)

Purpose

Merge symbolic representations of different channel to one representation.

Input Parameters:

Parameter	Description
symbRepObjectsList	List of all SymbRepObjects, which are supposed to be merged, as cell array

Return Parameters:

Parameter	Description
mclaSymbRepObject	SymbRepObject with merged symbolic representation

Description

Combines the symbols at every time stamp of an arbitrary number of channels to one symbolic representation.

2.3 Computations

2.3.1 compute1

Function Signature:

output = compute1(matrix, input)

Purpose

Execute convolution operation.

Input Parameters:

Parameter	Description
matrix	Convolution matrix, dense or sparse, or convolution vector
input	Input for the computation. This can be given as: 1. Vector: Double vector holding the data 2. Struct: Struct which holds the handle to the mdtsObject as input.object and the required tag as input.tag

Return Parameters:

Parameter	Description
output	Result as vector

Description

Apply a convolution with the matrix given in *matrix* on the specified input.

2.3.2 compute1Scalar

Function Signature:

output = compute1Scalar(operator, scalar, input)

Purpose

Execute operation on input.

Input Parameters:

Parameter	Description
operator	Computation operator. Possible options are: 1. '*': Element wise multiplikation 2. '/': Element wise division 3. '+': Element wise addition 4. '-': Element wise subtraction 5. handle: handle to other function
scalar	Scalar for the operation
input	Input for the computation. This can be given as: 1. Vector: Double vector holding the data 2. Struct: Struct which holds the handle to the mdtsObject as input.object and the required tag as input.tag

Return Parameters:

Parameter	Description
output	Result as vector

Description

Apply the operation specified in operator on the input, using the scalar given in scalar.

2.3.3 compute2

Function Signature:

outputVector = compute2(operator, input1, input2)

Purpose

Execute operation with two inputs.

Input Parameters:

Parameter	Description
operator	Computation operator. Possible options are: 1. '*': Element wise multiplikation 2. '/': Element wise division 3. '+': Element wise addition 4. '-': Element wise subtraction 5. 'dot': Inner product (dot product) of both vectors 6. 'outer': Outer product of both vectors 7. 'xcorr': Cross correlation between both inputs 8. handle: handle to other function
input1, input2	Inputs for the computation. They can be given as: 1. Vectors: Double vector holding the data 2. Structs: Struct which holds the handle to the mdtsObject as input.object and the required tag as input.tag

Return Parameters:

Parameter	Description
outputVector	Result as vector

Description

Apply the operation determined by *operator* on the data of both inputs. Return the result vector in the output parameter.

2.3.4 isValidInput

Function Signature:

$$\text{inputType} = \text{isValidInput}(\text{input})$$

Purpose

Validate input parameter

Input Parameters:

Parameter	Description
input	input parameter, which has to be validated

Return Parameters:

Parameter	Description
inputType	type of the input. Possible input types are: 1. 1: structured input. The input is a struct where input.object holds a reference to the mdtsObject and input.tag is a valid tag of this mdtsObject 2. 2: n x 1 vector of numeric elements 3. 0: input is invalid

Description

Checks if the given input is either a struct with a handle to the mdtsObject and a valid tag or a numeric vector. The output refers to one of these two types or to an invalid input.

2.3.5 LDOasConv

Function Signature:

derVec = LDOasConv(input, varargin)

Purpose

Compute local derivative on input

Input Parameters:

Parameter	Description
input	Input for the computation. This can be given as: 1. Vector: Double vector holding the data 2. Struct: Struct which holds the handle to the mdtsObject as input.object and the required tag as input.tag
varargin	The following parameters are available: 1. **ls:** Support length 2. **noBfs:** Number of basis functions 3. **order:** Order of the derivative. If set to 0, data of the given input will be smoothed without any differentiation

Return Parameters:

Parameter	Description
derVec	Result of the local derivative (or smoothing, respectively) of the given input as vector

Description

Generates a derivative matrix, using the *dop* function from the *DOP-Box*. The parameter m, n or $ls, noBfs$, respectively, can be passed as optional arguments. If they omitted, hard coded values in the *LDOasConv* function will be used. The generated matrix is passed to the *compute1* function (section 2.3.1), together with the input.

ATTENTION: If a vector, instead of a struct, is used as input, the step size of the time stamps will be assumed to be 1!

2.3.6 SmoothAsConv

Function Signature:

smoothedData = smoothAsConv(input, varargin)

Purpose

Smooth the data in input

Input Parameters:

Parameter	Description
input	Input for the computation. This can be given as: 1. Vector: Double vector holding the data 2. Struct: Struct which holds the handle to the mdtsObject as input.object and the required tag as input.tag
varargin	The following parameters are available: 1. **ls:** Support length 2. **noBfs:** Number of basis functions

Return Parameters:

Parameter	Description
smoothedData	Result of the smoothing of the given input as vector

Description

Smooths the data given in *input* by usage of the *LDOasConv* function with 'order' 0.

2.4 Rule Handling

2.4.1 computeFind

Function Signature:

conformElements = computeFind(operator, input, value)

Purpose

Find all time stamps in a channel which meet a certain condition

Input Parameters:

Parameter	Description
operator	Condition operator as string. Available options: $>, <, ==, \sim=, >=, <=$
input	Input for the computation as struct which holds the handle to the mdtsObject as input.object and the required tag as input.tag
value	Reference value for the condition

Return Parameters:

Parameter	Description
conformElements	Logical array which indicates the elements of *input* which fulfil the condition

Description

Finds all elements of *input* which fulfil the condition according to the operator and the value.

2.4.2 computeRule

Function Signature:

ruleElements = computeRule(operator, conformElements1, conformElements2)

Purpose

Combine or superpose two logical vectors according to a specified operator

Input Parameters:

Parameter	Description
operator	Rule operator as string. Available options: &, |
conformElements1, conformElements2	logical arrays which represent a condition applied to a data set

Return Parameters:

Parameter	Description
ruleElements	Logical array which indicates the rule conform elements

Description

Finds all elements of which apply to the rule according to the operator

2.5 Segmentation

The structure of the *segmentsObject* is illustrated in an example in figure 2.2.

segmentsObject

	Segments Tags	
	Machine Running	Machine Error
Time Stamp 1	True	False
Time Stamp 2	True	False
Time Stamp 3	True	False
Time Stamp 4	True	False
Time Stamp 5	False	False
Time Stamp 6	False	False
Time Stamp 7	False	False
Time Stamp 8	True	False
Time Stamp 9	True	False
Time Stamp 10	False	False
Time Stamp 11	False	False
Time Stamp 12	False	True
Time Stamp 13	False	True
Time Stamp 14	False	True

Figure 2.2: Structure of the segmentsObject

This example shows two segments in the segmentsObject: One with the tag label "Machine Running" and one with "Machine Error". The segment "Machine Running" shows two instances of this segment: From time stamp 1 to time stamp 4 and from time stamp 8 to time stamp 9. "Machine Error" shows only one instance from time stamp 12 to time stamp 14. Note that the label in this object refers to the name of the segments - they do not correlate with the tags of the mdtsObject in any way! To each channel of the mdtsObject one segmentsObject can be assigned.

2.5.1 segmentsObject

Properties

1. Core Data
   a) **nTimeStamps:** Total number of time stamps in the segments. Must correlate with the number of time stamps in the data of the according mdtsObject.
   b) **tags:** Names of the segments as row vector.
   c) **starts:** Will be renamed in future. Start indices of all instances of the according segment. Stored as cell array (row vector), where every cell contains a column vector representing all start indices of the according segment.

d) **durations:** Number of time stamps of every instance of the according segment. Stored as cell array (row vector), where every cell contains a column vector representing all durations of the according segment.

2. Dependend Properties

a) **startInds:** Start indices of all instances of the according segment. Stored as cell array (row vector), where every cell contains a column vector representing all start indices of the according segment.

b) **stopInds:** Stop indices of all instances of the according segment. Stored as cell array (row vector), where every cell contains a column vector representing all stop indices of the according segment.

Constructor

$$\text{obj} = \text{segmentsObject}(\text{nTimestamps})$$

nTimestamps must be given as numerical 1×1 array.

Methods

1. **obj = addSegmentVector(tagName, bVec)**

   Adds new segments given in the logical vector *bVec* with label *tagName* to the segmentsObject.

2. **obj = addSegmentVectorStartDuration(tagName, startInds, durations)**

   Adds new segments definded by the vectors *startInds* and *durations* with label *tagName* to the segmentsObject.

3. **bVec = getLogicalVector(tagName)**

   Returns the segment of the name *tagName* as logical vector (*starts* and *durations* expanded to a complete logical vector).

4. **lVec = getLabeledVector(tagName)**

   Returns the segment of the name *tagName* as labelled vector (*starts* and *durations* expanded to a complete logical vector). All n instances of the according segment are labelled as $1, 2, ...n$. In case of the example in figure 2.2, this would be

```
>> mySegmentsObject.getLabeledVector('MachineRunnning')

ans =

1
1
1
1
0
0
0
2
```

```
2
0
0
0
0
0
```

5. **newObj = extractRows(intervalIndices)**

   Extracts the time stamps given in *intervalIndices* as 1×2 vector from all segments of the according object and returns the extracted segmentsObject.

6. **[pa, tHandleAll] = plotOnAxes(axes_in, xTime, varargin)**

7. **ph = plotSegmentsOnAxes(axesIn, segmentTag, varargin)**
   Will be deprecated in future. This function plots the assigned segments on the given *axesIn* as semi-transparent patches. The input arguments are:
   a) **axes_in**: the axes on which the segments should be plotted
   b) **xTime**: the abscissae values used for plotting the patches and the indexes are referring to
   c) **segmentTags**:optional key-value pair; the names (nameIDs) of the segments to be plotted on all of the axes
   d) **colorDismiss**:optional key-value pair; color to be not used as patch color
   e) **plotSegName**:optional key-value pair; boolean flag indicating if the segment name (segTag) should be annotated as text within the patch
   f) **plotSegDuration**:optional key-value pair; boolean flag indicating if the segment duration should be annotated as text within the patch
   g) **plotSegDuration**:optional key-value pair; minimum number of datapoints (duration) within a segment to annotate the segment name and/or duration
   h) **additional key-value pairs**: these are forwarded to the Matlab

   returned are:
   a) **pa**: the handles to the plotted segments (handles to the patches objects objects).
   b) **tHandleAll**: the handles to annotated text objects.

2.6 Supporting Functions

2.6.1 plotmdtsObject

Function Signature:

[out, fM, ph] = plotmdtsObject(inputObject, varargin)

Purpose

Plot the complete mdtsObject given in *inputObject*.

Input Parameters:

Parameter	Description
inputObject	mdtsObject (with multiple channels) to be plotted
varargin	The following parameters are available: 1. **bUseDatetime:** Plot the time stamps as DateTime when set to true, plot as datenum when set to false. Default is true. 2. **bUseTimeRelative:** Plot the time stamps as relative time where the first time stamp starts at zero. Default is false. 3. **plotSymbolName:** Plot the name of a symbol within the symbol area 4. **plotSymbolDuration:** Plot the duration of a symbol within the symbol area 5. **plotSymbolNameMinLengthRelative:** Minimum number of time stamps which is necessary, to plot the symbol name or symbol duration within the symbol area. Must be given as ratio of the total number of time stamps from 0 to 1. 6. Additional parameters, which will be passed to the 'plotMulti' function

Return Parameters:

Parameter	Description
out	$1 \times n$ Axes array, which holds the handles to all axes of all n channels of the mdtsObject
fM	Handle to the figure manager
ph	$1 \times n$ Line array, which holds the handles to all plotted lines of all n channels of the mdtsObject

Description

Use the plotMulti function to plot the channels of the given object. Add events and symbolic representations, if available.

Note: If one wants to plot parts of the object only, one has to extract the required data from the object first and generate a new object with this extracted data. This can be achieved using the *getData* function or direct indexing of the mdtsObject.

2.6.2 plotSegments

Function Signature:

[out, fM, ph] = plotSegments(inputObject, segmentTag, varargin)

Purpose

Plot the mdtsObject given in *inputObject* and highlight all instances of the segment specified in *segmentTag*.

Input Parameters:

Parameter	Description
inputObject	mdtsObject (with multiple channels) to be plotted
segmentTag	tag of the required segment as string (character array)
varargin	The following parameters are available: 1. **Size:** Height and width of the plotted figure as 1×2 vector 2. **FontSize:** Font size in the plotted figure

Return Parameters:

Parameter	Description
out	$1 \times n$ Axes array, which holds the handles to all axes of all n channels of the mdtsObject
fM	Handle to the figure manager
ph	$1 \times n$ Line array, which holds the handles to all plotted lines of all n channels of the mdtsObject

Description

Use the plotMulti function to plot the channels of the given object. Highlight all instances of the given segment as red, transparent box in all axes of the plot.