Multivariate Splines Reference Manual

Generated by Doxygen 1.3.6

Fri Mar 18 16:43:00 2005

Contents

Chapter 1

Multivariate Spline Library

1.1 What is this

This library is designed for the manipulation of multi-indexed data in general and multivariate tensor-product splines in particular. Examples of multi-indexed data are usual vectors (one index) and matrices (two indices), but arrays with arbitrary number of indices can be constructed with equal ease. Several different **grid classes** are provided that are specially adopted for different needs. Such multi- indexed grids can be copied, read and written, reshaped, permuted on their indices, added together, multiplied and several other operations. The **multivariate spline class** represents a scalar spline (values in R) that is parameterized on an arbitrary number of parameters. This can be useful to describe smooth scalar fields that can be evaluated anywhere inside the parametric domain. Many operations can be used on the spline, including pointwise evaluation, evaluation inside a predefined area, knot insertion and order raising, reduction to a spline with fewer parameters, fitting to discrete, multi-indexed data, reading, writing, copying and many other operations. For the purpose of fitting such a spline to multi-indexed, discrete data, many methods can be used.

1.2 What is included

The main thee categories of objects are **grids**, **splines** and **approximators**.

1.2.1 Grids

There are several flavors of multi-indexed grids. Often, the user might have a bulk of data somewhere in memory that he or she wants to represent on a grid form, allowing for indexing to a specific element, permuting, streaming, etc. In that case, it is not necessary for the grid to own the data it refers to. In fact, ownership of such data would often be *highly un-recommendable* , since the data arrays can be very big and we want to avoid unnecessary copying and shuffling around of the data. (Often when working with multi-indexed data, the sheer size of the data can make it very impractical to work with unless it is stored centrally in one place and then referred to from various other objects). For this purpose we have created the template class Go::GoBorrowed-MVGrid<int, typename>. The first template parameters defines how many indices the grid has (one - "vector", two - "matrix"...), and the second tells which kind of elements are contained in the grid.

A disadvantage working with grids that do not own their data is that we always have to keep track of *who does* . Moreover, whenever creating or resizing an existing grid, we have to keep in mind how much memory is actually available at the location the grid refers to. It is, for instance, fully possible to resize a grid such that it can refer to elements that haven't been allocated in memory in the first place. For small and medium-sized grids, it can often be useful to let the grids own their actual data. Such a grid would automatically allocate enough memory when created or resized, and would delete it when it goes out of scope. It would be much slower to copy, but in some cases that can be perfectly tolerated. In this library, there are two grid that serve this purpose. The first is a template class called Go::GoSelfcontainedGrid<int, typename>, which is very similar to the previously described Go::GoBorrowedMVGrid<int, typename>, except that it has ownership to its data. (In fact, these two grids both derive from a base class template called Go::GoGeneric-Grid<int, typename>). The second self-owning grid is called Go::GoScalableGrid<typename>. This grid is only a template on the type of elements it contains, while the number of indices can be set (and changed) runtime.

1.2.2 Spline

There is only one multivariate spline class in this library, Go::GoTensorProductSpline<int, typename>. It is a template on number of parameters and type of element the coefficients should be. The actual element used must support basic arithmetic operations and also be multipliable with a `double`. Note that the spline's knot-vectors (one for each parameter) will always be defined using `double`. The spline stores its control points in a grid of type **Go::GoBorrowed-MVGrid**(p. 14)<int, typename> and thus does not own the memory area used for storing them. With other words, it is the user's responsibility to allocate the required memory before constructing the spline, as well as keep track of when it should be deleted.

Note:

One immediate use of this design is that the user can always express an already-existing multi-indexed dataset somewhere in memory as a *linear spline* , thus making it possible to continuously interpolate the values in the dataset.

Contrary to the well-known spline curves and surfaces, who usually take values in R^2 or R^3, the **Go::GoTensorProductSpline**(p. 51) takes its values in R only. This means that if a user wants to express a curve or surface, she will have to use several objects of type **Go::GoTensorProductSpline**(p. 51); one for each spatial dimension. A utility function, Go::createSplineSurface(...) is provided to convert such a group of **Go::GoTensorProduct-Spline**(p. 51) to the perhaps more familiar Go-object Go::SplineSurface.

1.2.3 Approximators

Approximators is a kind of objects that are used in conjunction with **Go::GoTensorProduct-Spline**(p. 51) to make the latter approximate a given dataset. Each approximator class defines an approximation method to use for this data fitting process. When the user wants to create a spline approximation of a given, multi-indexed dataset, he first has to allocate enough memory and create a **Go::GoTensorProductSpline**(p. 51) with one parameter for each index in the dataset (ex. a 4-indexed dataset required a 4-variate spline). He then should call the **Go::GoTensorProduct-Spline**(p. 51) member function **Go::GoTensorProductSpline::fit()**(p. 59) with an array of the approximators that describe the kind of approximation he wants. There must be one approximator defined for each parameter in the spline (so in our example, we would need four approximators). All approximator derive from the abstract base class **Go::GoApproximator**(p. 11). Currently defined approximators are:

- **Go::GoLeastSquareApproximator**(p. 35) (penalized least squares approximation)
- **Go::GoSchoenbergApproximator**(p. 45) (variation diminishing approximation)

- **Go::GoKnotremovalApproximator**(p. 32) (approximation using the knot-removal algorithm)
- **Go::GoHybridApproximator**(p. 29) (constrained least squares that tries to limit overshoots)
- **Go::GoNeutralApproximator**(p. 38) (no approximation along this parameter direction)

1.3 Sample use

The following **Practical example**(p. 65) page describes an imagined session where a user benefits from this library's classes and functions to manage the output from a big atmospheric simulation.

Chapter 2

Multivariate Splines Hierarchical Index

2.1 Multivariate Splines Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Multivariate Splines Class Index

3.1 Multivariate Splines Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

Multivariate Splines Page Index

4.1 Multivariate Splines Related Pages

Here is a list of all related documentation pages:

Chapter 5

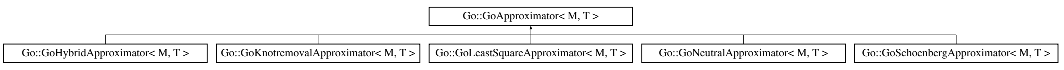
Multivariate Splines Class Documentation

5.1 Go::GoApproximator< M, T > Class Template Reference

The abstract base class for approximator objects, used by the **GoTensorProductSpline**(p. 51) class to specify a method for fitting a spline to a given dataset.

`#include <GoApproximator.h>`

Inheritance diagram for Go::GoApproximator< M, T >::


Go::GoApproximator< M, T >
Go::GoHybridApproximator< M, T >
Go::GoKnotremovalApproximator< M, T >
Go::GoLeastSquareApproximator< M, T >
Go::GoNeutralApproximator< M, T >
Go::GoSchoenbergApproximator< M, T >

Public Member Functions

- virtual void **approximate** (**GoBorrowedMVGrid**< M, T > ∗orig_data_array, **GoBorrowedMVGrid**< M, T > ∗cyclically_permuted_result_array, BsplineBasis &basis)=0

  A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

- virtual int **approximationSize** () const=0

  If the approximation algorithm is of a kind that makes it possible to predict the number of resulting spline control points (without actually running the algorithm first), then the following function should return this number.

5.1.1 Detailed Description

template<int M, typename T> class Go::GoApproximator< M, T >

The abstract base class for approximator objects, used by the **GoTensorProductSpline**(p. 51) class to specify a method for fitting a spline to a given dataset.

When the user wants to fit a multivariate spline (of the **GoTensorProductSpline**(p. 51) class) to a given dataset, she has to call the **GoTensorProductSpline::fit()**(p. 59) function. In this function, she has to specify which methods to use. Each of the M parameters in the spline can be assigned a method independently of the other parameters. For instance, if we want to approximate a spline to a computed atmospheric 3D grid (here considered as the "sampled" data), the parameters of the spline would typically be latitude, longitude and height. Each of these parameters could be fitted to the dataset independently, for instance using least squares (**GoLeastSquareApproximator**(p. 35)) to fit the longitude and latitude, and knotremoval (**GoKnotremovalApproximator**(p. 32)) to fit the height. Each class derived from **GoApproximator**(p. 11) represents a way to fit a spline to a dataset, and the parameters needed to specify the corresponding algorithm are typically specified in the constructor of the approximator class. The user can also encapsulate her own fitting algorithms in new objects that derives from **GoApproximator**(p. 11) and thus have it applied on the spline, without having to rewrite the **GoTensorProductSpline::fit()**(p. 59) function.

Definition at line 43 of file GoApproximator.h.

5.1.2 Member Function Documentation

5.1.2.1 template<int M, typename T> virtual void Go::GoApproximator< M, T >::approximate (GoBorrowedMVGrid< M, T > * *orig_data_array*, GoBorrowedMVGrid< M, T > * *cyclically_permuted_result_array*, BsplineBasis & *basis*) `[pure virtual]`

A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

All necessary parameters or modifiers to the encapsulated approximation algorithm should already have been set by the class' constructor or other methods. The sampled data is given in the form of an M-indexed grid. The approximation takes place along the index with the longest stride (the last index). In other words, if we have M indexes (0...M-1) and index m counts L_m elements, then we would consider the dataset a collection of L_{M-1} datapoints, where each datapoint has $D = Pi_{m=0}^{m-2} L_m$ components. The result is then a univariate spline with control points in R^D and a basis B. The resulting spline's coefficients is written to the grid pointed to by the second argument. It will be cyclically permuted by one step (see **GoGenericGrid::cyclicPermute()**(p. 20)) compared to the dataset, which allows us to ***use it as input*** for the **approximate()**(p. 12) function of the **GoApproximator**(p. 11) object that is to be applied on the next parameter. The generated basis will be returned in the last argument. The philosophy is that, for an M-variate dataset, applying the **approximate()**(p. 12) function M times (usually by M different **GoApproximator**(p. 11) objects), the first time on the sample data itself, later on the result from the last call of the **approximate()**(p. 12) function, then the resulting coefficient grid will be the control points for an M-variate spline approximating the original dataset. (We have to take care of the basis functions at each call too, in order to have a complete definition of the spline).

Note:

In practice, all this is taken care of by the **GoTensorProductSpline::fit()**(p. 59) function, so the only thing the user should usually be concerned about is the construction and definition of the desired approximator objects.

Parameters:

orig_data_array pointer to the gridded data that we want to 'fit' the spline to. The fitting will take place along the index with the longest stride, which is the last one.

cyclically_permuted_result_array Pointer to resulting spline's coefficient grid (to be filled out by the algorithm). It will be cyclically permuted (see **GoGenericGrid::cyclic-Permute()**(p. 20)) by one step.

basis the spline basis resulting from this approximation.

Implemented in **Go::GoHybridApproximator< M, T >** (p. 30), **Go::GoKnotremoval-Approximator< M, T >** (p. 33), **Go::GoLeastSquareApproximator< M, T >** (p. 36), **Go::GoNeutralApproximator< M, T >** (p. 38), and **Go::GoSchoenbergApproximator< M, T >** (p. 46).

5.1.2.2 template<int M, typename T> virtual int Go::GoApproximator< M, T >::approximationSize () const `[pure virtual]`

If the approximation algorithm is of a kind that makes it possible to predict the number of resulting spline control points (without actually running the algorithm first), then the following function should return this number.

If this is imossible to predict before execution of the approximation algorithm, (ie. the size of the approximation depends on the dataset to be approximated), a *negative* value will be returned. The absolute value of this negative value, multiplied with the number of datapoints to approximate (length of index with longest stride in the sample grid), will give an upper estimate about how much memory is needed to store the spline coefficients.

Returns:

The number of control points if it can be predicted, else a negative value

Implemented in **Go::GoHybridApproximator< M, T >** (p. 31), **Go::GoKnotremoval-Approximator< M, T >** (p. 33), **Go::GoLeastSquareApproximator< M, T >** (p. 37), **Go::GoNeutralApproximator< M, T >** (p. 39), and **Go::GoSchoenbergApproximator< M, T >** (p. 46).

The documentation for this class was generated from the following file:

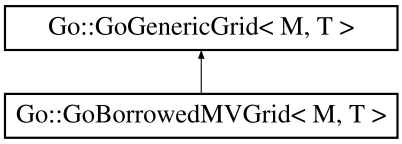
- GoApproximator.h

5.2 Go::GoBorrowedMVGrid< M, T > Class Template Reference

This is a multiindexed grid that derives from **GoGenericGrid**(p. 17). It does not own the memory array where the elements are stored.

`#include <GoBorrowedMVGrid.h>`

Inheritance diagram for Go::GoBorrowedMVGrid< M, T >::



Public Member Functions

- **GoBorrowedMVGrid** ()

  Constructor making an empty, multiindexed grid with M indices, where each.

- **GoBorrowedMVGrid** (T *data)

  Constructor making an empty grid with M indexes. The data storage area is specified.

- **GoBorrowedMVGrid** (T *data, const int *const new_size)

  Constructor making a M-indexed grid with a given shape.

- **GoBorrowedMVGrid**< M-1, T > **subgrid** (int i)

  Returns a M-1 multiindexed grid which is a subgrid of the 'this' grid.

- void **swap** (**GoBorrowedMVGrid**< M, T > &rhs)

  Rapidly swap two grids.

- void **setDataPointer** (T *address)

  Define which position in memory the grid should now use for storage/retrieval of its elements.

5.2.1 Detailed Description

template<int M, typename T> class Go::GoBorrowedMVGrid< M, T >

This is a multiindexed grid that derives from **GoGenericGrid**(p. 17). It does not own the memory array where the elements are stored.

In other words, it is not an "information carrier", but is only considered to logically simplify access to a data range that is already present in the computer memory.

Definition at line 35 of file GoBorrowedMVGrid.h.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 template<int M, typename T> Go::GoBorrowedMVGrid< M, T >::GoBorrowedMVGrid () `[inline]`

Constructor making an empty, multiindexed grid with M indices, where each.

The created grid is immediately useless, as it has size 0 and no designated storage area. It can however be assigned to other grids, have its size and data area manually set with other member functions, etc...

Definition at line 44 of file GoBorrowedMVGrid.h.

5.2.2.2 template<int M, typename T> Go::GoBorrowedMVGrid< M, T >::GoBorrowedMVGrid (T * *data*) `[inline]`

Constructor making an empty grid with M indexes. The data storage area is specified.

The created grid is immediately useless, since it has size 0. It can however be assigned to other grids, be copied into, have its size and data area manually set with other member functions, etc...

Parameters:

data pointer to a memory area where elements should be stored

Definition at line 53 of file GoBorrowedMVGrid.h.

5.2.2.3 template<int M, typename T> Go::GoBorrowedMVGrid< M, T >::GoBorrowedMVGrid (T * *data*, const int *const *new_size*) `[inline]`

Constructor making a M-indexed grid with a given shape.

Parameters:

new_size points to a M-sized array of integers, specifying the shape of the grid (length of each index).

data pointer to the start of the memory area where the user wants the grid to store/locate its elements. It is the user's responsibility that this memory area is allocated and that its size is sufficient.

Definition at line 63 of file GoBorrowedMVGrid.h.

5.2.3 Member Function Documentation

5.2.3.1 template<int M, typename T> void Go::GoBorrowedMVGrid< M, T >::setDataPointer (T * *address*) `[inline]`

Define which position in memory the grid should now use for storage/retrieval of its elements.

It is the user's responsibility that the memory area pointed to is sufficiently large to contain all elements in the grid.

Definition at line 110 of file GoBorrowedMVGrid.h.

5.2.3.2 template<int M, typename T> GoBorrowedMVGrid<M-1, T> Go::GoBorrowedMVGrid< M, T >::subgrid (int *i*) `[inline]`

Returns a M-1 multiindexed grid which is a subgrid of the 'this' grid.

A grid with M indexes (with index lengths L_1, L_2, ... L_M), can be seen as a set of grids with (M-1) indexes (where the index lengths are L_1, L_2, ... L_{M-1}. The number of such subgrids is L_M. The reason we can "decompose" the grid this way is because its elements are stored fortran-style so that the last index has the longest stride. This function returns one of these (M-1)-indexed grids. The returned grid will share its memory area with the 'this' grid (actually it is a subset of it). The returned subgrid should therefore be seen as an independent interpretation of a PART OF the original grid.

Parameters:

i this integer specifies which of the above defined subgrids is to be returned. *i* could be from 0 to L_M-1.

Returns:

the subgrid as defined above

Definition at line 80 of file GoBorrowedMVGrid.h.

References Go::GoGenericGrid< M, T >::rowlength().

5.2.3.3 template<int M, typename T> void Go::GoBorrowedMVGrid< M, T >::swap (GoBorrowedMVGrid< M, T > & *rhs*) `[inline]`

Rapidly swap two grids.

Parameters:

rhs the grid to swap with

Definition at line 100 of file GoBorrowedMVGrid.h.

The documentation for this class was generated from the following file:

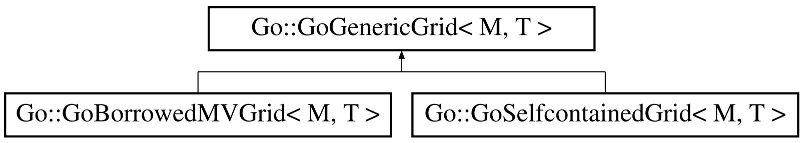
- GoBorrowedMVGrid.h

5.3 Go::GoGenericGrid< M, T > Class Template Reference

This is a basic, multiindexed grid that carries objects of type T. The number of indexes is M.

`#include <GoGenericGrid.h>`

Inheritance diagram for Go::GoGenericGrid< M, T >::


Go::GoGenericGrid< M, T >
Go::GoBorrowedMVGrid< M, T >
Go::GoSelfcontainedGrid< M, T >

Public Member Functions

- void **datacopy** (T ∗new_location) const

  Bulk copy of all grid values into a given memory area.

- void **fillValue** (T value)

  This function sets all values in the grid equal to the argument.

- void **fillValue** (T val, int ∗lower, int ∗upper)

  Sets all values in a subset of the grid to a particular value.

- void **clone** (**Go::GoGenericGrid**< M, T > &newgrid) const

  Make the argument grid a copy of this grid.

- void **read_ASCII** (std::istream &is)

  Reads a grid from a stream, using the ASCII format.

- void **write_ASCII** (std::ostream &os) const

  Writes the grid to a stream, using the ASCII format.

- void **read_BINARY** (std::istream &is)

  Reads the grid from a stream, using BINARY format.

- void **write_BINARY** (std::ostream &os) const

  Writes the grid to a stream, using the BINARY format.

- void **dumpCoefs** (std::ostream &os) const

  Write the contents of a grid to a stream, but not its shape. ASCII format is used.

- void **dumpCoefs_binary** (std::ostream &os) const

  Write the contents of a grid to a stream, but not its shape. BINARY format is used.

- void **blockRead** (const int ∗start_ix_this, const int ∗start_ix_other, const int ∗len_read, const **GoGenericGrid**< M, T > &other)

  Reads a block of elements from another grid and writes it into the 'this' grid.

- int **size** () const

  Returns the total number of elements in the grid.

- T **maxElem** () const

  Returns the maximum element in the grid.

- T **minElem** () const

  Returns the minimum element in the grid.

- virtual int **resize** (const int ∗const new_size)

  Redefining the shape of the grid (number of elements in each index).

- const int ∗const **rowlength** () const

  Returns a pointer to the memory area specifying the shape of the grid.

- int **rowlength** (int i) const

  Returns the number of elements along index i.

- T & **operator[ ]** (const int ∗const coords)

  Returns a reference to a specified element in the array.

- const T & **operator[ ]** (const int ∗const coords) const

  Returns a const reference to a specified element in the array.

- T ∗ **getDataPointer** ()

  Return a pointer to the start of the memory range where elements are stored.

- const T ∗const **getDataPointer** () const

  Return a const pointer to the start of the memory range where elements are stored.

- void **operator+=** (const **GoGenericGrid**< M, T > &rhs)

  Adds the elements of another grid to 'this' grid, element by element.

- void **operator-=** (const **GoGenericGrid**< M, T > &rhs)

  Subtracts the elements of another grid from 'this' grid, element by element.

- void **operator ∗=** (const T &val)

  Multiply each of the elements in the grid by a value.

- int **findPosition** (const int ∗const pos) const

  Finds the memory position of an indexed element, given as an offset from the start of the internal storage range, as accessed by **getDataPointer()**(p. 23).

- void **flipDirection** (int dir)

  Reverses the ordering of elements in the grid for a given index.

- void **permuteElements** (const int ∗permutation)

  Re-arranges the orders of the indexes to elements in the grid.

- void **permuteElements_memOpt** (int ∗permutation)

Does the same as **permuteElements()**(p. 25)*, but using much less memory.*

- void **cyclicPermute** (int steps)

  Cyclically permute the indexes of a grid.

5.3.1 Detailed Description

template<int M, typename T> class Go::GoGenericGrid< M, T >

This is a basic, multiindexed grid that carries objects of type T. The number of indexes is M.

The purpose of this grid class is to serve as a base class for other grid classes. Because of implementation issues, it cannot be abstract, but it should be considered to be, even though it contains no abstract functions. To enforce this, it doesn't have any public constructor. The user should always use one of the derived grid classes.

Note: The indexing of the elements in the grid is done fortran style, ie. the last index has the longest stride.

Definition at line 40 of file GoGenericGrid.h.

5.3.2 Member Function Documentation

5.3.2.1 template<int M, typename T> void Go::GoGenericGrid< M, T >::blockRead (const int * *start_ix_this*, const int * *start_ix_other*, const int * *len_read*, const GoGenericGrid< M, T > & *other*) [inline]

Reads a block of elements from another grid and writes it into the 'this' grid.

This function reads a block of elements from the 'other' grid and writes it to the current grid. start index positions for the block in 'this' grid (target) and the 'other' grid (source) are given by the first two arguments. The third argument gives the size (rowlengths) of the block to be read. If indexation causes a part of the specified block to be out of bounds for the grids involved, an exception will be thrown.

Parameters:

start_ix_this should point to an M-length array specifying the start index position in 'this' grid, to which the data should be written.

start_ix_other should point to an M-length array specifying the start index position in the 'other' grid, from where the data should be read.

len_read should point to an M-length array specifying the shape (number of elements in each index) of the block to be read/written

other the grid from which to read the elements.

Definition at line 271 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::operator[](), and Go::GoGenericGrid< M, T >::rowlength().

5.3.2.2 template<int M, typename T> void Go::GoGenericGrid< M, T >::clone (Go::GoGenericGrid< M, T > & *newgrid*) const [inline]

Make the argument grid a copy of this grid.

Parameters:

newgrid Reference to a grid (derived from this class) that will be reshaped and filled with values such that it becomes a copy of 'this' grid.

Definition at line 113 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::datacopy(), Go::GoGenericGrid< M, T >::getDataPointer(), and Go::GoGenericGrid< M, T >::resize().

Referenced by Go::GoNeutralApproximator< M, T >::approximate(), Go::GoKnotremovalApproximator< M, T >::approximate(), Go::GoSelfcontainedGrid< M, T >::GoSelfcontainedGrid(), and Go::GoGenericGrid< M, T >::permuteElements().

5.3.2.3 template<int M, typename T> void Go::GoGenericGrid< M, T >::cyclicPermute (int *steps*) `[inline]`

Cyclically permute the indexes of a grid.

A cyclical permutation with k "steps" will put the index currently at i in the new position (i - k)%M

Example: For a 3-indexed grid (i, j, k):

- cyclicPermute(0) - new index order will be (i, j, k) (unchanged)
- cyclicPermute(1) - new index order will be (j, k, i)
- cyclicPermute(2) - new index order will be (k, i, j)
- cyclicPermute(3) - new index order will be (i, j, k) (unchanged)
- etc...

Parameters:

steps number of steps to cyclic permute

Definition at line 783 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::permuteElements().

Referenced by Go::GoNeutralApproximator< M, T >::approximate(), and Go::GoKnotremovalApproximator< M, T >::approximate().

5.3.2.4 template<int M, typename T> void Go::GoGenericGrid< M, T >::datacopy (T * *new_location*) const `[inline]`

Bulk copy of all grid values into a given memory area.

Parameters:

new_location Pointer to memory area where grid values should be copied.

Definition at line 48 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::size().

Referenced by Go::GoGenericGrid< M, T >::clone().

5.3.2.5 template<int M, typename T> void Go::GoGenericGrid< M, T >::dumpCoefs (std::ostream & *os*) const `[inline]`

Write the contents of a grid to a stream, but not its shape. ASCII format is used.

The number of data units sent to the stream thus equals the total volume of the grid.

Parameters:

os the stream to write the contents to

Definition at line 226 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::size().

Referenced by Go::GoGenericGrid< M, T >::write_ASCII().

5.3.2.6 template<int M, typename T> void Go::GoGenericGrid< M, T >::dumpCoefs_binary (std::ostream & *os*) const `[inline]`

Write the contents of a grid to a stream, but not its shape. BINARY format is used.

The number of data units sent to the stream this equals the total volume of the grid.

Parameters:

os the stream to write the contents to

Definition at line 240 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::size().

Referenced by Go::GoGenericGrid< M, T >::write_BINARY().

5.3.2.7 template<int M, typename T> void Go::GoGenericGrid< M, T >::fillValue (T *val*, int * *lower*, int * *upper*) `[inline]`

Sets all values in a subset of the grid to a particular value.

This function sets all values to 'val' in the M-dimensional cube whose lower coordinate corner and upper coordinate corner is given in 'lower' and 'upper'. ('lower' and 'upper' should point to ranges of M elements, and upper[i] > lower[i] for i ranging from 0 to M-1).

Parameters:

val The value with which to fill the subset

lower Should point to a range of M elements, specifying the "lower" corner of the subset.

upper Should point to a range of M elements, specifying the "upper" corner of the subset.

Definition at line 78 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::operator[](), and Go::GoGenericGrid< M, T >::rowlength().

5.3.2.8 template<int M, typename T> void Go::GoGenericGrid< M, T >::fillValue (T *value*) `[inline]`

This function sets all values in the grid equal to the argument.

Parameters:

value The value to initialize the grid with.

Definition at line 59 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::size().

5.3.2.9 template<int M, typename T> int Go::GoGenericGrid< M, T >::findPosition (const int ∗const *pos*) const `[inline]`

Finds the memory position of an indexed element, given as an offset from the start of the internal storage range, as accessed by **getDataPointer()**(p. 23).

Finds the memory position of an indexed element, given as an offset from the start of the internal storage range, as accessed by **getDataPointer()**(p. 23).

Parameters:

pos pointer to an M-sized array of integers, specifying the multiindex of the requested element

Returns:

the position of the stored element, as an offset from the start of the internal storage range (the latter can be obtained by calling the function **getDataPointer()**(p. 23))

Definition at line 536 of file GoGenericGrid.h.

Referenced by Go::GoLeastSquareApproximator< M, T >::approximate(), Go::GoGenericGrid< M, T >::flipDirection(), Go::GoGenericGrid< M, T >::operator[](), and Go::GoGenericGrid< M, T >::permuteElements_memOpt().

5.3.2.10 template<int M, typename T> void Go::GoGenericGrid< M, T >::flipDirection (int *dir*) `[inline]`

Reverses the ordering of elements in the grid for a given index.

Reverses the ordering of elements in the grid for the index 'dir'. To 'reverse' a direction means that (supposing that N = rowlength(dir)) the element with positioned at i for the 'dir' index will now be positioned at N - i - 1.

Parameters:

dir the index that we want to reverse. Must be between 0 and M-1.

Definition at line 559 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::findPosition(), and Go::GoGenericGrid< M, T >::rowlength().

5.3.2.11 template<int M, typename T> const T∗ const Go::GoGenericGrid< M, T >::getDataPointer () const `[inline]`

Return a const pointer to the start of the memory range where elements are stored.

Returns:

const pointer to start of element storage memory range. Elements in this are stored by their indexes fortran-style, which means that the first index has the lowest stride. (Ex. in a 2-indexed array of shape (I, J), element e(i,j) is immediately followed by element e(i+1, j) as long as i+1 < I. Element e(I-1, j) is followed by e(0, j+1)).

Definition at line 464 of file GoGenericGrid.h.

5.3.2.12 template<int M, typename T> T* Go::GoGenericGrid< M, T >::getDataPointer () `[inline]`

Return a pointer to the start of the memory range where elements are stored.

Returns:

pointer to start of element storage memory range. Elements in this are stored by their indexes fortran-style, which means that the first index has the lowest stride. (Ex. in a 2-indexed array of shape (I, J), element e(i,j) is immediately followed by element e(i+1, j) as long as i+1 < I. Element e(I-1, j) is followed by e(0, j+1)).

Definition at line 451 of file GoGenericGrid.h.

Referenced by Go::GoSchoenbergApproximator< M, T >::approximate(), Go::GoLeastSquareApproximator< M, T >::approximate(), Go::GoKnotremovalApproximator< M, T >::approximate(), Go::GoHybridApproximator< M, T >::approximate(), Go::GoGenericGrid< M, T >::clone(), Go::GoGenericGrid< M, T >::operator *=(), Go::GoGenericGrid< M, T >::operator+=(), and Go::GoGenericGrid< M, T >::operator-=().

5.3.2.13 template<int M, typename T> T Go::GoGenericGrid< M, T >::maxElem () const `[inline]`

Returns the maximum element in the grid.

Returns:

the maximum element in the grid

Definition at line 328 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::size().

5.3.2.14 template<int M, typename T> T Go::GoGenericGrid< M, T >::minElem () const `[inline]`

Returns the minimum element in the grid.

Returns:

the minimum element in the grid

Definition at line 347 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::size().

5.3.2.15 template<int M, typename T> void Go::GoGenericGrid< M, T >::operator *= (const T & *val*) `[inline]`

Multiply each of the elements in the grid by a value.

Each of the elements in the grid are multiplied with the value 'val'. A compile-time error will occur if the type T does not define a multiplication operator.

Parameters:

val the value that each of the elements in the grid will be multiplied with

Definition at line 518 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::getDataPointer(), and Go::GoGenericGrid< M, T >::size().

5.3.2.16 template<int M, typename T> void Go::GoGenericGrid< M, T >::operator+= (const GoGenericGrid< M, T > & *rhs*) `[inline]`

Adds the elements of another grid to 'this' grid, element by element.

The elements of the 'rhs' grid are added to those of 'this' grid, element by element. The grids must have the same shape, or a runtime error will be thrown. A compile-time error will occur if the type T does not define the addition operation.

Parameters:

rhs a grid to "add" to 'this' grid, element by element

Definition at line 476 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::getDataPointer(), Go::GoGenericGrid< M, T >::rowlength(), and Go::GoGenericGrid< M, T >::size().

5.3.2.17 template<int M, typename T> void Go::GoGenericGrid< M, T >::operator-= (const GoGenericGrid< M, T > & *rhs*) `[inline]`

Subtracts the elements of another grid from 'this' grid, element by element.

The elements of the 'rhs' grid are added to those of 'this' grid, element by element. The grids must have the same shape, or a runtime error will be thrown. A compile-time error will occur if the type T does not define the subtraction operation.

Parameters:

rhs a grid to "subtract" from 'this' grid, element by element

Definition at line 497 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::getDataPointer(), Go::GoGenericGrid< M, T >::rowlength(), and Go::GoGenericGrid< M, T >::size().

5.3.2.18 template<int M, typename T> const T& Go::GoGenericGrid< M, T >::operator[ ] (const int *const *coords*) const `[inline]`

Returns a const reference to a specified element in the array.

The element returned is the one having the indexes given as argument.

Parameters:

coords pointer to an M-sized range of integers, representing the indexes for the requested element.

Returns:

a const reference to the element requested

Definition at line 436 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::findPosition().

5.3.2.19 template<int M, typename T> T& Go::GoGenericGrid< M, T >::operator[ ] (const int *const *coords*) `[inline]`

Returns a reference to a specified element in the array.

The element returned is the one having the indexes given as argument.

Parameters:

coords pointer to an M-sized range of integers, representing the indexes for the requested element.

Returns:

a reference to the element requested

Definition at line 423 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::findPosition().

Referenced by Go::GoGenericGrid< M, T >::blockRead(), and Go::GoGenericGrid< M, T >::fill-Value().

5.3.2.20 template<int M, typename T> void Go::GoGenericGrid< M, T >::permuteElements (const int * *permutation*) `[inline]`

Re-arranges the orders of the indexes to elements in the grid.

Permutes the order of the indexes to elements in the grid. The permutation is specified by the argument, which is a pointer to an array of M integers that should be a permutation of the numbers 0...M-1. (if this is not the case, an exception will be thrown in DEBUG mode). The integer at permutation[k] tells which position the k'th index (as specified by the 'this' grid, will have after the permutation.

Example: In a 3-indexed grid (i, j, k) we call permuteElements with the argument pointing to the integer range [0, 2, 1]. This means that the new indexation of the grid will be [i, k, j].

Definition at line 628 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::clone(), and Go::GoGenericGrid< M, T >::size().

Referenced by Go::GoGenericGrid< M, T >::cyclicPermute().

5.3.2.21 template<int M, typename T> void Go::GoGenericGrid< M, T >::permuteElements_memOpt (int * *permutation*) `[inline]`

Does the same as **permuteElements()**(p. 25), but using much less memory.

Does the same as **permuteElements()**(p. 25), but use much less extra memory to do so. The usual **permuteElements()**(p. 25) function needs extra memory equivalent to the total storage size of the grid to be permuted, while this function only uses about 1/64 of that size. However, the function can be much slower (measured to about 50% to 100% slower) than its memory-hungry equivalent. **NB**: This function has not been well tested, and may contain bugs!

Parameters:

permutation read explanation for **permuteElements()**(p. 25)

Definition at line 690 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::findPosition(), Go::GoGenericGrid< M, T >::rowlength(), and Go::GoGenericGrid< M, T >::size().

5.3.2.22 template<int M, typename T> void Go::GoGenericGrid< M, T >::read_ASCII (std::istream & *is*) `[inline]`

Reads a grid from a stream, using the ASCII format.

First the shape (length of each tensor direction) is read. Then the data are read from the stream into the memory block pointed to by the grid. Enough data is read to fill the whole volume of the grid.

Parameters:

is the stream from which to read the data

Definition at line 125 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::resize(), and Go::GoGenericGrid< M, T >::size().

5.3.2.23 template<int M, typename T> void Go::GoGenericGrid< M, T >::read_BINARY (std::istream & *is*) `[inline]`

Reads the grid from a stream, using BINARY format.

Same function as **read_ASCII()**(p. 26), but uses binary data instead of ASCII.

Parameters:

is the stream from which to read the data

Definition at line 171 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::resize(), and Go::GoGenericGrid< M, T >::size().

5.3.2.24 template<int M, typename T> virtual int Go::GoGenericGrid< M, T >::resize (const int ∗const *new_size*) `[inline, virtual]`

Redefining the shape of the grid (number of elements in each index).

Each index's number of elements is set to the corresponding number in the array pointed to by the argument.

Parameters:

new_size points to an array of M integers, specifying the number of elements in each index of the grid.

Returns:

the new total number of elements of the grid

Reimplemented in **Go::GoSelfcontainedGrid< M, T >** (p. 50).

Definition at line 369 of file GoGenericGrid.h.

Referenced by Go::GoSchoenbergApproximator< M, T >::approximate(), Go::GoLeastSquareApproximator< M, T >::approximate(), Go::GoKnotremovalApproximator< M, T

>::approximate(), Go::GoHybridApproximator< M, T >::approximate(), Go::GoGenericGrid< M, T >::clone(), Go::GoGenericGrid< M, T >::read_ASCII(), and Go::GoGenericGrid< M, T >::read_BINARY().

5.3.2.25 template<int M, typename T> int Go::GoGenericGrid< M, T >::rowlength (int *i*) const `[inline]`

Returns the number of elements along index *i*.

Returns:

the number of elements along index *i*

Definition at line 409 of file GoGenericGrid.h.

5.3.2.26 template<int M, typename T> const int∗ const Go::GoGenericGrid< M, T >::rowlength () const `[inline]`

Returns a pointer to the memory area specifying the shape of the grid.

By the *shape* of the grid, we mean the number of elements in each of the M indexes. This is stored in an array of M integers, which can be accessed by the pointer returned from this function.

Returns:

the above mentioned pointer

Definition at line 400 of file GoGenericGrid.h.

Referenced by Go::GoSchoenbergApproximator< M, T >::approximate(), Go::GoNeutral-Approximator< M, T >::approximate(), Go::GoLeastSquareApproximator< M, T >::approximate(), Go::GoKnotremovalApproximator< M, T >::approximate(), Go::Go-HybridApproximator< M, T >::approximate(), Go::GoGenericGrid< M, T >::blockRead(), Go::GoGenericGrid< M, T >::fillValue(), Go::GoGenericGrid< M, T >::flipDirection(), Go::GoGenericGrid< M, T >::operator+=(), Go::GoGenericGrid< M, T >::operator-=(), Go::GoGenericGrid< M, T >::permuteElements_memOpt(), and Go::GoBorrowedMVGrid< M, T >::subgrid().

5.3.2.27 template<int M, typename T> int Go::GoGenericGrid< M, T >::size () const `[inline]`

Returns the total number of elements in the grid.

Returns:

the total number of elements in the grid

Definition at line 315 of file GoGenericGrid.h.

Referenced by Go::GoSchoenbergApproximator< M, T >::approximate(), Go::GoLeast-SquareApproximator< M, T >::approximate(), Go::GoKnotremovalApproximator< M, T >::approximate(), Go::GoHybridApproximator< M, T >::approximate(), Go::GoGenericGrid< M, T >::datacopy(), Go::GoGenericGrid< M, T >::dumpCoefs(), Go::GoGenericGrid< M, T >::dumpCoefs_binary(), Go::GoGenericGrid< M, T >::fillValue(), Go::GoGenericGrid< M, T >::maxElem(), Go::GoGenericGrid< M, T >::minElem(), Go::GoGenericGrid< M, T >::operator

∗=(), Go::GoGenericGrid< M, T >::operator+=(), Go::GoGenericGrid< M, T >::operator-=(), Go::GoGenericGrid< M, T >::permuteElements(), Go::GoGenericGrid< M, T >::permute-Elements_memOpt(), Go::GoGenericGrid< M, T >::read_ASCII(), and Go::GoGenericGrid< M, T >::read_BINARY().

5.3.2.28 template<int M, typename T> void Go::GoGenericGrid< M, T >::write_ASCII (std::ostream & *os*) const `[inline]`

Writes the grid to a stream, using the ASCII format.

First the shape (length of each tensor direction) is written to the stream. Then an amount of data equal to the volume of the grid is dumped from the memory block pointed to by the grid, onto the stream.

Parameters:
os the stream to write the data to

Definition at line 152 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::dumpCoefs().

5.3.2.29 template<int M, typename T> void Go::GoGenericGrid< M, T >::write_BINARY (std::ostream & *os*) const `[inline]`

Writes the grid to a stream, using the BINARY format.

Same function as **write_ASCII()**(p. 28), but uses binary data instead of ASCII.

Parameters:
os the stream to write the data to

Definition at line 201 of file GoGenericGrid.h.

References Go::GoGenericGrid< M, T >::dumpCoefs_binary().

The documentation for this class was generated from the following file:

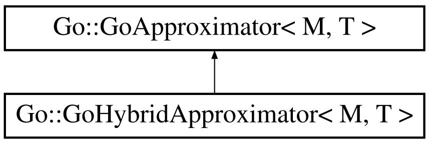
- GoGenericGrid.h

5.4 Go::GoHybridApproximator< M, T > Class Template Reference

Fits the spline to a set of sample values, using a scheme that minimizes the l2- norm of the error under certain, monotonity-preserving constraints on the control points.

`#include <GoHybridApproximator.h>`

Inheritance diagram for Go::GoHybridApproximator< M, T >::



Public Member Functions

- **GoHybridApproximator** (int resolution, int order, T rigidity, T equality_threshold, T epsilon)
- virtual void **approximate** (**GoBorrowedMVGrid**< M, T > *orig_data_array, **GoBorrowedMVGrid**< M, T > *cyclically_permuted_result_array, BsplineBasis &basis)

    A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

- virtual int **approximationSize** () const

    The size of the approximation (measured in number of control points) is equal to the number of control points that the user gave as input to the constructor for this object.

5.4.1 Detailed Description

template<int M, typename T> class Go::GoHybridApproximator< M, T >

Fits the spline to a set of sample values, using a scheme that minimizes the l2- norm of the error under certain, monotonity-preserving constraints on the control points.

The method uses ideas from both least square minimization and variation diminishing splines, and could be seen as a hybrid between the two. It tries to combine the accuracy obtained with least squares with the variation diminishing property. The method is entirely developed during the DINVIS project, and is described more thoroughly in the file `hybridscheme.pdf`

Definition at line 119 of file GoHybridApproximator.h.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 template<int M, typename T> Go::GoHybridApproximator< M, T >::GoHybridApproximator (int *resolution*, int *order*, T *rigidity*, T *equality_threshold*, T *epsilon*)

Parameters:

resolution number of control points in the resulting spline (degrees of freedom in the fitting process).

order order of the spline to be generated

rigidity factor that determines how much "spline rigidity" to mix into the equation. Higher values of this factor will result in a smoother spline with less oscillations. Lower values of this factor will result in a more accurate approximation.

equality_threshold parameter defining how big the difference of two successive sample point values need to have in order not to be defined as equal

epsilon a very small number, used to check when to end the internal optimizing loop.

Definition at line 167 of file GoHybridApproximator.h.

5.4.3 Member Function Documentation

5.4.3.1 template<int M, typename T> void Go::GoHybridApproximator< M, T >::approximate (GoBorrowedMVGrid< M, T > ∗ *orig_data_array*, GoBorrowedMVGrid< M, T > ∗ *cyclically_permuted_result_array*, BsplineBasis & *basis*) `[virtual]`

A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

All necessary parameters or modifiers to the encapsulated approximation algorithm should already have been set by the class' constructor or other methods. The sampled data is given in the form of an M-indexed grid. The approximation takes place along the index with the longest stride (the last index). In other words, if we have M indexes (0...M-1) and index m counts L_m elements, then we would consider the dataset a collection of L_{M-1} datapoints, where each datapoint has $D = Pi_{m=0}^{m-2} L_m$ components. The result is then a univariate spline with control points in R^D and a basis B. The resulting spline's coefficients is written to the grid pointed to by the second argument. It will be cyclically permuted by one step (see **GoGenericGrid::cyclicPermute()**(p. 20)) compared to the dataset, which allows us to *use it as input* for the **approximate()**(p. 30) function of the **GoApproximator**(p. 11) object that is to be applied on the next parameter. The generated basis will be returned in the last argument. The philosophy is that, for an M-variate dataset, applying the **approximate()**(p. 30) function M times (usually by M different **GoApproximator**(p. 11) objects), the first time on the sample data itself, later on the result from the last call of the **approximate()**(p. 30) function, then the resulting coefficient grid will be the control points for an M-variate spline approximating the original dataset. (We have to take care of the basis functions at each call too, in order to have a complete definition of the spline).

Note:

In practice, all this is taken care of by the **GoTensorProductSpline::fit()**(p. 59) function, so the only thing the user should usually be concerned about is the construction and definition of the desired approximator objects.

Parameters:

orig_data_array pointer to the gridded data that we want to 'fit' the spline to. The fitting will take place along the index with the longest stride, which is the last one.

cyclically_permuted_result_array Pointer to resulting spline's coefficient grid (to be filled out by the algorithm). It will be cyclically permuted (see **GoGenericGrid::cyclicPermute()** (p. 20)) by one step.

basis the spline basis resulting from this approximation.

Implements **Go::GoApproximator< M, T >** (p. 12).

Definition at line 203 of file GoHybridApproximator.h.

References Go::GoGenericGrid< M, T >::getDataPointer(), Go::GoGenericGrid< M, T >::resize(), Go::GoGenericGrid< M, T >::rowlength(), and Go::GoGenericGrid< M, T >::size().

5.4.3.2 template<int M, typename T> int Go::GoHybridApproximator< M, T >::approximationSize () const `[virtual]`

The size of the approximation (measured in number of control points) is equal to the number of control points that the user gave as input to the constructor for this object.

Returns:

The number of control points in the resulting spline after approximation

Implements **Go::GoApproximator< M, T >** (p. 13).

Definition at line 192 of file GoHybridApproximator.h.

The documentation for this class was generated from the following file:

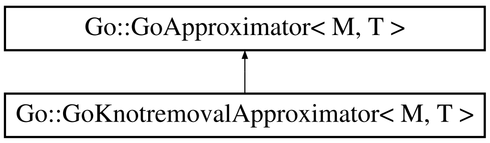
- GoHybridApproximator.h

5.5 Go::GoKnotremovalApproximator< M, T > Class Template Reference

Fits the spline to a set of sample values, using the knot removal scheme (Lyche/Mørken).

`#include <GoKnotremovalApproximator.h>`

Inheritance diagram for Go::GoKnotremovalApproximator< M, T >::


Go::GoApproximator< M, T >
Go::GoKnotremovalApproximator< M, T >

Public Member Functions

- **GoKnotremovalApproximator** (int order, T tolerance, T afctol, int fixed_derivs)
- virtual void **approximate** (**GoBorrowedMVGrid**< M, T > ∗orig_data_array, **GoBorrowedMVGrid**< M, T > ∗cyclically_permuted_result_array, BsplineBasis &basis)

  A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

- virtual int **approximationSize** () const

5.5.1 Detailed Description

template<int M, typename T> class Go::GoKnotremovalApproximator< M, T >

Fits the spline to a set of sample values, using the knot removal scheme (Lyche/Mørken).

Definition at line 28 of file GoKnotremovalApproximator.h.

5.5.2 Constructor & Destructor Documentation

5.5.2.1 template<int M, typename T> Go::GoKnotremovalApproximator< M, T >::GoKnotremovalApproximator (int *order*, T *tolerance*, T *afctol*, int *fixed_derivs*)

Parameters:

order the spline order of the approximation

tolerance a bound for the max error of the approximation as compared to the data input

afctol should be a number between 0 and 1, indicating how the tolerance is to be distributed between the two data reduction stages inherent to the algorithm. (It is usually best to keep it low, like 0.1).

fixed_derivs the number of derivatives that are not allowed to change on the boundaries of the spline.

Definition at line 66 of file GoKnotremovalApproximator.h.

5.5.3 Member Function Documentation

5.5.3.1 template<int M, typename T> void Go::GoKnotremovalApproximator< M, T >::approximate (GoBorrowedMVGrid< M, T > ∗ *orig_ data_ array*, GoBorrowedMVGrid< M, T > ∗ *cyclically_ permuted_ result_ array*, BsplineBasis & *basis*) `[virtual]`

A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

All necessary parameters or modifiers to the encapsulated approximation algorithm should already have been set by the class' constructor or other methods. The sampled data is given in the form of an M-indexed grid. The approximation takes place along the index with the longest stride (the last index). In other words, if we have M indexes (0...M-1) and index m counts L_m elements, then we would consider the dataset a collection of L_{M-1} datapoints, where each datapoint has $D = Pi_{m=0}^{m-2} L_m$ components. The result is then a univariate spline with control points in R^D and a basis B. The resulting spline's coefficients is written to the grid pointed to by the second argument. It will be cyclically permuted by one step (see **GoGenericGrid::cyclicPermute()**(p. 20)) compared to the dataset, which allows us to *use it as input* for the **approximate()**(p. 33) function of the **GoApproximator**(p. 11) object that is to be applied on the next parameter. The generated basis will be returned in the last argument. The philosophy is that, for an M-variate dataset, applying the **approximate()**(p. 33) function M times (usually by M different **GoApproximator**(p. 11) objects), the first time on the sample data itself, later on the result from the last call of the **approximate()**(p. 33) function, then the resulting coefficient grid will be the control points for an M-variate spline approximating the original dataset. (We have to take care of the basis functions at each call too, in order to have a complete definition of the spline).

Note:

In practice, all this is taken care of by the **GoTensorProductSpline::fit()**(p. 59) function, so the only thing the user should usually be concerned about is the construction and definition of the desired approximator objects.

Parameters:

orig_ data_ array pointer to the gridded data that we want to 'fit' the spline to. The fitting will take place along the index with the longest stride, which is the last one.

cyclically_ permuted_ result_ array Pointer to resulting spline's coefficient grid (to be filled out by the algorithm). It will be cyclically permuted (see **GoGenericGrid::cyclicPermute()**(p. 20)) by one step.

basis the spline basis resulting from this approximation.

Implements **Go::GoApproximator< M, T >** (p. 12).

Definition at line 93 of file GoKnotremovalApproximator.h.

References Go::GoGenericGrid< M, T >::clone(), Go::GoGenericGrid< M, T >::cyclicPermute(), Go::GoGenericGrid< M, T >::getDataPointer(), Go::GoGenericGrid< M, T >::resize(), Go::GoGenericGrid< M, T >::rowlength(), and Go::GoGenericGrid< M, T >::size().

5.5.3.2 template<int M, typename T> virtual int Go::GoKnotremovalApproximator< M, T >::approximationSize () const `[inline, virtual]`

The size of the approximation cannot be predicted with the knotremoval algorithm, and this function will therefore return a negative value (read the corresponding documentation on **GoApproximator::approximationSize()**(p. 13))

Returns:

A negative value. Multiplied with the number of datapoints that we want to approximate, we get an upper estimate on the number of resulting control points.

Implements **Go::GoApproximator< M, T >** (p. 13).

Definition at line 51 of file GoKnotremovalApproximator.h.

The documentation for this class was generated from the following file:

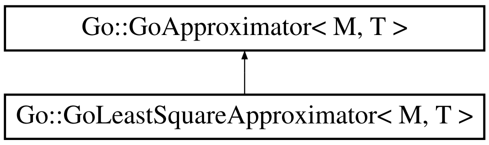
- GoKnotremovalApproximator.h

5.6 Go::GoLeastSquareApproximator< M, T > Class Template Reference

Fits the spline to a set of sample values, using least squares.

`#include <GoLeastSquareApproximator.h>`

Inheritance diagram for Go::GoLeastSquareApproximator< M, T >::


Go::GoApproximator< M, T >
Go::GoLeastSquareApproximator< M, T >

Public Member Functions

- **GoLeastSquareApproximator** (int resolution, int order, T rigidity)
- virtual void **approximate** (**GoBorrowedMVGrid**< M, T > *orig_data_array, **GoBorrowedMVGrid**< M, T > *cyclically_permuted_result_array, BsplineBasis &basis)

  A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

- virtual int **approximationSize** () const

  The size of the approximation (measured in number of control points) is equal to the number of control points that the user gave as input to the constructor for this object.

5.6.1 Detailed Description

template<int M, typename T> class Go::GoLeastSquareApproximator< M, T >

Fits the spline to a set of sample values, using least squares.

Definition at line 33 of file GoLeastSquareApproximator.h.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 template<int M, typename T> Go::GoLeastSquareApproximator< M, T >::GoLeastSquareApproximator (int *resolution*, int *order*, T *rigidity*)

Parameters:

resolution number of control points in the resulting spline (degrees of freedom in the fitting process).

order order of the spline to be generated

rigidity factor that determines how much "spline rigidity" to mix into the equation. Higher values of this factor will result in a smoother spline with less oscillations. Lower values of this factor will result in a more accurate approximation. The 'rigidity' parameter expresses the ratio of the frobenius norms of the smoothing matrix and of the system matrix. A rigidity of 0 eliminates the smoothing matrix altogether.

Definition at line 71 of file GoLeastSquareApproximator.h.

5.6.3 Member Function Documentation

5.6.3.1 template<int M, typename T> void Go::GoLeastSquareApproximator< M, T >::approximate (GoBorrowedMVGrid< M, T > * *orig_data_array*, GoBorrowedMVGrid< M, T > * *cyclically_permuted_result_array*, BsplineBasis & *basis*) `[virtual]`

A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

All necessary parameters or modifiers to the encapsulated approximation algorithm should already have been set by the class' constructor or other methods. The sampled data is given in the form of an M-indexed grid. The approximation takes place along the index with the longest stride (the last index). In other words, if we have M indexes (0...M-1) and index m counts L_m elements, then we would consider the dataset a collection of L_{M-1} datapoints, where each datapoint has $D = Pi_{m=0}^{m-2} L_m$ components. The result is then a univariate spline with control points in R^D and a basis B. The resulting spline's coefficients is written to the grid pointed to by the second argument. It will be cyclically permuted by one step (see **GoGenericGrid::cyclicPermute()**(p. 20)) compared to the dataset, which allows us to ***use it as input*** for the **approximate()**(p. 36) function of the **GoApproximator**(p. 11) object that is to be applied on the next parameter. The generated basis will be returned in the last argument. The philosophy is that, for an M-variate dataset, applying the **approximate()**(p. 36) function M times (usually by M different **GoApproximator**(p. 11) objects), the first time on the sample data itself, later on the result from the last call of the **approximate()**(p. 36) function, then the resulting coefficient grid will be the control points for an M-variate spline approximating the original dataset. (We have to take care of the basis functions at each call too, in order to have a complete definition of the spline).

Note:

In practice, all this is taken care of by the **GoTensorProductSpline::fit()**(p. 59) function, so the only thing the user should usually be concerned about is the construction and definition of the desired approximator objects.

Parameters:

orig_data_array pointer to the gridded data that we want to 'fit' the spline to. The fitting will take place along the index with the longest stride, which is the last one.

cyclically_permuted_result_array Pointer to resulting spline's coefficient grid (to be filled out by the algorithm). It will be cyclically permuted (see **GoGenericGrid::cyclicPermute()**(p. 20)) by one step.

basis the spline basis resulting from this approximation.

Implements **Go::GoApproximator< M, T >** (p. 12).

Definition at line 86 of file GoLeastSquareApproximator.h.

References Go::GoGenericGrid< M, T >::findPosition(), Go::GoGenericGrid< M, T >::getDataPointer(), Go::GoGenericGrid< M, T >::resize(), Go::GoGenericGrid< M, T >::rowlength(), and Go::GoGenericGrid< M, T >::size().

5.6.3.2 template<int M, typename T> virtual int Go::GoLeastSquare-Approximator< M, T >::approximationSize () const `[inline, virtual]`

The size of the approximation (measured in number of control points) is equal to the number of control points that the user gave as input to the constructor for this object.

Returns:

The number of control points in the resulting spline after approximation

Implements **Go::GoApproximator< M, T >** (p. 13).

Definition at line 57 of file GoLeastSquareApproximator.h.

The documentation for this class was generated from the following file:

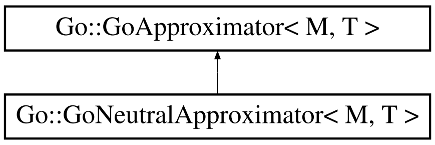
- GoLeastSquareApproximator.h

5.7 Go::GoNeutralApproximator< M, T > Class Template Reference

This approximator is "neutral".

`#include <GoNeutralApproximator.h>`

Inheritance diagram for Go::GoNeutralApproximator< M, T >::



Public Member Functions

- virtual void **approximate** (**GoBorrowedMVGrid**< M, T > ∗orig_data_array, **GoBorrowedMVGrid**< M, T > ∗cyclically_permuted_result_array, BsplineBasis &basis)

  A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

- virtual int **approximationSize** () const

5.7.1 Detailed Description

template<int M, typename T> class Go::GoNeutralApproximator< M, T >

This approximator is "neutral".

That means that the resulting control point grid doesn't differ from the raw data in any way (except for being cyclically permuted by one step, according to the specification of the GoApproxmator::approximate() function. Each point of the raw data is seen as a control point for a piecewise constant spline, so that the data keeps its original character even though represented by a spline. This class can be used in for the **GoTensorProductSpline**(p. 51) class to "fill in" approximators for those parameters where the user does not want any altering of the data.

Definition at line 41 of file GoNeutralApproximator.h.

5.7.2 Member Function Documentation

5.7.2.1 template<int M, typename T> void Go::GoNeutralApproximator< M, T >::approximate (GoBorrowedMVGrid< M, T > ∗ *orig_data_array*, GoBorrowedMVGrid< M, T > ∗ *cyclically_permuted_result_array*, BsplineBasis & *basis*) `[virtual]`

A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

All necessary parameters or modifiers to the encapsulated approximation algorithm should already have been set by the class' constructor or other methods. The sampled data is given in the form of an M-indexed grid. The approximation takes place along the index with the longest stride (the last index). In other words, if we have M indexes (0...M-1) and index m counts L_m elements, then we would consider the dataset a collection of L_{M-1} datapoints, where each datapoint has $D = Pi_{m=0}^{m-2} L_m$ components. The result is then a univariate spline with control points in R^D and a basis B. The resulting spline's coefficients is written to the grid pointed to by the second argument. It will be cyclically permuted by one step (see **GoGenericGrid::cyclicPermute()**(p. 20)) compared to the dataset, which allows us to ***use it as input*** for the **approximate()**(p. 38) function of the **GoApproximator**(p. 11) object that is to be applied on the next parameter. The generated basis will be returned in the last argument. The philosophy is that, for an M-variate dataset, applying the **approximate()**(p. 38) function M times (usually by M different **GoApproximator**(p. 11) objects), the first time on the sample data itself, later on the result from the last call of the **approximate()**(p. 38) function, then the resulting coefficient grid will be the control points for an M-variate spline approximating the original dataset. (We have to take care of the basis functions at each call too, in order to have a complete definition of the spline).

Note:

In practice, all this is taken care of by the **GoTensorProductSpline::fit()**(p. 59) function, so the only thing the user should usually be concerned about is the construction and definition of the desired approximator objects.

Parameters:

orig_data_array pointer to the gridded data that we want to 'fit' the spline to. The fitting will take place along the index with the longest stride, which is the last one.

cyclically_permuted_result_array Pointer to resulting spline's coefficient grid (to be filled out by the algorithm). It will be cyclically permuted (see **GoGenericGrid::cyclicPermute()**(p. 20)) by one step.

basis the spline basis resulting from this approximation.

Implements **Go::GoApproximator< M, T >** (p. 12).

Definition at line 71 of file GoNeutralApproximator.h.

References Go::GoGenericGrid< M, T >::clone(), Go::GoGenericGrid< M, T >::cyclicPermute(), and Go::GoGenericGrid< M, T >::rowlength().

5.7.2.2 template<int M, typename T> virtual int Go::GoNeutralApproximator< M, T >::approximationSize () const `[inline, virtual]`

With this class, the result does not differ from the input data, except from the cyclic permutation that always takes place (according to the specification of the **GoApproximator::approximate()**(p. 12) function). The size is thus dependent (and equal to) the size of the input data. Therefore, in accordance with the specification of **GoApproximator::approximationSize()**(p. 13), this function always returns -1.

Returns:

-1 (read **GoApproximator::approximationSize()**(p. 13) to understand why)

Implements **Go::GoApproximator< M, T >** (p. 13).

Definition at line 59 of file GoNeutralApproximator.h.

The documentation for this class was generated from the following file:

- GoNeutralApproximator.h

5.8 Go::GoScalableGrid< T > Class Template Reference

A multiindexed grid whose number of indexes can be changed at runtime.

`#include <GoScalableGrid.h>`

Public Member Functions

- **GoScalableGrid** ()

  Default constructor making a grid with no indexes and no size.

- **GoScalableGrid** (int num_dims, int ∗rowlengths)

  Constructor creating a grid with a certain shape.

- void **reshape** (int num_dims, const int ∗rowlengths, T fill_value)

  Change the shape of the grid and fills it with a specified value.

- void **reshape** (int num_dims, const int ∗rowlengths)

  Change the shape of the grid, and fills it with an unspecified value.

- int **numDims** () const

  Return the current number of indexes in the grid.

- int **rowlength** (int dim) const
- const int ∗const **rowlength** () const

  Return a const pointer to the memory area specifying the shape of the grid.

- const T ∗ **getDataPointer** () const

  Return a pointer to the element storage memory area owned and used by the grid.

- void **clone** (**Go::GoScalableGrid**< T > &newgrid) const

  Makes the argument grid a copy of itself.

- T & **operator[ ]** (const int ∗const coords)

  Returns a reference to a specified element in the array.

- const T & **operator[ ]** (const int ∗const coords) const

  Returns a const reference to a specified element in the array.

- int **size** () const

  Returns the total number of elements in the grid.

- void **permuteElements** (const int ∗permutation)

  Re-arranges the orders of the indexes to elements in the grid.

5.8.1 Detailed Description

template<typename T> class Go::GoScalableGrid< T >

A multiindexed grid whose number of indexes can be changed at runtime.

It owns the memory area where elements are stored, which makes copying expensive but memory management easier.

Definition at line 30 of file GoScalableGrid.h.

5.8.2 Constructor & Destructor Documentation

5.8.2.1 template<typename T> Go::GoScalableGrid< T >::GoScalableGrid () `[inline]`

Default constructor making a grid with no indexes and no size.

Even though it is not immediately useful, it can later be reshaped or assigned into.

Definition at line 38 of file GoScalableGrid.h.

5.8.2.2 template<typename T> Go::GoScalableGrid< T >::GoScalableGrid (int *num_dims*, int * *rowlengths*) `[inline]`

Constructor creating a grid with a certain shape.

Parameters:

num_dims number of indexes for the multiindexed grid

rowlengths points to an array of integers. The array must have 'num_dims' elements, specifying the lengths of each index of the multiindexed grid to be created.

Definition at line 47 of file GoScalableGrid.h.

References Go::GoScalableGrid< T >::reshape().

5.8.3 Member Function Documentation

5.8.3.1 template<typename T> void Go::GoScalableGrid< T >::clone (Go::GoScalableGrid< T > & *newgrid*) const `[inline]`

Makes the argument grid a copy of itself.

Parameters:

newgrid the grid which will become a copy of the 'this' grid

Definition at line 141 of file GoScalableGrid.h.

References Go::GoScalableGrid< T >::getDataPointer(), Go::GoScalableGrid< T >::numDims(), Go::GoScalableGrid< T >::reshape(), Go::GoScalableGrid< T >::rowlength(), and Go::GoScalableGrid< T >::size().

Referenced by Go::GoScalableGrid< T >::permuteElements().

5.8.3.2 template<typename T> const T∗ Go::GoScalableGrid< T >::getDataPointer () const [inline]

Return a pointer to the element storage memory area owned and used by the grid.

Returns:
a pointer to the element storage memory area used by the grid

Definition at line 127 of file GoScalableGrid.h.

Referenced by Go::GoScalableGrid< T >::clone().

5.8.3.3 template<typename T> int Go::GoScalableGrid< T >::numDims () const [inline]

Return the current number of indexes in the grid.

Returns:
the current number of indexes in the grid

Definition at line 88 of file GoScalableGrid.h.

Referenced by Go::GoScalableGrid< T >::clone(), Go::GoScalableGrid< T >::permuteElements(), and Go::GoScalableGrid< T >::rowlength().

5.8.3.4 template<typename T> const T& Go::GoScalableGrid< T >::operator[ ] (const int ∗const *coords*) const [inline]

Returns a const reference to a specified element in the array.

The element returned is the one having the indexes given as argument.

Parameters:
coords pointer to an M-sized range of integers, representing the indexes for the requested element.

Returns:
a const reference to the element requested

Definition at line 171 of file GoScalableGrid.h.

5.8.3.5 template<typename T> T& Go::GoScalableGrid< T >::operator[ ] (const int ∗const *coords*) [inline]

Returns a reference to a specified element in the array.

The element returned is the one having the indexes given as argument.

Parameters:
coords pointer to an M-sized range of integers, representing the indexes for the requested element.

Returns:
a reference to the element requested

Definition at line 157 of file GoScalableGrid.h.

5.8.3.6 template<typename T> void Go::GoScalableGrid< T >::permuteElements (const int * *permutation*) `[inline]`

Re-arranges the orders of the indexes to elements in the grid.

Permutes the order of the indexes to elements in the grid. The permutation is specified by the argument, which is a pointer to an array of M integers that should be a permutation of the numbers 0...M-1. (if this is not the case, an exception will be thrown in DEBUG mode). The integer at permutation[k] tells which position the k'th index (as specified by the 'this' grid, will have after the permutation.

Example: In a 3-indexed grid (i, j, k) we call permuteElements with the argument pointing to the integer range [0, 2, 1]. This means that the new indexation of the grid will be [i, k, j].

Definition at line 204 of file GoScalableGrid.h.

References Go::GoScalableGrid< T >::clone(), Go::GoScalableGrid< T >::numDims(), and Go::GoScalableGrid< T >::size().

5.8.3.7 template<typename T> void Go::GoScalableGrid< T >::reshape (int *num_dims*, const int * *rowlengths*) `[inline]`

Change the shape of the grid, and fills it with an unspecified value.

Parameters:

num_dims specifies how many indexes the multiindexed grid should have

rowlengths should point to an array of 'num_dims' integers, specifying the lengths of each index of the multiindexed grid

Definition at line 79 of file GoScalableGrid.h.

References Go::GoScalableGrid< T >::reshape().

5.8.3.8 template<typename T> void Go::GoScalableGrid< T >::reshape (int *num_dims*, const int * *rowlengths*, T *fill_value*) `[inline]`

Change the shape of the grid and fills it with a specified value.

Parameters:

num_dims specifies how many indexes the multiindexed grid should have

rowlengths should point to an array of 'num_dims' integers, specifying the lengths of each index of the multiindexed grid

fill_value the value to fill the all elements of the grid with after reshaping it

Definition at line 59 of file GoScalableGrid.h.

Referenced by Go::GoScalableGrid< T >::clone(), Go::GoScalableGrid< T >::GoScalableGrid(), and Go::GoScalableGrid< T >::reshape().

5.8.3.9 template<typename T> const int* const Go::GoScalableGrid< T >::rowlength () const `[inline]`

Return a const pointer to the memory area specifying the shape of the grid.

The shape of the grid is internally defined as an array of integers. The array has a length equal to the number of indices in the multiindexed grid, and each entry in the array defined the corresponding length of that index. This function return a pointer to this array, so that the shape of an unknown grid can be inspected from outside

Returns:
a pointer to the memory area specifying the shape of the grid

Definition at line 115 of file GoScalableGrid.h.

Referenced by Go::GoScalableGrid< T >::clone().

5.8.3.10 template<typename T> int Go::GoScalableGrid< T >::rowlength (int *dim*) const `[inline]`

Return the length of an index

Parameters:
dim the index we want to know the length of

Returns:
the length of the specified index

Definition at line 98 of file GoScalableGrid.h.

References Go::GoScalableGrid< T >::numDims().

5.8.3.11 template<typename T> int Go::GoScalableGrid< T >::size () const `[inline]`

Returns the total number of elements in the grid.

This number is obtained by multiplying together the lengths of all the indexes

Returns:
the total number of elements in the grid

Definition at line 183 of file GoScalableGrid.h.

Referenced by Go::GoScalableGrid< T >::clone(), and Go::GoScalableGrid< T >::permuteElements().

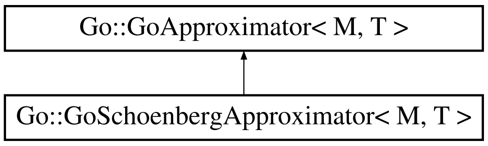
The documentation for this class was generated from the following file:

- GoScalableGrid.h

5.9 Go::GoSchoenbergApproximator< M, T > Class Template Reference

`#include <GoSchoenbergApproximator.h>`

Inheritance diagram for Go::GoSchoenbergApproximator< M, T >::


Go::GoApproximator< M, T >
Go::GoSchoenbergApproximator< M, T >

Public Member Functions

- **GoSchoenbergApproximator** (int resolution, int order)
- virtual void **approximate** (**GoBorrowedMVGrid**< M, T > ∗orig_data_array, **GoBorrowedMVGrid**< M, T > ∗cyclically_permuted_result_array, BsplineBasis &basis)

  A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

- virtual int **approximationSize** () const

  The size of the approximation (measured in number of control points) is equal to the number of control points that the user gave as input to the constructor for this object.

5.9.1 Detailed Description

template<int M, typename T> class Go::GoSchoenbergApproximator< M, T >

Fits the spline to a set of sample values, using Schoenberg's variation diminishing spline scheme.

Definition at line 42 of file GoSchoenbergApproximator.h.

5.9.2 Constructor & Destructor Documentation

5.9.2.1 template<int M, typename T> Go::GoSchoenbergApproximator< M, T >::GoSchoenbergApproximator (int *resolution*, int *order*)

Parameters:

resolution number of control points in the resulting spline (degrees of freedom in the fitting process).

order order of the spline to be generated

Definition at line 74 of file GoSchoenbergApproximator.h.

5.9.3 Member Function Documentation

5.9.3.1 template<int M, typename T> void Go::GoSchoenbergApproximator< M, T >::approximate (GoBorrowedMVGrid< M, T > * *orig_data_array*, GoBorrowedMVGrid< M, T > * *cyclically_permuted_result_array*, BsplineBasis & *basis*) [virtual]

A call to this function invokes the approximation algorithm encapsulated in this class on a set of gridded sample data.

All necessary parameters or modifiers to the encapsulated approximation algorithm should already have been set by the class' constructor or other methods. The sampled data is given in the form of an M-indexed grid. The approximation takes place along the index with the longest stride (the last index). In other words, if we have M indexes (0...M-1) and index m counts L_m elements, then we would consider the dataset a collection of L_{M-1} datapoints, where each datapoint has $D = Pi_{m=0}^{m-2} L_m$ components. The result is then a univariate spline with control points in R^D and a basis B. The resulting spline's coefficients is written to the grid pointed to by the second argument. It will be cyclically permuted by one step (see **GoGenericGrid::cyclicPermute()**(p. 20)) compared to the dataset, which allows us to *use it as input* for the **approximate()**(p. 46) function of the **GoApproximator**(p. 11) object that is to be applied on the next parameter. The generated basis will be returned in the last argument. The philosophy is that, for an M-variate dataset, applying the **approximate()**(p. 46) function M times (usually by M different **GoApproximator**(p. 11) objects), the first time on the sample data itself, later on the result from the last call of the **approximate()**(p. 46) function, then the resulting coefficient grid will be the control points for an M-variate spline approximating the original dataset. (We have to take care of the basis functions at each call too, in order to have a complete definition of the spline).

Note:

In practice, all this is taken care of by the **GoTensorProductSpline::fit()**(p. 59) function, so the only thing the user should usually be concerned about is the construction and definition of the desired approximator objects.

Parameters:

orig_data_array pointer to the gridded data that we want to 'fit' the spline to. The fitting will take place along the index with the longest stride, which is the last one.

cyclically_permuted_result_array Pointer to resulting spline's coefficient grid (to be filled out by the algorithm). It will be cyclically permuted (see **GoGenericGrid::cyclicPermute()**(p. 20)) by one step.

basis the spline basis resulting from this approximation.

Implements **Go::GoApproximator< M, T >** (p. 12).

Definition at line 88 of file GoSchoenbergApproximator.h.

References Go::GoGenericGrid< M, T >::getDataPointer(), Go::GoGenericGrid< M, T >::resize(), Go::GoGenericGrid< M, T >::rowlength(), and Go::GoGenericGrid< M, T >::size().

5.9.3.2 template<int M, typename T> virtual int Go::GoSchoenbergApproximator< M, T >::approximationSize () const [inline, virtual]

The size of the approximation (measured in number of control points) is equal to the number of control points that the user gave as input to the constructor for this object.

Returns:

The number of control points in the resulting spline after approximation

Implements **Go::GoApproximator< M, T >** (p. 13).

Definition at line 61 of file GoSchoenbergApproximator.h.

The documentation for this class was generated from the following file:

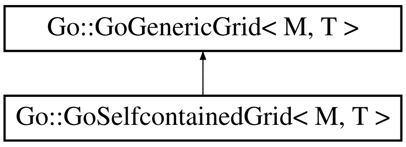
- GoSchoenbergApproximator.h

5.10 Go::GoSelfcontainedGrid< M, T > Class Template Reference

This is a multiindexed grid that derives from **GoGenericGrid**(p. 17). It owns its data, which makes copying an expensive process.

`#include <GoSelfcontainedGrid.h>`

Inheritance diagram for Go::GoSelfcontainedGrid< M, T >::



Public Member Functions

- **GoSelfcontainedGrid** ()

  Constructor making an empty, multiindexed grid with M indices, where each index is of length zero.

- **GoSelfcontainedGrid** (const int ∗const rowlength)

  Constructor making a multiindexed grid with M indices, where each index has a specified length.

- **GoSelfcontainedGrid** (const **GoSelfcontainedGrid** &rhs)

  Copy constructor (potentially expensive for large grids).

- **GoSelfcontainedGrid** & **operator=** (const **GoSelfcontainedGrid** &rhs)

  Exception-safe assignment operator (potentially expensive for large grids).

- virtual void **swap** (**GoSelfcontainedGrid** &rhs)

  Rapidly swap two grid.

- virtual int **resize** (const int ∗const new_size)

  Redefining the shape of the grid (number of elements in each index).

- const **GoBorrowedMVGrid**< M, T > **borrowedCopy** () const

  The grid returned can be considered a 'copy' of this grid, but it shares the same element storage area.

5.10.1 Detailed Description

template<int M, typename T> class Go::GoSelfcontainedGrid< M, T >

This is a multiindexed grid that derives from **GoGenericGrid**(p. 17). It owns its data, which makes copying an expensive process.

Definition at line 26 of file GoSelfcontainedGrid.h.

5.10.2 Constructor & Destructor Documentation

5.10.2.1 template<int M, typename T> Go::GoSelfcontainedGrid< M, T >::GoSelfcontainedGrid () `[inline]`

Constructor making an empty, multiindexed grid with M indices, where each index is of length zero.

The created grid is immediately useless, since it has size 0. It can however be assigned to other grids, resized, etc.

Definition at line 34 of file GoSelfcontainedGrid.h.

5.10.2.2 template<int M, typename T> Go::GoSelfcontainedGrid< M, T >::GoSelfcontainedGrid (const int *const *rowlength*) `[inline]`

Constructor making a multiindexed grid with M indices, where each index has a specified length.

Parameters:

rowlength pointer to an array of M integers specifying the length of the respective indexes of the grid to be created

Definition at line 53 of file GoSelfcontainedGrid.h.

References Go::GoSelfcontainedGrid< M, T >::resize().

5.10.2.3 template<int M, typename T> Go::GoSelfcontainedGrid< M, T >::GoSelfcontainedGrid (const GoSelfcontainedGrid< M, T > & *rhs*) `[inline]`

Copy constructor (potentially expensive for large grids).

Parameters:

rhs grid to copy

Definition at line 64 of file GoSelfcontainedGrid.h.

References Go::GoGenericGrid< M, T >::clone().

5.10.3 Member Function Documentation

5.10.3.1 template<int M, typename T> const GoBorrowedMVGrid<M, T> Go::GoSelfcontainedGrid< M, T >::borrowedCopy () const `[inline]`

The grid returned can be considered a 'copy' of this grid, but it shares the same element storage area.

The ownership of the element storage memory area remains within 'this' grid, while the resulting Go::GoBorrowedMVGrid<M, T> only points to the same memory.

Returns:

a grid with the same shape that uses the same memory storage area as 'this' grid

Definition at line 132 of file GoSelfcontainedGrid.h.

5.10.3.2 template<int M, typename T> GoSelfcontainedGrid& Go::GoSelfcontainedGrid< M, T >::operator= (const GoSelfcontainedGrid< M, T > & *rhs*) `[inline]`

Exception-safe assignment operator (potentially expensive for large grids).

Parameters:
rhs grid to copy

Definition at line 73 of file GoSelfcontainedGrid.h.

References Go::GoSelfcontainedGrid< M, T >::swap().

5.10.3.3 template<int M, typename T> virtual int Go::GoSelfcontainedGrid< M, T >::resize (const int *const *new_size*) `[inline, virtual]`

Redefining the shape of the grid (number of elements in each index).

Each index's number of elements is set to the corresponding number in the array pointed to by the argument.

Parameters:
new_size points to an array of M integers, specifying the number of elements in each index of the grid.

Returns:
the new total number of elements of the grid

Reimplemented from **Go::GoGenericGrid< M, T >** (p. 26).

Definition at line 98 of file GoSelfcontainedGrid.h.

Referenced by Go::GoSelfcontainedGrid< M, T >::GoSelfcontainedGrid().

5.10.3.4 template<int M, typename T> virtual void Go::GoSelfcontainedGrid< M, T >::swap (GoSelfcontainedGrid< M, T > & *rhs*) `[inline, virtual]`

Rapidly swap two grid.

Parameters:
rhs the grid to swap with

Definition at line 85 of file GoSelfcontainedGrid.h.

References Go::GoSelfcontainedGrid< M, T >::data_storage_.

Referenced by Go::GoSelfcontainedGrid< M, T >::operator=().

The documentation for this class was generated from the following file:

- GoSelfcontainedGrid.h

5.11 Go::GoTensorProductSpline< M, T > Class Template Reference

This is a template class that represent a multivariate spline with **M** parameters and coefficients of type **T**.

```
#include <GoTensorProductSpline.h>
```

Public Member Functions

- **GoTensorProductSpline** ()

  This constructor does not make a valid **GoTensorProductSpline**(p. 51) *object.*

- **GoTensorProductSpline** (T ∗storage_area)

  This constructor does not make a valid **GoTensorProductSpline**(p. 51) *object.*

- **GoTensorProductSpline** (const **GoBorrowedMVGrid**< M, T > &coefgrid, const int ∗const order, bool is_kreg)

  Constructor that creates a tensor product B-spline whose coefficents is the M-dimensional grid given as the first argument.

- **GoTensorProductSpline** (const **GoBorrowedMVGrid**< M, T > &coefs, const int ∗const order, const T ∗const start_param_vals, const T ∗const end_param_vals, bool k_reg)

  Constructor that creates a tensor product B-spline whose coefficients is the M-dimensional grid given as the first argument.

- **GoTensorProductSpline** (const **GoBorrowedMVGrid**< M, T > &coefs, const std::vector< double > ∗knotvectors, const int ∗const order)

  Constructor that creates a tensor product B-spline whose coefficients is the M-dimensional grid given as the first argument.

- **GoTensorProductSpline** (const **GoTensorProductSpline**< M, T > &rhs)

  Copy constructor (weak - the new **GoTensorProductSpline**(p. 51) *will share the same coefficient storage area as 'rhs').*

- void **clone** (**GoTensorProductSpline**< M, T > &rhs) const

  Clones itself into the argument spline, who becomes a hard copy of 'this' spline.

- void **read_ASCII** (std::istream &is)

  Read the object's contents from an input stream as ASCII data.

- void **write_ASCII** (std::ostream &os) const

  Write the object's contents to an output stream as ASCII data.

- void **read_BINARY** (std::istream &is)

  Read the object's contents from an input stream as BINARY data.

- void **write_BINARY** (std::ostream &os) const

  Write the object's contents to an output stream as BINARY data.

- Array< T, 2 *M > **paramspan** () const

  Returns an array containing the start and end parameter values.

- T **point** (const double *const params) const

  Evaluate the spline for a certain choice of parameter values.

- void **setUniformKnotVector** (int tensor_dir, T start, T end, int order, bool is_kreg)

  Redefines the basis for the specified parameter using a uniform knotvector and a given order.

- void **rescaleKnotVector** (int tensor_dir, T start_value, T end_value)

  Rescales the knotvector for the given parameter.

- std::vector< double > **getKnotVector** (int tensor_dir) const

  Returns the knotvector corresponding to the specified parameter.

- **GoBorrowedMVGrid**< M, T > **coefficientGrid** () const

  Returns the coefficient grid.

- void **setDataPointer** (T *p)

  Sets the spline's coefficient storage area to start at the memory address given by 'p'.

- void **evalSurf** (int no_u, int no_v, int par_dir_1, int par_dir_2, double start_param_1, double start_param_2, double end_param_1, double end_param_2, bool reverse_dir_1, bool reverse_dir_2, const double *const fixed, T *res) const

  Fast evaluation of a whole, bivariate surface on the spline.

- int **findRange** (int par_dir, double value) const

  For a given parameter, identifies the interval on the corresponding knotvector where the specified parameter value is located.

- **GoTensorProductSpline**< M-1, T > **integrate** (int dir, double *domain_start, double *domain_end, T *storage_area) const

  Integrates the spline along a specified parameter, producing a spline with one parameter less.

- **GoTensorProductSpline**< M-1, T > **reduce** (int dir, T param_val, T *data_area) const

  Making a spline with one less parameter from the original, obtained by fixing this parameter's value.

- **GoTensorProductSpline**< M, T > **extract** (T *storage_area, int *knot_start_indexes, int *knot_end_indexes) const

  Creates a new **GoTensorProductSpline**(p. 51) *whose parameter domain is a subset of the original spline.*

- void **insertKnots** (int dir, const std::vector< T > &new_knots)

  Knot insertion in a given parameter.

- void **raiseOrderTo** (int new_order, int dir)

  Raise the order of the basis for a specified parameter.

- void **evalVolume** (std::pair< double *, double * > *sample_points, T *res) const

Fast evaluation of the whole spline object for many parameter values.

- void **evalVolume** (int *resolutions, double *range_start, double *range_end, T *res) const

  Fast evaluation of the whole spline object for many parameter values.

- void **fullEvaluate** (**GoGenericGrid**< M, T > &res) const

  Fast evaluation of the whole spline object, at equidistant parameter values.

- void **fit** (const **GoGenericGrid**< M, T > &rawdata, Array< **GoApproximator**< M, T > *, M > &methods, **GoGenericGrid**< M, T > &errors, bool calculate_errors=true)

  Fit the spline to a grid of sample values, using a specified approximation method.

- int **size** () const

  Return the number of spline coefficients.

- int **rowlength** (int i) const

  Return the number of coefficients for the specified parameter.

- int **order** (int i) const

  Returns the order of the spline for a given parameter.

- const BsplineBasis & **getBasis** (int i) const

  Returns the basis for a specified parameter.

- bool **operator==** (const **GoTensorProductSpline**< M, T > &rhs) const

  Equality operator.

- void **cyclicPermute** (int steps)

  Cyclically permute the parameters of the spline.

5.11.1 Detailed Description

template<int M, typename T> class Go::GoTensorProductSpline< M, T >

This is a template class that represent a multivariate spline with **M** parameters and coefficients of type **T**.

Notes:

- The spline is defined by a set of M *basises* (one for each parameter), and a set of *coefficients*. Each basis consists of a set of *basis functions*, and the coefficients are located in an M-indexed grid (of the class **GoBorrowedMVGrid**(p. 14)).
- This means that the spline carries information about the basises it is using, but is only borrowing the memory area where the coefficients are being held (since they are stored in a grid that does not own its elements). In other words, the user is responsible for making sure that the spline always has enough memory to store its necessary coefficients. This is especially delicate in situations where the spline changes its number of coefficients, as in knot insertion.

- The **GoTensorProductSpline**(p. 51) is a mapping from R^M to R, and several objects of this class is therefore necessary if the user wants to describe a spline object lying in R^D, $D > 1$. A utility function, CreateSplineSurface is available for making a Go::SplineSurface with arbitrary dimension out of a set of bivariate **GoTensorProductSpline**(p. 51) with similar basises.

Definition at line 60 of file GoTensorProductSpline.h.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 template<int M, typename T> Go::GoTensorProductSpline< M, T >::GoTensorProductSpline ()

This constructor does not make a valid **GoTensorProductSpline**(p. 51) object.

However, it can later be assigned another object and thus become valid.

5.11.2.2 template<int M, typename T> Go::GoTensorProductSpline< M, T >::GoTensorProductSpline (T * *storage_area*)

This constructor does not make a valid **GoTensorProductSpline**(p. 51) object.

However, the pointer to the coefficient storage area is set, so that it is possible to have another **GoTensorProductSpline**(p. 51) clone itself into it.

Parameters:

storage_area pointer to the coefficient storage area

5.11.2.3 template<int M, typename T> Go::GoTensorProductSpline< M, T >::GoTensorProductSpline (const GoBorrowedMVGrid< M, T > & *coefgrid*, const int *const *order*, bool *is_kreg*)

Constructor that creates a tensor product B-spline whose coefficents is the M-dimensional grid given as the first argument.

The basis functions in each parameter will be created with the given order and with a uniform knotvector whose range spans from 0 to 1, and that is k-regular or not, depending on the last argument. The 'order' argument should point to a memory location where the M consecutive spline orders are stored.

Parameters:

coefgrid the coefficient grid

order pointer to an M-sized array of integers representing the spline orders

is_kreg dictates whether or not the knotvectors should be k-regulars.

5.11.2.4 template<int M, typename T> Go::GoTensorProductSpline< M, T >::GoTensorProductSpline (const GoBorrowedMVGrid< M, T > & *coefs*, const int *const *order*, const T *const *start_param_vals*, const T *const *end_param_vals*, bool *k_reg*)

Constructor that creates a tensor product B-spline whose coefficients is the M-dimensional grid given as the first argument.

The 'order' argument is a pointer to M consecutive integers that give the order in each parameter, while the following two arrays, 'start_param_val' and 'end_param_val' give the start and end parameter values for the knot vector in each parameter. The knotvectors will be uniform, and the last boolean argument 'k_reg' defines whether they will have multiple knots in the end ('k_reg' = true).

Parameters:

coefs the coefficient grid

order pointer to an M-sized array of integers representing the spline orders

start_param_vals pointer to an M-sized array dictating the start values for each parameter range.

end_param_vals pointer to an M-sized array dictating the end values for each parameter range.

k_reg whether or not the knotvectors should be k-regular.

5.11.2.5 template<int M, typename T> Go::GoTensorProductSpline< M, T >::GoTensorProductSpline (const GoBorrowedMVGrid< M, T > & *coefs*, const std::vector< double > ∗ *knotvectors*, const int ∗const *order*)

Constructor that creates a tensor product B-spline whose coefficients is the M-dimensional grid given as the first argument.

The knotvectors for each parameter are defined by the user through the M consecutive STL-vectors pointed to by the second argument. The argument named 'order' points to a M-sized range of integers specifying the spline order in each parameter. The routine will check that the length of any given knotvector will be equal to the corresponding rowlength in 'coefs' plus the order. If that is not the case, an exception will be thrown.

Parameters:

coefs the coefficient grid

knotvectors pointer to an M-sized array of STL-vectors definining the M knotvectors.

order pointer to an M-sized array of integers representing the spline orders

5.11.2.6 template<int M, typename T> Go::GoTensorProductSpline< M, T >::GoTensorProductSpline (const GoTensorProductSpline< M, T > & *rhs*)

Copy constructor (weak - the new **GoTensorProductSpline**(p. 51) will share the same coefficient storage area as 'rhs').

Parameters:

rhs initialization spline

5.11.3 Member Function Documentation

5.11.3.1 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::clone (GoTensorProductSpline< M, T > & *rhs*) const

Clones itself into the argument spline, who becomes a hard copy of 'this' spline.

Note: It is the *user's* responsibility to make sure that the 'rhs' spline object's coefficient storage memory area is big enough to accomodate all the data that will be written into it.

Parameters:

rhs target spline object for the cloning

5.11.3.2 template<int M, typename T> GoBorrowedMVGrid<M, T> Go::GoTensorProductSpline< M, T >::coefficientGrid () const

Returns the coefficient grid.

NB: The returned grid will share its coefficients with the spline.

Returns:

the spline's grid of coefficients (control points)

5.11.3.3 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::cyclicPermute (int *steps*) `[inline]`

Cyclically permute the parameters of the spline.

A cyclical permutation with k "steps" will put parameter number i at position (i-k)%M

Example: For a trivariate spline with parameters (r, s, t):

- cyclicPermute(0) - new index order will be (r, s, t) (unchanged)
- cyclicPermute(1) - new index order will be (s, t, r)
- cyclicPermute(2) - new index order will be (t, r, s)
- cyclicPermute(3) - new index order will be (r, s, t) (unchanged)
- etc...

  Parameters:

  steps number of steps to cyclic permute

Definition at line 554 of file GoTensorProductSpline.h.

5.11.3.4 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::evalSurf (int *no_u*, int *no_v*, int *par_dir_1*, int *par_dir_2*, double *start_param_1*, double *start_param_2*, double *end_param_1*, double *end_param_2*, bool *reverse_dir_1*, bool *reverse_dir_2*, const double *const *fixed*, T * *res*) const

Fast evaluation of a whole, bivariate surface on the spline.

Only valid for spline objects where M >= 2. The 'surface' is defined as keeping all parameters fixed exept for two, which moves through the range specified by the arguments 'start_param_-1', 'start_param_2', 'end_param_1' and 'end_param_2' (must be within the valid range for the respective parameters' knot vectors). The number of samples in each of these two ranges are defined in 'no_u' and 'no_v', which can be seen as the "resolution" of the surface. The arguments 'par_dir_1' and 'par_dir_2' specifies which two of the spline's M parameters that should move (the others being fixed). The arguments 'reverse_dir_1' and 'reverse_dir_2' specifies whether the moving parameters should gradually increase from their min to their max value, or if they should decrease from their max to their min value. The array pointed to by 'fixed' gives the values of the fixed parameters (in increasing order), and the pointer 'res' points to a memory location in which the 'no_u'∗'no_v' computed result values should be written.

Parameters:

no_ u number of samples along the first moving parameter

no_ v number of samples along the second moving parameter

par_ dir_ 1 specify the first moving parameter (value from 0 to M-1)

par_ dir_ 2 specify the second moving parameter (value from 0 to M-1)

start_ param_ 1 start value for the sampled range of the first moving parameter

start_ param_ 2 start value for the sampled range of the second moving parameter

end_ param_ 1 end value for the sampled range of the first moving parameter

end_ param_ 2 end value for the sampled range of the second moving parameter

reverse_ dir_ 1 set to true if sampling for the first moving parameter should be done starting at the range's end value and ending at the start value, rather than the other way around

reverse_ dir_ 2 set to true if sampling for the second moving parameter should be done starting at the range's end value and ending at the start value, rather than the other way around

fixed pointer to an array of `double` s specifying the values of those parameters that remains fixed (in increasing order)

res pointer to the memory location where the result should be written

5.11.3.5 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::evalVolume (int ∗ *resolutions*, double ∗ *range_ start*, double ∗ *range_ end*, T ∗ *res*) const

Fast evaluation of the whole spline object for many parameter values.

The number of points to be calculated for each parameter is given in 'resolutions'.

Parameters:

resolutions should point to an M-sized array of integers, dictating how many (equidistant) values should be calculated for each parameter

range_ start should point to an M-sized array of `double` s, dictating the start of the range to be evaluated for each parameter

range_ end should point to an M-sized array of `double` s, dictating the end of the range to be evaluated for each parameter

res pointer to the memory area where the result should be written. If we have M parameters and want to evaluate for K_m parameter values along parameter m, then the total number of results will be $\Pi_{m=0}^{M-1} K_m$.

5.11.3.6 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::evalVolume (std::pair< double ∗, double ∗ > ∗ *sample_ points*, T ∗ *res*) const

Fast evaluation of the whole spline object for many parameter values.

The parameter values for the datapoints to be evaluated are given by the 'sample_points' argument, and the result is written to the memory area starting at the position indicated by the 'res' pointer.

Parameters:

sample_points points to a M-sized range of `pair`< ∗`double`, ∗`double`> (one entry for each parameter). The first of the `double` pointers in such a `pair` points to the beginning of a range of parametric values, and the second points to *one past* the end of the range. (This means that the number of samples to be effectuated for a given parameter 'm' is equal to (sample_points[m].second - sample_points[m].first).

res pointer to the memory area where the result should be written. If we have M parameters and want to evaluate for K_m parameter values along parameter m, then the total number of results will be $\Pi_{m=0}^{M-1} K_m$.

5.11.3.7 template<int M, typename T> GoTensorProductSpline<M, T> Go::GoTensorProductSpline< M, T >::extract (T ∗ *storage_ area*, int ∗ *knot_ start_ indexes*, int ∗ *knot_ end_ indexes*) const

Creates a new **GoTensorProductSpline**(p. 51) whose parameter domain is a subset of the original spline.

The parameter intervals that should be kept in this spline are delimited by the intervals starting with the knots specified in 'knot_start_indexes' and 'knot_end_indexes' (should be M-length integer arrays).

Parameters:

storage_ area a pointer to the coefficient storage area that the resulting spline should use. **NB**: The user should make sure enough memory is allocated!

knot_ start_ indexes pointer to an array of M integers, giving the indexes of the knotvector knots that defines the start of the selected range for each parameter

knot_ end_ indexes pointer to an array of M integers, giving the indexes of the knotvector knots that defines the end of the selected range for each parameter

Returns:

a spline that represent the original spline in a subset of the original parameter range

5.11.3.8 template<int M, typename T> int Go::GoTensorProductSpline< M, T >::findRange (int *par_ dir*, double *value*) const

For a given parameter, identifies the interval on the corresponding knotvector where the specified parameter value is located.

Parameters:

par_ dir the concerned parameter

value the value we want to determine the knot interval for

Returns:

the index of the start knot of the interval where 'value' is located on the specified knot vector.

5.11.3.9 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::fit (const GoGenericGrid< M, T > & *rawdata*, Array< GoApproximator< M, T > *, M > & *methods*, GoGenericGrid< M, T > & *errors*, bool *calculate_errors* = true)

Fit the spline to a grid of sample values, using a specified approximation method.

The samples to fit the spline to are given in the grid 'rawdata', and the approximation method to use for each parameter is given in the 'methods' argument. There are several different approximation methods (see **GoApproximator**(p. 11)), and the user can also write her own. If 'calculate_errors' is set to 'true', then the approximation error at each sample value is filled into the 'errors' grid, which must have exactly the same shape as 'rawdata'.

NB: It is the users responsibility to make sure that enough memory is allocated for the spline coefficients. The required memory depends on the chosen approximation method. By multiplying together the results from calling **GoApproximator::approximationSize()**(p. 13) on each of the applied approximator objects used, the number of coefficients (and thus the necessary memory requirement) is obtained. This only works for approximators that can calculate the number of coefficients they will generate *before* execution of their approximation algorithm. For other approximators, the approximation size will have to be estimated in another way. Again, refer to the documentation for **GoApproximator::approximationSize()**(p. 13) for a solution to this problem.

NB: It is also the users responsibility to make sure that enough memory is allocated for the 'errors' grid.

Parameters:

- ***rawdata*** the grid containing the sample values that we want to approximate with this spline.
- ***methods*** an M-sized arrays of pointers to **GoApproximator**(p. 11). The **GoApproximator**(p. 11) class is an abstract base class from which several approximation objects inherits. Each method will be applied on the corresponding parameter.
- ***errors*** a grid to which the approximation errors for each of the original datapoints are written (if 'calculate_errors" is set to true - if not, this parameter remains unused).
- ***calculate_errors*** if this boolean variable is set to true, the approximation error will be calculated for each of the datapoints. The error is written into the 'errors' grid.

5.11.3.10 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::fullEvaluate (GoGenericGrid< M, T > & *res*) const

Fast evaluation of the whole spline object, at equidistant parameter values.

The number of datapoints calculated for each parameter corresponds to the resolution of the grid 'res'.

Parameters:

- ***res*** The grid to which we want to write the result of the evaluation. At the same time, the resolution (rowlengths) of the grid dictates how many samples we want to calculate for each parameter.

5.11.3.11 template<int M, typename T> const BsplineBasis& Go::GoTensorProductSpline< M, T >::getBasis (int *i*) const `[inline]`

Returns the basis for a specified parameter.

Parameters:

i the parameter

Returns:

the basis for the specified parameter

Definition at line 510 of file GoTensorProductSpline.h.

5.11.3.12 template<int M, typename T> std::vector<double> Go::GoTensorProductSpline< M, T >::getKnotVector (int *tensor_ dir*) const

Returns the knotvector corresponding to the specified parameter.

Returns:

the knotvector for the given parameter

5.11.3.13 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::insertKnots (int *dir*, const std::vector< T > & *new_ knots*)

Knot insertion in a given parameter.

Insert knots at the knotvector for the parameter specified by 'dir'. The values of the knots to be inserted are given in the 'new_knots' vector.

Note: When knots are inserted, the number of control points also increases. It is the user's responsibility to assure that the spline has enough memory allocated to accommodate these new coefficients.

NB: This routine only works for T = `double`, since it makes use of the SISL library (function 1018).

Parameters:

dir the parameter whose knotvector the knots should be inserted to

new_ knots vector containing the knots to be inserted

5.11.3.14 template<int M, typename T> GoTensorProductSpline<M-1, T> Go::GoTensorProductSpline< M, T >::integrate (int *dir*, double * *domain_ start*, double * *domain_ end*, T * *storage_ area*) const

Integrates the spline along a specified parameter, producing a spline with one parameter less.

Returns a **GoTensorProductSpline**(p. 51) whose number of parameters N is one lower than for the original spline. It is produced by integrating the original spline along the parameter given as the first argument. The 'domain_start' and 'domain_end' parameters specify which part of the spline should be integrated. They should both point to an array in memory of M values, which specify the limiting parameter values for the part to integrate. The 'domain_start[dir]' and 'domain_end[dir]' values specify the range of integration (which is to take place for this parameter), while the other 'domain_start' and 'domain_end' values give which domain of the spline should be integrated.

Parameters:

dir specify which parameter to integrate

domain_ start points to an array of M `double` s specifying the start values for each parameter

domain_ end points to an array of M `double` s specifying the end values for each parameter

storage_ area the returned spline will store its coefficients in the memory location pointed to by this argument. **NB**: The user should make sure enough memory is allocated!

Returns:

a spline with one parameter less than the original spline, representing the integrated spline. It's start and end parameter values are as specified by domain_start and domain_end (except for domain_start[dir] and domain_end[dir] which expresses the range of integration)

5.11.3.15 template<int M, typename T> bool Go::GoTensorProductSpline< M, T >::operator== (const GoTensorProductSpline< M, T > & *rhs*) const `[inline]`

Equality operator.

Returns true only if all the basises of the two splines are equal, and that they share the same coefficient memory area.

Definition at line 523 of file GoTensorProductSpline.h.

References Go::GoTensorProductSpline< M, T >::basis_, and Go::GoTensorProductSpline< M, T >::data_.

5.11.3.16 template<int M, typename T> int Go::GoTensorProductSpline< M, T >::order (int *i*) const `[inline]`

Returns the order of the spline for a given parameter.

Parameters:

i the parameter

Returns:

the order of the spline for the specified parameter

Definition at line 498 of file GoTensorProductSpline.h.

5.11.3.17 template<int M, typename T> Array<T, 2 ∗ M> Go::GoTensorProductSpline< M, T >::paramspan () const

Returns an array containing the start and end parameter values.

The spline's parameter i has it start value at the array position $(2i)$, and its end parameter value is found at position $(2i + 1)$.

Returns:

an array of start and end parameter values

5.11.3.18 template<int M, typename T> T Go::GoTensorProductSpline< M, T >::point (const double *const *params*) const

Evaluate the spline for a certain choice of parameter values.

The parameter values are given in the argument array.

Note: If you want to evaluate a whole surface at a time, it will be much faster to use the **evalSurf()**(p. 56) function.

Parameters:

params pointer to an M-sized array containing the parameter values that should be used in the evaluation

Returns:

the value of the spline function for the given parameters

5.11.3.19 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::raiseOrderTo (int *new_order*, int *dir*)

Raise the order of the basis for a specified parameter.

The basis is raised to 'new_order'. It is required that 'new_order' is superior or equal to the current order.

Note: When the order is raised, new knots and control points are inserted. It is the user's responsibility to assure that the spline has enough memory allocated in its coefficient storage area to accomodate the new coefficients. Upper estimate of number of coefficients in a given parameter is:

(rowlength(dir) - order(dir)) * (new_order - order(dir) + 1) + new_order

The actual resulting number of coefficients may be less if the spline contains internal multiple knots.

NB: This function only works for T = `double`, since it makes use of the SISL library (function s1750).

Parameters:

new_order the new order for the given parameter. Must be superior or equal to the current order for that parameter.

dir the parameter for which the order should be raised

5.11.3.20 template<int M, typename T> GoTensorProductSpline<M-1, T> Go::GoTensorProductSpline< M, T >::reduce (int *dir*, T *param_val*, T * *data_area*) const

Making a spline with one less parameter from the original, obtained by fixing this parameter's value.

Returns a **GoTensorProductSpline**(p. 51) whose number or parameters N is one lower than the original spline. One parameter is removed by fixing its parameter value to 'param_val'. The parameter to remove is specified by 'dir'

Parameters:

dir the parameter to remove in the result spline

param_val the value that this parameter should take when generating the result spline

data_area a pointer to the coefficient storage area that the resulting spline should use. **NB**: The user should make sure enough memory is allocated!

Returns:

a spline with one parameter less than the original spline, obtained by fixing the removed parameter to a given value.

5.11.3.21 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::rescaleKnotVector (int *tensor_dir*, T *start_value*, T *end_value*)

Rescales the knotvector for the given parameter.

The parameter's new start and end values will match those given as arguments. The ratio between the different knot intervals will be preserved.

Parameters:

tensor_dir the parameter that we want to rescale the knotvector for

start_value new start value for the parameter range

end_value new end value for the parameter range

5.11.3.22 template<int M, typename T> int Go::GoTensorProductSpline< M, T >::rowlength (int *i*) const `[inline]`

Return the number of coefficients for the specified parameter.

Parameters:

i the parameter

Returns:

the number of coefficients for the specified parameter

Definition at line 488 of file GoTensorProductSpline.h.

5.11.3.23 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::setDataPointer (T * *p*)

Sets the spline's coefficient storage area to start at the memory address given by 'p'.

It is the user's responsibility to ensure that there is enough allocated memory to contain the size of the spline's coefficient grid. When setting the datapointer, no copying of coefficients from the previous location will take place.

Parameters:

p the start of the memory area where the spline should look up and store its coefficients.

5.11.3.24 template<int M, typename T> void Go::GoTensorProductSpline< M, T >::setUniformKnotVector (int *tensor_dir*, T *start*, T *end*, int *order*, bool *is_kreg*)

Redefines the basis for the specified parameter using a uniform knotvector and a given order.

Parameters:

tensor_dir the parameter that we want to redefine the basis for

start the start value of the parameter range

end the end value of the parameter range

order the requested order for the basis to be constructed

is_kreg defines whether the knotvector should be k-regular or not

5.11.3.25 template<int M, typename T> int Go::GoTensorProductSpline< M, T >::size () const `[inline]`

Return the number of spline coefficients.

Returns:

the number of spline coefficients

Definition at line 478 of file GoTensorProductSpline.h.

The documentation for this class was generated from the following file:

- GoTensorProductSpline.h

Chapter 6

Multivariate Splines Page Documentation

6.1 Practical example

Kari is a climate researcher that has just run a certain simulation on the earth's atmosphere. Her evaluation grid contains values at each degree confluence point on the planet, and for 18 pressure levels ("quasi-vertical coordinate"). Her simulation grid therefore contains 360 x 180 x 18 = 1166400 nodes.

In each node, a total of 128 physical quantities are calculated for each step of the simulation. The simulation covers a time span of 15 years, with four evaluations each year (one for each season). The total number of values in the grid is therefore:

$$128\ values/node * 1166400\ nodes/timestep\ * 60\ timesteps = 9\ billion\ values.$$

Each calculated value could therefore be referred to with the following indices:

- Physical quantity p
- Latitude lat
- Longitude lon
- Pressure level lev
- Timestep t

The simulation software writes the result to memory as a long consecutive list of values such that longitude has the shortest stride, then comes latitude, then comes time, then the concerned physical quantity, and finally the timestep which has the longest stride of all.

Kari is not satisfied with the way the result data is ordered in memory. Sorted by stride, the indices now are:

$$lat, lon, lev, p, t$$

This means that all grid information for a certain timestep t_i is consecutively stored before the grid information for the succeeding timestep t_{i+1}. This way, the data is globally sorted by timestep. Kari would rather collect all information pertaining to a given physical quantity together, which translates into the index ordering:

$$lat, lon, lev, t, p$$

To do this, she first establishes a grid that "borrows" the memory where the simulation data is located:

```
double* data_pointer = ...  // (Insert function here to set the pointer to the start
                            // of the result memory area)
int[5] grid_dimensions;
grid_dimensions[0] = 360; // number of longitude steps
grid_dimensions[1] = 180; // number of latitude steps
grid_dimensions[2] = 18;  // number of pressure levels
grid_dimensions[3] = 128; // number of physical variables
grid_dimensions[4] = 60;  // number of timesteps

Go::GoBorrowedMVGrid<5, double> result_grid(data_pointer, grid_dimensions);
```

`result_grid` could be considered a convenient wrapper for the data already stored in memory. Kari now defines a permutation that will change the indexing order such that the longest stride will be on the *physical quantity* index rather than the *time* index.

```
int[] permutation[] = {0, 1, 2, 4, 3}; // permutation swapping the two last indices
result_grid.permuteElements(permutation);
```

Note:

The permuting operation will need twice as much memory as the logical size of the grid. This can be prohibitive for very large grids. In our case the grid contains 9 billion values of type `double`, and would therefore require 72 billion bytes of storage. Depending on Kari's hardware, she may or may not be able to carry out this operation. In case of problems, she could use the slower, less memory-hungry equivalent member function **Go::GoGeneric-Grid::permuteElements_memOpt()**(p. 25).

From now on, Kari only wants to work on the physical variable called "Temperature". It is the 30th of the 128 variables, and since the physical valuables now has the longest stride, she can define a sub-grid that only covers "Temperature":

```
Go::GoBorrowedMVGrid<4, double> temperature_grid; // define an uninitialized grid
temperature_grid = result_grid.subgrid(30);   // access the subgrid for the 30th variable
```

Note:

No copying of results has taken place. The newly created `temperature_grid` just refer to a smaller part of the same data that is referred to by `result_grid`.

The temperature is stored as degrees Kelvin. For various reasons, Kari would like to have the temperature in degrees Celsius, which means that she should be subtracting 273.15 from each value in the grid. This is done in one code line:

```
const double kelvin_shift = 273.15;
temperature_grid -= kelvin_shift;
```

Finally, Kari decides to save the temperature data to disk:

```
#include <ifstream>
ofstream os("temperature.grid");
temperature_grid.write_BINARY(os);
os.close();
```

For presentation purposes, Kari wants to generate an 2048 x 1024 pixel image of the temperature at surface level worldwide. To accomplish this, she already has a function that converts a memory area with floating-point (`double`) values to a grey-scale JPEG image. But if she applies this function on any subset of the data generated in the above mentioned simulation, she can only generate pictures with the resolution 360 x 180. She wants to linearly interpolate between grid points whenever needed, and this is a situation where a linear, bivariate spline can come in handy. Let us see how she does this:

```
Using the simulation grid-points as control points in a linear spline
int basis_order[] = {2, 2, 2, 2}; // linear for longitude, latitude, level and time
Go::GoTensorProductSpline<4, double> spline(temperature_grid, basis_order, true);
// all four spline parameters will run from 0 (begin) to 1 (end)

// Sampling the spline at the requested points
const int res_x = 2048;
const int res_y = 1024;
double evaluated_storage[res_x * res_y];
double level_and_time_parameter_values[2];
level_and_time_parameter_values[0] = 0; // we want to evaluate the spline at sea level
level_and_time_parameter_values[1] = 0; // we want to evaluate the spline at the start of
                                           the simulation period
spline.evalSurf(res_x, // number of pixels in x direction
                res_y, // number of pixels in y direction
                0,     // x direction evaluated along first parameter (longitude)
                1,     // y direction evaluated along second parameter (latitude)
                0,     // x evaluation starts at parameter value 0
                0,     // y evaluation starts at parameter value 0
                1,     // x evaluation ends at parameter value 1
                1,     // y evaluation ends at parameter value 1
                false, // calculate x values using INCREASING parameter values
                false, // calculate y values using INCREASING parameter values
                level_and_time_parameter_values, // values of the fixed parameters
                evaluated_storage); // where to write the result
```

Now, the memory allocated for `evaluated_storage` contains the evaluation of the spline at sea level, at the start of the simulation, and using the requested resolution. A pointer to this memory area can now be given to Kari's function generating JPEG images.

Kari wants to send the temperature field to a colleague by email. However, the file that she saved on disk earlier in this example is very big, it contains about 70 million `double` values, which makes the file size something around 560 megabytes. This is too big to send over mail. However, she is willing to compromise the accuracy of the data, since she knows that her colleague will only use it to generate visual presentations and not for further simulation activities. She therefore chooses not to send him the real 4D temperature grid, but rather a spline approximation of it. The approximation she generates in the following way:

```
Go::Array<GoApproximator<4, double>*, 4> methods;
// ** compression method for LONGITUDE **
methods[0] = new Go::GoLeastSquareApproximator(60,    // control points
                                               4,     // order
                                               0.05); // rigidity parameter
// ** compression method for LATITUDE **
methods[1] = new Go::GoLeastSquareApproximator(30,    // control points
                                               4,     // order
                                               0.05); // rigidity parameter
// ** compression method for LEVEL **
methods[2] = new Go::GoLeastSquareApproximator(12,    // control points
                                               4,     // order
                                               0.05); // rigidity parameter
// ** compression method for TIME **
methods[3] = new Go::GoKnotremovalApproximator(3,   // order
                                                      3.0, // tolerance of +/- three degrees
                                                      0.1, // afctol
                                                      0);  // no preservation of border deriv.

    int necessary_data_size = methods[0].approximationSize() *
                              methods[1].approximationSize() *
                              methods[2].approximationSize();
    // the last approximator (Go::GoKnotremovalApproximator) is not able to predict
    // how many control points it will need before runtime (approximationSize() will return
    // a negative number.  In order to make sure we have enough storage memory, we multiply
    // by the following quantity:
    int max_possible_time_control_points =
       (-1) * methods[3].size() * temperature_grid.rowlength(3);

    necessary_data_size *= max_possible_time_control_points;

    std::vector<double> storage(necessary_data_size);
    Go::GoTensorProductSpline spl(&storage[0]); // specify where the spline should store its
                                                // coefficients, but no specification of the
                                               // spline itself

    // define a grid to store the error
    std::vector<double> error_storage(temperature_grid.size()); // reserve the same amount of
                                                                   memory to store error as
                                                                   is used to store the
                                                                   original temperature grid.
    Go::GoBorrowedMVGrid<4, double> error(&error_storage[0]); // specify where to store the
                                                              // grid's coefficients, but no
                                                              // specification on the grid
                                                              // itself.
    spl.fit(temperature_grid, methods, errors, true);

    ofstream os("temperature.spline");
    spl.write_BINARY(os);
```

Note:

Notice how we deal with the fact that we cannot estimate the necessary size of the control point storage space before actually running the **Go::GoTensorProductSpline::fit()**(p. 59) function. The problem is that the **Go::GoKnotremovalApproximator**(p. 32) cannot calculate this value a priori. Instead, by multiplying the absolute value of the return value of its approximationSize() member function by the number of timesteps in the simulation grid, we get an upper estimate on how many control points that can theoretically result from this process. (In practice, the real number is usually much lower).

The spline is fitted to the data, and later saved in the file "temperature.spline" The error is computed and stored in the `error` grid (we could have prevented this computation by setting the last argument to the fit() function to 'false' instead of 'true').

Kari later found out that the number of spline control points along the time direction (after applying the knot-removal approximator) was 40. Therefore the total storage space needed for storing the spline's coefficients are: 60 x 30 x 12 x 40 = 864000 values, which requires (multiply by sizeof()) about 7 megabytes of storage space - a much more reasonable quantity to send by mail. If Kari wanted to find out what the maximum approximation error was, she could have inspected the resulting `error` grid in this way:

```
    double max_error = max(fabs(error.minValue()), fabs(error.maxValue()));
```

Index