Structured Blocks Reference Manual

Generated by Doxygen 1.3.6

Fri Mar 18 16:50:53 2005

Contents

Chapter 1

Structured Blocks

1.1 What is this

This library works with multidimensional data-blocks, either standalone or connected in a cluster (or **block-structure**(p. 66), which is the word we use in this documentation). In short, a data-block represents a region of space with a certain shape, and which can have one or more scalar fields attached to it. Several **blocks**(p. 66) connected in a **structure**(p. 66) can in this way represent a subdivision of space, with scalar fields stretching out continuously across **blocks**(p. 66).

One obvious application of **blocks**(p. 66) and **block-structures**(p. 66) is representation of block-structured *grids*, frequently encountered in numerical simulations. Such grids (usually 2D or 3D) are defined to be *block-wise* regular grids, ie. they topologically consists of several regular grids pieced together along their **interfaces**(p. 67); the union of these grids constitutes a simulation mesh for the object of the simulation. Each of these regular grids can be expressed as a **block**(p. 66), and the whole simulation grid can be presented as a **blockstructure**(p. 66). The numerical results from the simulation on this grid can be represented as scalar fields on the **blocks**(p. 66). (To keep this idea in mind, we will from now on refer to the **blocks**(p. 66)' scalar fields as the **blocks**(p. 66)' **results**(p. 68)).

The benefit of expressing a simulation grid using this library's **blocks**(p. 66) is the possibility for *compression*. Highly refined simulation grids can often be extremely large and bulky, making them impractical to copy, move around or inspect from a distance on a band-limited network. Blocks, as provided by this library, can convert grids to continuous functions that can be compressed considerably, and presented with several **levels**(p. 67) of detail. This compression is lossy, but can be extremely effective (down to one percent or less of the original data), and the quality of the resulting representation is often good enough for many tasks, like visual inspection. This makes it possible to, for instance, distribute or inspect huge models on a band-limited network without a huge time overhead, and can even allow engineers to transport fairly big grids on an ordinary USB memory pen or other lightweight storage devices.

1.2 Concepts

1.2.1 Geometry, results, spatial and auxiliary parameters

More technically, a **block**(p. 66) represents a closed **S-manifold**(p. 67) in **geometric space**(p. 66) R^D (with S $\leq$ D). It has a shape (which we will refer to as its **geometry**(p. 66)) and optionally one or more scalar fields (referred to as **results**(p. 68)). Its **geometry**(p. 66) and **results**(p. 68)

are parameterized with the same set of parameters, from a parameter space which is the unit cube of R^M ($M \geq 1$).

The parameters do not all have the same status, and we will rather say that the parameter space is the unit cube of R^{S+C} where $S+C=M$. The reason for this precision is to clearly distinguish between what we call **spatial parameters**(p. 68) on one hand, and **auxiliary parameters**(p. 66) on the other. The **block**(p. 66), which is an **S-manifold**(p. 67), should be **homeomorphic**(p. 67) with the unit cube in R^S, and that is why the S first parameters are qualified as *spatial*. The remaining C parameters do not necessarily (though they might) modify the geometrical position of points in the **block**(p. 66).

Example 1:

An easy example to illustrate the difference between **spatial**(p. 68) and **auxiliary parameters**(p. 66) is to describe the earth's atmosphere as a **block**(p. 66) using our definitions. It could be parameterized on the **spatial parameters**(p. 68) **longitude**, **latitude** and **height**, while the **auxiliary parameter**(p. 66) **time** could be used to describe evolution. In this case, the **auxiliary parameter**(p. 66) is completely detached from any spatial context.

Example 2:

On the other hand, we could express a hot air balloon floating in the atmosphere as a **block**(p. 66). We had somehow defined a function which takes three **spatial parameters**(p. 68) and transform them into the shape of the balloon, and still have **time** defined as an **auxiliary parameter**(p. 66). In this case, the **auxiliary parameter**(p. 66) modifies position as well, since the balloon is drifting through the air, and maybe deforming itself in the process. However, the distinction between **spatial**(p. 68) and **auxiliary parameters**(p. 66) is still useful, since it is the **spatial parameters**(p. 68) that play the primary role in defining the shape of the object, and since it is the bijection from the **block**(p. 66)'s **geometry**(p. 66) to the unit cube in R^S that defines our **homeomorphism**(p. 67) necessary for talking about a **manifold**(p. 67).

Note:

It is this requirement that makes it meaningless to talk about a **block**(p. 66) where $S > D$.

1.2.2 Block connectivity

Block(p. 15) **interfaces**:

Before starting to talk about how **blocks**(p. 66) are connected together, we must define the **interfaces**(p. 67) along which they can connect.

As mentioned above, a **block**(p. 66) is an **S-manifold**(p. 67) in D-dimensional space, defined by a **homeomorphism**(p. 67) between the unit cube in R^S and the concerned region in the **geometric space**(p. 66). The unit cube is a bounded and closed region, and so will our **manifold**(p. 67) be. The boundaries of the unit cube will map to the boundaries of the **block**(p. 66). The boundary of a closed **S-manifold**(p. 67) is itself a **(S-1)-manifold**(p. 67). The boundary of the unit cube can be subdivided into $2S$ "faces", each defined by fixing one of the S parameters to one of its extremal values $\{0,1\}$. Each of these faces maps to a certain part of the **manifold's**(p. 67) boundary. This way, we define a subdivision of the **block**(p. 66)'s boundary into $2S$ separate regions, each of which is an **(S-1)-manifold**(p. 67). We call the **block**(p. 66)'s **(S-1)-interfaces**(p. 68).

We can go further. Each of the **interfaces**(p. 67) of the unit cube in R^S is itself a unit cube in R^{S-1}. It has a boundary consisting of $2(S-1)$ parts, obtained by fixing one of its S-1 parameters to one of its extremal values $\{0,1\}$. When mapping these boundaries back to our original **manifold**(p. 67), we obtain it's sm_interface "(S-2)-interfaces", which are of course the points on the boundaries between the **manifold's**(p. 67) **(S-1)-interfaces**(p. 68). We can continue recursively to define sm_interface "(S-m)-interfaces", where m is a value in $[1, 2, ..S]$. The smallest unit of boundary

is the **0-manifold**(p. 67), where all defining parameters are fixed to either 0 or 1, and which constitutes a single point on the **manifold's**(p. 67) boundary. It is easy to **show**(p. 64) that for a **S-block**(p. 68) (a **block**(p. 66) with S parameters, an **S-manifold**(p. 67)), its boundary will contain exactly $2^m\binom{S}{m}$ **(S-m)-interfaces**(p. 68).

Example:

We place ourselves in 3D-space, and use a **block**(p. 66) representing a **3-manifold**(p. 67) (S=3). Moreover, our **homeomorphism**(p. 67) is the identity operator, so that the **geometry**(p. 66) of our **block**(p. 66) is identical to the unit cube. Then our **block**(p. 66) has the following **interfaces**(p. 67):

2-**interfaces**(p. 67): $m = 1$, $2^1\binom{3}{1} = 6$ **interfaces**(p. 67) (the 6 faces of the cube)

1-**interfaces**(p. 67): $m = 2$, $2^2\binom{3}{2} = 12$ **interfaces**(p. 67) (the 12 edges of the cube)

0-**interfaces**(p. 67): $m = 3$, $2^3\binom{3}{3} = 8$ **interfaces**(p. 67) (the 8 **corner**(p. 66)s of the cube)

Interfaces(p. 67) **shared by several blocks**(p. 66)

Two or more **blocks**(p. 66) can be connected by sharing a given **(S-m)-interface**(p. 68). This means that they must be sharing *all* of the $2(S - m)$ **corner**(p. 66) corners of the **interface**(p. 67). It is not allowed for several **blocks**(p. 66) to share only *parts of* an **interface**(p. 67) with each other. Unless the dimension of the **geometric space**(p. 66) is higher than the dimension of the **manifold**(p. 67), it is not possible for more than *two* **blocks**(p. 66) to share a **(S-1)-interface**(p. 68) (for instance, if we are working with 3D-blocks in 3D-space, a maximum of two **blocks**(p. 66) can share the same, complete face). There are no such restrictions on **(S-m)-interfaces**(p. 68) where $m \geq 2$.

Visual example:

Here(p. 65) is an illustration of **block**(p. 66) connectivity in the 3D case.

1.2.3 This library

The Block(p. 15):

The library is highly configurable to cover as many needs as possible. The **blocks**(p. 66) can lie in a space of arbitrary dimension D, be a **manifold**(p. 67) of arbitrary number S (inferior or equal to D), have as many **auxiliary parameters**(p. 66) C as desired, and contain an arbitrary number of **results**(p. 68) (in addition to its **geometry**(p. 66)). (Of course, for expressing simulation grids, D and C are usually 2 or 3). The functions used to express **geometry**(p. 66) and **results**(p. 68) are **multivariate**(p. 67) *splines* , defined in the class **Go::GoTensorProductSpline** . The abstract, template base class for a **block**(p. 66) is called **Block**(p. 15). It doesn't implement anything, but specifies the minimum **interface**(p. 67) that all **blocks**(p. 66) should have in order to be used in a corresponding **BlockStructure**(p. 21). A **Block**(p. 15) is a template of S, C and D (as defined above). You can use one of the predefined block-types, or define your own, as long as you stick to the imposed **interface**(p. 67) (eventually with extensions of your own). A basic, instantiable **block**(p. 66) type provided by this library is **SingleLevelSplineBlock**(p. 56).

A more elaborate concept is that of a **MultilevelBlock**(p. 29), which contains several **levels**(p. 67) of detail. The user can set which **level**(p. 67) should be "active" through appropriate member functions, either by specifying it directly, or (for more elaborate **block**(p. 66) types) by specifying an error tolerance that should not be exceeded by the selected **level**(p. 67). The provided class **MultilevelSplineBlock**(p. 33) provides this functionality. The **active level**(p. 66) (and even the total number of **levels**(p. 67)) can be set independently for each **result**(p. 68) in the **block**(p. 66), as well as for the **block**(p. 66)'s **geometry**(p. 66).

The BlockStructure(p. 21)

The **BlockStructure**(p. 21) is a template of S, C, D and on *the kind of* **block**(p. 66) *used* . The BlockStructure's primary *raison d'etre* is to store a selection of **Block**(p. 15) s, as well as keeping track of how **blocks**(p. 66) are connected with each other. It has no notion of "compression" or **levels**(p. 67) (this is a concept that applies purely to the individual **block**(p. 66)), so you are free to use any kind of **block**(p. 66) you like, as long as it derives from **Block**(p. 15).

A problem that arises with compressed **blocks**(p. 66) is that their shape might change somewhat after compression, which may cause cracks and **block**(p. 66) intersections in the model. **Here**(p. 63) is an example. To overcome this problem, the **BlockStructure**(p. 21) can "force" continuity on **blocks**(p. 66) it knows to be neighbors. This is a very useful feature while working with compressed **blocks**(p. 66), especially since it handles any configuration of block connectivity as described in **Block connectivity**(p. 2) . Remember that since the **BlockStructure**(p. 21) does not have any notion of compression or **levels**(p. 67), the user has to explicitly specify when it is desired to enforce continuity.

The PlotableSubblock(p. 54)

When a user wants to evaluate a geometric position and/or a **result**(p. 68) value for a certain **block**(p. 66) and choice of parameter values, she has two options:

- She could ask the concerned **Block**(p. 15) *directly* to return the requested **Go::GoTensorProductSpline** s, and she could evaluate them directly for the parameter values concerned.
- If the **Block**(p. 15) is contained in a **BlockStructure**(p. 21), she could also call the latter's member function **BlockStructure::getPlotableRep()**(p. 23). This case does not require that the user has direct access to the **Block**(p. 15) itself, but there's another advantage too, namely that she can choose to enforce continuity if she wishes. The **result**(p. 68) is returned as a vector of "block fragments": **PlotableSubblock**(p. 54) . Each **PlotableSubblock**(p. 54) covers a part of the parametric domain that defines the **Block**(p. 15), and contains the necessary **Go::GoTensorProductSpline** s to calculate the concerned values for this part of the **block**(p. 66). If the user has asked for enforced continuity, some of the returned splines will differ from those really contained in the **Block**(p. 15), to assure that continuity is preserved.

  Note:

  Since it can be bothersome for the user to "piece together" a **block**(p. 66) from all its fragments returned upon a call to the **BlockStructure::getPlotableRep()**(p. 23) function, **BlockStructure**(p. 21) provides a pair of static utility functions that can do exactly this: **BlockStructure::sampleBlockGeometry()**(p. 26) and **BlockStructure::sampleBlockResult()**(p. 27).

1.3 Sample of use

UNFINISHED!

1.4 Glossary

Here(p. 66) you can find explanations for the most commonly used terms in this documentation.

Chapter 2

Structured Blocks Namespace Index

2.1 Structured Blocks Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 3

Structured Blocks Hierarchical Index

3.1 Structured Blocks Class Hierarchy

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

Chapter 4

Structured Blocks Class Index

4.1 Structured Blocks Class List

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

Chapter 5

Structured Blocks Page Index

5.1 Structured Blocks Related Pages

Here is a list of all related documentation pages:

Chapter 6

Structured Blocks Namespace Documentation

6.1 BlockEnum Namespace Reference

This namespace contains the necessary concepts and definitions for enumerating all **(S-m)-interfaces**(p. 68) of an S-block. (Read also **Block Connectivity**(p. 2)).

Classes

- class **ParamConfig**

  An array holding a total of S different **ParamState**(p. 14) *s.*

- class **ParamConfigEnumeration**

  Array of S^3 entries, each holding a **ParamConfig**(p. 49) *set upon construction of the array. It describes all the S^3* **(S-m)-interfaces**(p. 68) *that an S-block has.*

Enumerations

- enum **ParamState**

  Describes the state of a parameter.

6.1.1 Detailed Description

This namespace contains the necessary concepts and definitions for enumerating all **(S-m)-interfaces**(p. 68) of an S-block. (Read also **Block Connectivity**(p. 2)).

The objects in this namespace should not be very interesting to the programmer that only wants to benefit from this library's API. They are more closely related with the inner workings of the **BlockStructure**(p. 21).

6.1.2 Enumeration Type Documentation

6.1.2.1 enum BlockEnum::ParamState

Describes the state of a parameter.

If the state is **MIN**, then the parameter is considered "locked" to its minimum value (0). If the state is **MAX**, the parameter is considered "locked" to its maximum value (1). If the state is **RUN**, then the parameter is not locked, and can take any value in its domain. By associating one **ParamState**(p. 14) with each of the block's **spatial parameters**(p. 68), we can refer to a certain **interface**(p. 67) of the **block**(p. 66).

Example: For a 3D **block**(p. 66) with 3 **spatial parameters**(p. 68), we associate 3 **ParamState**(p. 14) s. If we set these three **ParamState**(p. 14) s to respectively {RUN, RUN, RUN} (no locked parameters), we are refering to the *interior* of the **block**(p. 66). If we set them to {MIN, RUN, RUN} (first parameter locked), we are refering to the "face" (2-**interface**(p. 67)) where the first parameter is locked to its *minimum* position. {MAX, RUN, RUN} would refer to the "face" with the first parameter locked to its *maximum* position, etc. It's easy to see that we can specify six different faces. Likewise, {MIN, MIN, RUN} would refer to an "edge" (1-**interface**(p. 67)), for which there are 12 different possibilities. {MIN, MIN, MIN} would refer to a **corner**(p. 66) (0-**interface**(p. 67)), and again we see that we have 8 different configurations, each corresponding to a different **corner**(p. 66) in the **block**(p. 66). Read the section on **Block Connectivity**(p. 2) for more.

Definition at line 53 of file BlockEnum.h.

Referenced by BlockEnum::ParamConfigEnumeration< S >::ParamConfigEnumeration().

Chapter 7

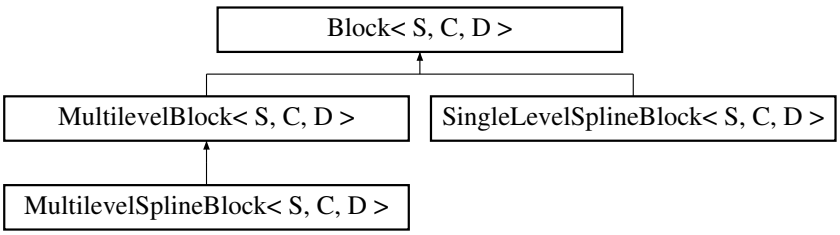
Structured Blocks Class Documentation

7.1 Block< S, C, D > Class Template Reference

This is the base class for all **blocks**(p. 66).

`#include <Block.h>`

Inheritance diagram for Block< S, C, D >::



Public Member Functions

- virtual void **write** (std::ostream &os, bool ascii=false) const=0

  Write the **block**(p. 66) *to an output stream.*

- virtual void **read** (std::istream &is, bool ascii=false)=0

  Read the **block**(p. 66) *from an input stream.*

- virtual Go::BoundingBox **boundingBox** () const=0

  Returns a bounding box enclosing the geometric area occupied by the **block**(p. 66).

- virtual void **getCornerPosition** (const bool *max, const double *C_values, double *res) const=0

  Function to retrieve the position of a block's **corner**(p. 66) *in geometrical space for a given set of* **auxiliary parameters**(p. 66) *(C-parameters).*

- virtual const Go::Array< **Go::GoTensorProductSpline**< S+C, double >, D > **geometrySplines** () const=0

  Returns the **multivariate**(p. 67) *splines that define this block's* **geometry**(p. 66).

- virtual const **Go::GoTensorProductSpline**< S+C, double > **resultSplines** (int resultID) const=0

  Returns the **multivariate**(p. 67) *spline describing one of the block's* **results**(p. 68).

- virtual std::vector< boost::shared_ptr< Go::GeomObject > > **outline** (double *aux_par_values) const=0

  Returns the **outline**(p. 67) *of the* **block**(p. 66).

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

  Returns the number of **results**(p. 68) *that this* **block**(p. 66) *contains.*

- virtual int **getGeometryLevel** () const

  Return the index number of the currently **active geometry level**(p. 66) *(the one whose splines are returned upon a call to* **geometrySplines()**(p. 17) *).*

- virtual int **getResultLevel** (int res) const

  Return the index number of the currently **active result level**(p. 66) *for the requested* **result**(p. 68) *(the one whose spline is returned upon a call to* **resultSplines()**(p. 19) *).*

7.1.1 Detailed Description

template<int S, int C, int D> class Block< S, C, D >

This is the base class for all **blocks**(p. 66).

Classes deriving from this class can be used in combination with the **BlockStructure**(p. 21) template class. The template parameters S, C and D respectively represent the **manifold**(p. 67) dimension of the **block**(p. 66) (number of **spatial parameters**(p. 68) that defines its **geometry**(p. 66)), the number of **auxiliary parameters**(p. 66), and the dimension of the **geometric space**(p. 66) where the **block**(p. 66) is situated. For a more detailed explanation, refer to the **main page**(p. 1). The **block**(p. 66) is defined by its **geometry**(p. 66), as well an optional number of **results**(p. 68), which are scalar fields defined on the volume occupied by the **block**(p. 66). Both the block's **geometry**(p. 66) and **result**(p. 68) *fields* are defined using M-variate splines, where M is equal to S+C.

Definition at line 55 of file Block.h.

7.1.2 Member Function Documentation

7.1.2.1 template<int S, int C, int D> virtual Go::BoundingBox Block< S, C, D >::boundingBox () const [pure virtual]

Returns a bounding box enclosing the geometric area occupied by the **block**(p. 66).

Returns:
the a bounding box enclosing the geometric area occupied by the **block**(p. 66)

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 36), and **SingleLevelSplineBlock< S, C, D >** (p. 58).

7.1.2.2 template<int S, int C, int D> virtual const Go::Array<Go::GoTensorProductSpline<S+C, double>, D> Block< S, C, D >::geometrySplines () const [pure virtual]

Returns the **multivariate**(p. 67) splines that define this block's **geometry**(p. 66).

There are D such splines, each expressing the position of the **block**(p. 66) in one spatial dimension, as a function of the S+C parameters. (For instance, in 3D, we would have one spline expressing the x-coordinate, another expressing the y-coordinate and a third one expressing the z-coordinate).

Returns:

A D-sized array containing the **multivariate**(p. 67) splines describing the **geometry**(p. 66) of the **block**(p. 66) in each of the spatial dimensions.

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 37), and **SingleLevelSplineBlock< S, C, D >** (p. 58).

7.1.2.3 template<int S, int C, int D> virtual void Block< S, C, D >::getCornerPosition (const bool * *max*, const double * *C_values*, double * *res*) const [pure virtual]

Function to retrieve the position of a block's **corner**(p. 66) in geometrical space for a given set of **auxiliary parameters**(p. 66) (C-parameters).

The block's **corners**(p. 66) are defined as its **0-interfaces**(p. 67) (see **Block connectivity**(p. 2)). A **corner**(p. 66) is characterized by all the **spatial parameters**(p. 68) taking up an extremal value (either 0 or 1). The *max* argument is used to specify which **spatial parameters**(p. 68) are 0 and which are 1. It should point to an array of S bools. If *max*[i] = `true`, it means that for the requested **corner**(p. 66), the **spatial parameter**(p. 68) i is set to 1, else it is set to 0. The spatial coordinates of the requested **corner**(p. 66) are written to the memory location pointed to by *res*. This area should of course be big enough to store D `double` s.

Parameters:

max Pointer to an array of S `bool` s, specifying whether the corresponding **spatial parameter**(p. 68) is *maximum* (1) or *minimum* (0) at this **corner**(p. 66). A value of `true` means *maximum*.

C_values Pointer to an array of C `double` s, specifying the values of the **auxiliary parameters**(p. 66) to use when evaluating the position of the **corner**(p. 66) corners.

res pointer to the memory location where the **corner**(p. 66) position (D `double` s) will be written. It is the user's responsibility to allocate enough memory.

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 38), and **SingleLevelSplineBlock< S, C, D >** (p. 58).

7.1.2.4 template<int S, int C, int D> virtual int Block< S, C, D >::getGeometryLevel () const [inline, virtual]

Return the index number of the currently **active geometry level**(p. 66) (the one whose splines are returned upon a call to **geometrySplines()**(p. 17)).

This number is always 0 for **single-leveled**(p. 67) **blocks**(p. 66). It only becomes interesting for **blocks**(p. 66) inheriting from **MultilevelBlock**(p. 29), reimplementing this function. The reason this function is present here, is that the **BlockStructure**(p. 21) object needs access to this function when adjusting two **interface**(p. 67)-sharing **blocks**(p. 66) in **order**(p. 68) to assure continuity. Even though the **BlockStructure**(p. 21) is supposed to be ignorant of the concept of **levels**(p. 67), it has been implemented to take advantage of this information. This happens because when **BlockStructure**(p. 21) is trying to enforce continuity, it needs to know which **blocks**(p. 66) in the structure provide the most "reliable" (most detailed) information.

Returns:
0 unless overridden by a **block**(p. 66) supporting multiple **levels**(p. 67)

Reimplemented in **MultilevelBlock**< **S, C, D** > (p. 30), and **MultilevelSplineBlock**< **S, C, D** > (p. 39).

Definition at line 165 of file Block.h.

7.1.2.5 template<int S, int C, int D> virtual int Block< S, C, D >::getResultLevel (int *res*) const [inline, virtual]

Return the index number of the currently **active result level**(p. 66) for the requested **result**(p. 68) (the one whose spline is returned upon a call to **resultSplines()**(p. 19)).

This number is always 0 for **single-leveled**(p. 67) **blocks**(p. 66). It only becomes interesting for **blocks**(p. 66) inheriting from **MultilevelBlock**(p. 29), reimplementing this function. The reason this function is present here, is that the **BlockStructure**(p. 21) object needs access to this function when adjusting two **interface**(p. 67)-sharing **blocks**(p. 66) in order to assure continuity. Even though the **BlockStructure**(p. 21) is supposed to be ignorant of the concept of **levels**(p. 67), it has been implemented to take advantage of this information. This happens because when **BlockStructure**(p. 21) is trying to enforce continuity, it needs to know which **blocks**(p. 66) in the structure provide the most "reliable" (most detailed) information.

Parameters:
res index of the requested **result**(p. 68). Must be inside the range [0, **numResults()**(p. 18) - 1]

Returns:
0 unless overridden by a **block**(p. 66) supporting multiple **levels**(p. 67)

Reimplemented in **MultilevelBlock**< **S, C, D** > (p. 30), and **MultilevelSplineBlock**< **S, C, D** > (p. 41).

Definition at line 184 of file Block.h.

7.1.2.6 template<int S, int C, int D> virtual int Block< S, C, D >::numResults () const [pure virtual]

Returns the number of **results**(p. 68) that this **block**(p. 66) contains.

The **results**(p. 68) will always be refered to by using their index number 0 to **numResults()**(p. 18) - 1

Returns:
the number of **results**(p. 68) that this **block**(p. 66) contains

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 44), and **SingleLevelSplineBlock< S, C, D >** (p. 59).

7.1.2.7 template<int S, int C, int D> virtual std::vector<boost::shared_ptr<Go::GeomObject> > Block< S, C, D >::outline (double * *aux_par_values*) const [pure virtual]

Returns the **outline**(p. 67) of the **block**(p. 66).

The **outline**(p. 67) is defined as the union of all the block's **1-interfaces**(p. 67) (edges), which is collected, expressed as Go-objects (usually Go::SplineCurve s) and returned as the **result**(p. 68) of the function call. This function can be used for all instantiations of the Block template, provided that S > 0. The (fixed) values for the **auxiliary parameters**(p. 66) are specified by the user in an C-sized array pointed to by the argument aux_par_values.

Parameters:

aux_par_values points to an array of C *double* s, defining the values of the **auxiliary parameters**(p. 66) for which we want the **outline**(p. 67) of the **block**(p. 66).

Returns:

a vector of Go-objects (usually Go::SplineCurves) expressing the **outline**(p. 67) of the **block**(p. 66) for the specified values for the **auxiliary parameters**(p. 66).

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 44), and **SingleLevelSplineBlock< S, C, D >** (p. 59).

7.1.2.8 template<int S, int C, int D> virtual void Block< S, C, D >::read (std::istream & *is*, bool *ascii* = false) [pure virtual]

Read the **block**(p. 66) from an input stream.

Parameters:

is the input stream where the **block**(p. 66) is read from

ascii if the user sets this argument to *true*, then the **block**(p. 66) will be read in ASCII format, else it will be read in BINARY format.

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 44), and **SingleLevelSplineBlock< S, C, D >** (p. 60).

7.1.2.9 template<int S, int C, int D> virtual const Go::GoTensorProductSpline<S+C, double> Block< S, C, D >::resultSplines (int *resultID*) const [pure virtual]

Returns the **multivariate**(p. 67) spline describing one of the block's **results**(p. 68).

Parameters:

resultID The index number of the requested **result**(p. 68). Valid values are from 0 to **numResults()**(p. 18) - 1.

Returns:

A **multivariate**(p. 67) spline describing the requested **result**(p. 68) scalar field

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 45), and **SingleLevelSplineBlock< S, C, D >** (p. 60).

7.1.2.10 template<int S, int C, int D> virtual void Block< S, C, D >::write (std::ostream & *os*, bool *ascii* = false) const [pure virtual]

Write the **block**(p. 66) to an output stream.

Parameters:

os the output stream where the **block**(p. 66) will be written

ascii if the user sets this argument to *true*, then the **block**(p. 66) will be written in ASCII format, else it will be written in BINARY format.

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 47), and **SingleLevelSplineBlock< S, C, D >** (p. 61).

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

- Block.h

7.2 BlockStructure< S, C, D, BlockType > Class Template Reference

This **blockstructure**(p. 66) class stores a group of **Block**(p. 15) s, and keeps track of how they are connected to each other (ie. which **blocks**(p. 66) shares what **interfaces**(p. 67)).

`#include <BlockStructure.h>`

Public Member Functions

- **BlockStructure** ()

  Default constructor, generating a **blockstructure**(p. 66) *with no blocks.*

- **BlockStructure** (std::vector< boost::shared_ptr< BlockType< S, C, D > > > &blocks, double cornerdist)

  This constructor is initialized with a vector of **blocks**(p. 66)*, and uses automatic detection to determine which* **blocks**(p. 66) *are connected along which* **interfaces**(p. 67)*.*

- void **write** (std::ostream &os, bool ascii=false) const

  write the complete **blockstructure**(p. 66) *to an output stream*

- void **read** (std::istream &is, bool ascii=false)

  Read the **blockstructure**(p. 66) *from an input stream.*

- std::vector< boost::shared_ptr< **PlotableSubblock**< S, D > > > **getPlotableRep** (const std::vector< int > &block_numbers, const std::vector< int > &result_numbers, AllocatedMemoryHolder &amh, bool adapt, std::vector< int > &block_start_indexes, const double *fixed_params=0) const

  Return a vector of **PlotableSubblock**(p. 54) *s collectively representing the* **blocks**(p. 66) *the user have requested.*

- int **numBlocks** () const

  return the number of **blocks**(p. 66) *in the* **blockstructure**(p. 66)

- std::string **connectivityInformation** () const

  return a string describing (in text format) how the **blocks**(p. 66) *contained in this* **blockstructure**(p. 66) *are connected with each other*

- bool **hasNeighbourAmong** (int block_ix, const **BlockEnum::ParamConfig**< S > &pc, const std::vector< int > &selected_blocks) const

  Reports on whether or not a given **(S-m)-interface**(p. 68) *in a specified* **block**(p. 66) *is shared by a neighbour (from a set of specified* **blocks**(p. 66)*).*

Static Public Member Functions

- template<typename FloatType> void **sampleBlockGeometry** (typename std::vector< boost::shared_ptr< **PlotableSubblock**< S, D > > >::const_iterator start_range, typename std::vector< boost::shared_ptr< **PlotableSubblock**< S, D > > >::const_iterator end_range, Go::Array< std::vector< double >, S > sample_values, double tolerance, FloatType *result_pointer)

This is a utility function for sampling a block's **geometry**(p. 66) *at a set of specified specified parameter values.*

- template<typename FloatType> void **sampleBlockResult** (int result_ix_local, typename std::vector< boost::shared_ptr< **PlotableSubblock**< S, D > > >::const_iterator start_range, typename std::vector< boost::shared_ptr< **PlotableSubblock**< S, D > > >::const_iterator end_range, Go::Array< std::vector< double >, S > sample_values, double tolerance, FloatType *result_pointer)

  This is a utility function for sampling one of the block's **results**(p. 68) *at a set of specified specified parameter values.*

7.2.1 Detailed Description

template<int S, int C, int D, template< int, int, int > class BlockType> class BlockStructure< S, C, D, BlockType >

This **blockstructure**(p. 66) class stores a group of **Block**(p. 15) s, and keeps track of how they are connected to each other (ie. which **blocks**(p. 66) shares what **interfaces**(p. 67)).

The class is highly templatized, on the dimension of the **manifold**(p. 67) (the block's number of **spatial parameters**(p. 68)), on the number of **auxiliary parameters**(p. 66), and on the dimension of the geometric space in which the **blocks**(p. 66) lie.

There is functionality for enforcing C0 continuity between **blocks**(p. 66) sharing **interfaces**(p. 67). This is useful for **blockstructures**(p. 66) containing compressed **blocks**(p. 66), where we cannot be guaranteed that the different **blocks**(p. 66)' **geometries**(p. 66) and **result**(p. 68) fields will perfectly match at the **interfaces**(p. 67) due to detail loss during compression.

Definition at line 143 of file BlockStructure.h.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 template<int S, int C, int D, template< int, int, int > class BlockType> BlockStructure< S, C, D, BlockType >::BlockStructure () [inline]

Default constructor, generating a **blockstructure**(p. 66) with no blocks.

It can later be **read()**(p. 25) into.

Definition at line 150 of file BlockStructure.h.

7.2.2.2 template<int S, int C, int D, template< int, int, int > class BlockType> BlockStructure< S, C, D, BlockType >::BlockStructure (std::vector< boost::shared_ptr< BlockType< S, C, D > > > & *blocks*, double *cornerdist*) [inline]

This constructor is initialized with a vector of **blocks**(p. 66), and uses automatic detection to determine which **blocks**(p. 66) are connected along which **interfaces**(p. 67).

Parameters:

block an STL vector with (**shared pointers**(p. 68) to) the **blocks**(p. 66) in question. The **blocks**(p. 66) are never copied, only the **shared pointers**(p. 68).

cornerdist **corners**(p. 66) (0-**interfaces**(p. 67)) from different **blocks**(p. 66) that are closer to each other (Euclidian distance) than this value, are considered "connected". If both **corners**(p. 66) of a block's 1-**interface**(p. 67) (edge) is connected with two corresponding **corners**(p. 66) in another block's 1-**interface**(p. 67), then the 1-**interfaces**(p. 67) are considered connected. Generally, if the $2^(S-m)$ **corners**(p. 66) of one of the block's **(S-m)-interfaces**(p. 68) are connected with those of another block's **(S-m)-interface**(p. 68), the **interfaces**(p. 67) are considered to be connected.

Note:

It is not allowed to let two or more **blocks**(p. 66) share only *parts of an* **interface**(p. 67) with each other (for instance, having two 3D-**blocks**(p. 66)'s sharing only 3 of the four **corners**(p. 66) of a face (1-**manifold**(p. 67)) with each other). The **blocks**(p. 66) can share 0-**interfaces**(p. 67), 1-**interfaces**(p. 67) and so on, but in each case, the *complete* **interface**(p. 67) must be shared. Read **the section on block connectivity**(p. 2) for more information about block connectivity.

The user should make sure that *cornerdist* is a much smaller number than the sides of the smallest block's bounding box (to avoid errors such as connecting two **corners**(p. 66) of the same **block**(p. 66)).

Definition at line 35 of file BlockStructure_templates.h.

7.2.3 Member Function Documentation

7.2.3.1 template<int S, int C, int D, template< int, int, int > class BlockType> std::string BlockStructure< S, C, D, BlockType >::connectivityInformation () const [inline]

return a string describing (in text format) how the **blocks**(p. 66) contained in this **blockstructure**(p. 66) are connected with each other

This function was used for debugging purposes, but is left here, as it can come in handy in conveying connectivity information to the user.

Definition at line 310 of file BlockStructure_templates.h.

7.2.3.2 template<int S, int C, int D, template< int, int, int > class BlockType> std::vector< boost::shared_ptr< PlotableSubblock< S, D > > > BlockStructure< S, C, D, BlockType >::getPlotableRep (const std::vector< int > & *block_numbers*, const std::vector< int > & *result_numbers*, AllocatedMemoryHolder & *amh*, bool *adapt*, std::vector< int > & *block_start_indexes*, const double * *fixed_params* = 0) const [inline]

Return a vector of **PlotableSubblock**(p. 54) s collectively representing the **blocks**(p. 66) the user have requested.

This function considers all **blocks**(p. 66) requested by the user in 'block_numbers', with all the **results**(p. 68) requested in 'result_numbers'. It generates a vector of (**shared pointers**(p. 68) to) **PlotableSubblock**(p. 54) s that can be used to evaluate the requested **blocks**(p. 66) and **results**(p. 68).

Note:

Technical: The reader might ask why not just return the **blocks**(p. 66) themselves, since they provide direct access to the GoTensorProductSpline s defining their **geometry**(p. 66)

and all their **results**(p. 68). Why go to the seemingly awkward step of (possibly) splitting up the **blocks**(p. 66) and return a vector of **PlotableSubblock**(p. 54)?
The answer to this question is that we want to be able to *assure continuity* between **blocks**(p. 66). block "Blocks" registered as "neighbours" by the BlockStructure, have no knowledge of this themselves, and will not be able to adjust their **interfaces**(p. 67) to each other in order to provide exact C0 continuity. The BlockStructure, on the other hand, can use this additional information to adjust their spline coefficients in the **interface**(p. 67) areas such that C0 continuity is assured.
In order to do this, its often necessary to insert additional knots into the splines' knotvectors. We don't want to modify the **blocks**(p. 66)' internal splines directly. Moreover, we want to keep knot insertion local, avoiding refining the whole **block**(p. 66). Therefore, we build up an entirely new spline model that assures C0 continuity, and return this model to the user. The **blocks**(p. 66)' original splines are used to the fullest possible extent, and in region where adjustment have to be made, new splines are defined to make the transitions.
As long as the user sticks to the splines and associated parameter domains provided by the returned **PlotableSubblock**(p. 54) s, she should not have to worry about any of this.
The BlockStructure provides two static functions to facilitate working with **PlotableSubblock**(p. 54) s rather than **Block**(p. 15) s. These are **sampleBlockGeometry()**(p. 26) and **sampleBlockResult()**(p. 27)

Parameters:

block_numbers vector containing the indexes of the **blocks**(p. 66) that we want to have a **plotable**(p. 68) representation for (valid numbers are 0 to **numBlocks()**(p. 25) - 1)

result_numbers vector containing the indexes of the **results**(p. 68) that we want to have a **plotable**(p. 68) representation for (valid numbers are 0 to the number of **results**(p. 68) present in the **blocks**(p. 66) - 1)

amh Since the data structures contained in the PlotableSubblocks s do not own their data, any additional memory allocated during execution of this function will be stored in this memory holder. The user doesn't need it for anything special, but she should not dispose of it (let it go out of scope) before doing the same with the generated **PlotableSubblock**(p. 54) s.

adapt if set to 'true', the BlockStructure will generate "transitional splines" at the **interfaces**(p. 67) of neighbouring **blocks**(p. 66) in order to assure C0 continuity between them. The user will not notice the presence of these transitional splines as long as she sticks to using the **PlotableSubblock**(p. 54) s the way they are supposed to. (Read documentation for **PlotableSubblock**(p. 54)).

block_start_indexes for this function, all **PlotableSubblock**(p. 54) s collectively representing a certain **block**(p. 66) are stored *consecutively* in the returned vector. To indicate where the representation of one **block**(p. 66) ends and another one begins, the vector 'block_start_indexes' will contain the start indices in the resulting vector for the representation of each requested **block**(p. 66).

Exemple: The user requested the **blocks**(p. 66) numbered 2, 4 and 5. For the returned vector **V**, **block**(p. 66) 2 will be represented by the **PlotableSubblock**(p. 54) s indexed from V[0] to V[*block_start_indices*[1]-1]. **Block**(p. 15) 4 will be represented from V[*block_start_indices*[1]] to V[*block_start_indices*[2]-1]. **Block**(p. 15) 5 will be represented from V[*block_start_indices*[2]] up to the last entry of the vector.

fixed_params pointer to an array of C `double` s, representing the values that the **auxiliary parameters**(p. 66) should take when constructing the **PlotableSubblock**(p. 54) s. (The splines contained in the **PlotableSubblock**(p. 54) s themselves only depend on the S **spatial parameters**(p. 68)).

Definition at line 136 of file BlockStructure_templates.h.

7.2.3.3 template<int S, int C, int D, template< int, int, int > class BlockType> bool BlockStructure< S, C, D, BlockType >::hasNeighbourAmong (int *block_ix*, const BlockEnum::ParamConfig< S > & *pc*, const std::vector< int > & *selected_blocks*) const [inline]

Reports on whether or not a given **(S-m)-interface**(p. 68) in a specified **block**(p. 66) is shared by a neighbour (from a set of specified **blocks**(p. 66)).

Parameters:

block_ix the index of the **block**(p. 66) we want to query

pc the specification of the **interface**(p. 67) we want to query (read **BlockEnum::ParamConfig**(p. 49) to learn how to specify an **interface**(p. 67) with this object)

selected_blocks indexes the candidate **blocks**(p. 66) for neighbourship

Returns:

'true' if the **block**(p. 66) with the index 'block_ix' has a neighbour among those indexed in 'selected_blocks' along the **interface**(p. 67) specified by 'pc'.

Definition at line 3179 of file BlockStructure_templates.h.

7.2.3.4 template<int S, int C, int D, template< int, int, int > class BlockType> int BlockStructure< S, C, D, BlockType >::numBlocks () const [inline]

return the number of **blocks**(p. 66) in the **blockstructure**(p. 66)

Returns:

the number of **blocks**(p. 66) in the structure

Definition at line 162 of file BlockStructure_templates.h.

7.2.3.5 template<int S, int C, int D, template< int, int, int > class BlockType> void BlockStructure< S, C, D, BlockType >::read (std::istream & *is*, bool *ascii* = false) [inline]

Read the **blockstructure**(p. 66) from an input stream.

Parameters:

is the input stream where the **blockstructure**(p. 66) is read from

ascii if the user sets this argument to *true*, then the block structure will be read in ASCII format, else it will be read in BINARY format.

Definition at line 104 of file BlockStructure_templates.h.

7.2.3.6 template<int S, int C, int D, template< int, int, int > class BlockType> template<typename FloatType> void BlockStructure< S, C, D, BlockType >::sampleBlockGeometry (typename std::vector< boost::shared_ptr< PlotableSubblock< S, D > > >::const_iterator *start_range*, typename std::vector< boost::shared_ptr< PlotableSubblock< S, D > > >::const_iterator *end_range*, Go::Array< std::vector< double >, S > *sample_values*, double *tolerance*, FloatType * *result_pointer*) [inline, static]

This is a utility function for sampling a block's **geometry**(p. 66) at a set of specified specified parameter values.

It takes as input the **PlotableSubblock**(p. 54) s returned from the **getPlotableRep()**(p. 23) function. The **PlotableSubblock**(p. 54) s are those contained in the range 'start_range' - 'end_range' Together they should represent and cover the parameter domain of *one* **block**(p. 66), with no parameter domain overlap among themselves. Using these 'fragments', the corresponding block's **geometry**(p. 66) will be sampled at the parameter values specified by the S vectors in 'sample_values', and the samples will be written consecutively to the memory area pointed to by 'result_pointer'. (Total number of D-dimensional samples is equal to the product of the length of the 'sample_values' vectors). The 'tolerance' argument defines how 'far outside' a parametric value can lie outside a **subblock**(p. 69)'s parametric domain and still be considered part of the domain.

NB: The vectors of 'sample_values' should contain monotonously increasing values! However for optimality reasons, this routine does not verify this condition.

Parameters:

start_range start of the range of (**shared pointers**(p. 68) to) the **PlotableSubblock**(p. 54) s that represent the **block**(p. 66).

end_range end of the range of (**shared pointers**(p. 68) to) the **PlotableSubblock**(p. 54) s that represent the **block**(p. 66).

sample_values Array containing S vectors, each associated with one of the **spatial parameters**(p. 68) of the **block**(p. 66), and specifying for which values of this parameter the block's geometric position should be sampled.

tolerance defines how 'far outside' a certain parametric value can lie outside a **subblock**(p. 69)'s parametric domain and still be considered part of the domain. Should be kept to a very small value.

result_pointer the result of the sample process will be written to the memory area pointed to by this pointer. The number of D-dimensional samples will be equal to the value obtained by multiplying the length of the S vectors in 'sample_values' together. Each sample is written with the dimension as the shortest **stride**(p. 68) (D different values), then the first parameter. The last parameter is sampled with the longest **stride**(p. 68).

Definition at line 23 of file SampleBlock_templates.h.

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

7.2.3.7 template<int S, int C, int D, template< int, int, int > class BlockType> template<typename FloatType> void BlockStructure< S, C, D, BlockType >::sampleBlockResult (int *result_ix_local*, typename std::vector< boost::shared_ptr< PlotableSubblock< S, D > > >::const_iterator *start_range*, typename std::vector< boost::shared_ptr< PlotableSubblock< S, D > > >::const_iterator *end_range*, Go::Array< std::vector< double >, S > *sample_values*, double *tolerance*, FloatType * *result_pointer*) [inline, static]

This is a utility function for sampling one of the block's **results**(p. 68) at a set of specified specified parameter values.

It takes as input the **PlotableSubblock**(p. 54) s returned from the **getPlotableRep()**(p. 23) function. The **PlotableSubblock**(p. 54) s are those contained in the range 'start_range' - 'end_range' Together they should represent and cover the parameter domain of *one* **block**(p. 66), with no parameter domain overlap among themselves. Using these 'fragments', the corresponding block's requested **result**(p. 68) will be sampled at the parameter values specified by the S vectors in 'sample_values', and the samples will be written consecutively to the memory area pointed to by 'result_pointer'. (Total number of samples is equal to the product of the length of the 'sample_values' vectors). The 'tolerance' argument defines how 'far outside' a parametric value can lie outside a **subblock**(p. 69)'s parametric domain and still be considered part of the domain.

NB: The vectors of 'sample_values' should contain monotonously increasing values! However for optimality reasons, this routine does not verify this condition.

Parameters:

result_ix_local the index of the **result**(p. 68) we want to evaluate

start_range start of the range of (**shared pointers**(p. 68) to) the **PlotableSubblock**(p. 54) s that represent the **block**(p. 66).

end_range end of the range of (**shared pointers**(p. 68) to) the **PlotableSubblock**(p. 54) s that represent the **block**(p. 66).

sample_values Array containing S vectors, each associated with one of the **spatial parameters**(p. 68) of the **block**(p. 66), and specifying for which values of this parameter the block's **result**(p. 68) should be sampled.

tolerance defines how 'far outside' a certain parametric value can lie outside a **subblock**(p. 69)'s parametric domain and still be considered part of the domain. Should be kept to a very small value.

result_pointer the result of the sample process will be written to the memory area pointed to by this pointer. The number of samples will be equal to the value obtained by multiplying the length of the S vectors in 'sample_values' together. The writing order is such that the first parameter is sampled with the lowest **stride**(p. 68); the last parameter with the longest.

Definition at line 109 of file SampleBlock_templates.h.

References Go::GoGenericGrid< M, T >::blockRead(), Go::GoTensorProductSpline< M, T >::evalVolume(), and Go::GoGenericGrid< M, T >::getDataPointer().

7.2.3.8 template<int S, int C, int D, template< int, int, int > class BlockType> void BlockStructure< S, C, D, BlockType >::write (std::ostream & *os*, bool *ascii* = false) const [inline]

write the complete **blockstructure**(p. 66) to an output stream

Parameters:

os the output stream where the **block**(p. 66) will be written

ascii if the user sets this argument to 'true', the structure will be written in ASCII format, else it will be written in BINARY format

Definition at line 77 of file BlockStructure_templates.h.

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

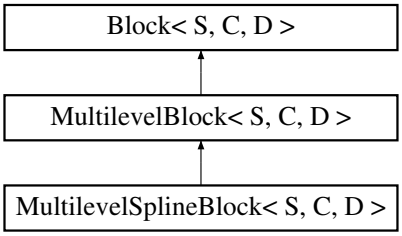
- BlockStructure.h
- BlockStructure_templates.h
- SampleBlock_templates.h

7.3 MultilevelBlock< S, C, D > Class Template Reference

This is the (abstract) base class for all **blocks**(p. 66) supporting multiple **levels**(p. 67) of detail.

`#include <Block.h>`

Inheritance diagram for MultilevelBlock< S, C, D >::



Public Member Functions

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

  Return the number of **geometry**(p. 66) **levels**(p. 67) *that this* **block**(p. 66) *contains.*

- virtual int **numResultLevels** (int result) const=0

  Return the number of **levels**(p. 67) *contained in this* **block**(p. 66) *to represent a specified* **result**(p. 68).

- virtual void **setResultLevel** (int resID, int lev)=0

  Sets the **active level**(p. 66) *for the* **result**(p. 68) *indexed 'resID' to 'lev'.*

- virtual void **setGeometryLevel** (int lev)=0

  Sets the **active geometry level**(p. 66) *to 'lev'.*

- virtual int **getResultLevel** (int resID) const=0

  Get the index of the currently **active level**(p. 66) *for the* **result**(p. 68) *indexed 'resID'.*

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

  Get the index of the currently **active geometry level**(p. 66).

- virtual bool **setResultPrecision** (int resID, double prec)=0

  If the **block**(p. 66) *contains information regarding the intrinsic error for each* **result**(p. 68) **level**(p. 67)*, this function allows for automatic selection of* **level**(p. 67) *for a given* **result**(p. 68)*, based on an error criterion.*

- virtual bool **setGeometryPrecision** (double prec)=0

  If the **block**(p. 66) *contains information regarding the intrinsic error for each* **geometry**(p. 66) **level**(p. 67)*, this function allows for automatic selection of* **level**(p. 67) *for the* **geometry**(p. 66)*, based on an error criterion.*

7.3.1 Detailed Description

template<int S, int C, int D> class MultilevelBlock< S, C, D >

This is the (abstract) base class for all **blocks**(p. 66) supporting multiple **levels**(p. 67) of detail.

Classes deriving from this class can be used in combination with the **BlockStructure**(p. 21) template class. The template parameters S, C and D represent respectively the **manifold**(p. 67) dimension of the **block**(p. 66) (number of **spatial parameters**(p. 68) that defines its **geometry**(p. 66)), the number of **auxiliary parameters**(p. 66), and the dimension of the **geometric space**(p. 66) where the **block**(p. 66) is situated. For a more detailed explanation, refer to the **main page**(p. 1). The **block**(p. 66) is defined by its **geometry**(p. 66), as well as an optional number of **results**(p. 68) , which are scalar fields defined on the volume occupied by the **block**(p. 66). Both the block's **geometry**(p. 66) and **result**(p. 68) *fields* are defined using M-variate splines, where M is equal to S+C. Moreover, both the **geometry**(p. 66) and each of the **results**(p. 68) can have multiple representations, with different **level**(p. 67) of detail, each represented by a different set of splines. Usually, these **levels**(p. 67) are all approximations with different accuracy of an "original, exact representation". The user can set the **active level**(p. 66) for the **geometry**(p. 66) and each of the **results**(p. 68), and in this way impose which spline representation should be returned when the **geometrySplines()**(p. 17) or **resultSplines()**(p. 19) functions are called.

Definition at line 213 of file Block.h.

7.3.2 Member Function Documentation

7.3.2.1 template<int S, int C, int D> virtual int MultilevelBlock< S, C, D >::getGeometryLevel () const [pure virtual]

Get the index of the currently **active geometry level**(p. 66).

Returns:
the index of the currently **active geometry level**(p. 66)

Reimplemented from **Block< S, C, D >** (p. 17).

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 39).

7.3.2.2 template<int S, int C, int D> virtual int MultilevelBlock< S, C, D >::getResultLevel (int *resID*) const [pure virtual]

Get the index of the currently **active level**(p. 66) for the **result**(p. 68) indexed 'resID'.

Parameters:
resID the **result**(p. 68) for which we want to know the **active level**(p. 66)

Returns:
the index of the **active level**(p. 66) for the specified **result**(p. 68)

Reimplemented from **Block< S, C, D >** (p. 18).

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 41).

7.3.2.3 template<int S, int C, int D> virtual int MultilevelBlock< S, C, D >::numGeometryLevels () const [pure virtual]

Return the number of **geometry**(p. 66) **levels**(p. 67) that this **block**(p. 66) contains.

Returns:
the number of **geometry**(p. 66) **levels**(p. 67) that this **block**(p. 66) contains

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 43).

7.3.2.4 template<int S, int C, int D> virtual int MultilevelBlock< S, C, D >::numResultLevels (int *result*) const [pure virtual]

Return the number of **levels**(p. 67) contained in this **block**(p. 66) to represent a specified **result**(p. 68).

Parameters:
\ref result "result" the index of the **result**(p. 68) for which we want to know the number of **levels**(p. 67)

Returns:
the number of **levels**(p. 67) for the specified **result**(p. 68)

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 43).

7.3.2.5 template<int S, int C, int D> virtual void MultilevelBlock< S, C, D >::setGeometryLevel (int *lev*) [pure virtual]

Sets the **active geometry level**(p. 66) to 'lev'.

Parameters:
lev the number of the **level**(p. 67) we want to set as 'active'. Valid values are from 0 to **numGeometryLevels()**(p. 31) - 1.

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 45).

7.3.2.6 template<int S, int C, int D> virtual bool MultilevelBlock< S, C, D >::setGeometryPrecision (double *prec*) [pure virtual]

If the **block**(p. 66) contains information regarding the intrinsic error for each **geometry**(p. 66) **level**(p. 67), this function allows for automatic selection of **level**(p. 67) for the **geometry**(p. 66), based on an error criterion.

The error of the chosen **level**(p. 67) should be inferior to the given precision, while still choosing a **level**(p. 67) containing as little information as possible (as low on detail as possible). If there is no sufficiently precise **level**(p. 67), the most accurate **level**(p. 67) is selected, and 'false' is returned. On the other hand, if there *are* sufficiently precise **levels**(p. 67) present, the least detailed of these will be chosen, and 'true' is returned. If the **block**(p. 66) type does not this kind of automatic **level**(p. 67) selection, an exception should be thrown.

Parameters:
prec the maximum tolerated error (as compared with the exact representation).

Returns:
'true' if it managed to find a suitable **level**(p. 67), 'false' if no sufficiently accurate **level**(p. 67) could be found.

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 46).

7.3.2.7 template<int S, int C, int D> virtual void MultilevelBlock< S, C, D >::setResultLevel (int *resID*, int *lev*) [pure virtual]

Sets the **active level**(p. 66) for the **result**(p. 68) indexed 'resID' to 'lev'.

Parameters:
resID the index of the **result**(p. 68) for which we want to set the **active level**(p. 66). This index should be between 0 and **numResults()**(p. 18) - 1.

lev the number of the **level**(p. 67) that we want to set as 'active' for this **result**(p. 68). This should be between 0 and **numResultLevels()**(p. 31) - 1, where the argument to **numResultLevels()**(p. 31) is 'resID'.

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 46).

7.3.2.8 template<int S, int C, int D> virtual bool MultilevelBlock< S, C, D >::setResultPrecision (int *resID*, double *prec*) [pure virtual]

If the **block**(p. 66) contains information regarding the intrinsic error for each **result**(p. 68) **level**(p. 67), this function allows for automatic selection of **level**(p. 67) for a given **result**(p. 68), based on an error criterion.

The error of the chosen **level**(p. 67) should be inferior to the given precision, while still choosing a **level**(p. 67) containing as little information as possible (as low on detail as possible). If there is no sufficiently precise **level**(p. 67), the most accurate **level**(p. 67) is selected, and 'false' is returned. On the other hand, if there *are* sufficiently precise **levels**(p. 67) present, the least detailed of these will be chosen, and 'true' is returned. If the **block**(p. 66) type does not this kind of automatic **level**(p. 67) selection, an exception should be thrown.

Parameters:
resID the **result**(p. 68) for which we want to set the **active level**(p. 66)

prec the maximum tolerated error (as compared with the exact representation).

Returns:
'true' if it managed to find a suitable **level**(p. 67), 'false' if no sufficiently accurate **level**(p. 67) could be found.

Implemented in **MultilevelSplineBlock< S, C, D >** (p. 46).

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

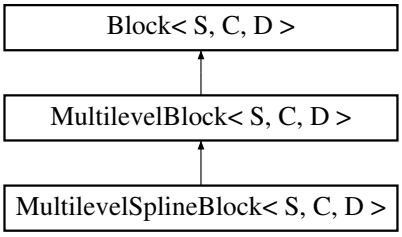
- Block.h

7.4 MultilevelSplineBlock< S, C, D > Class Template Reference

This is a **multilevel**(p. 67) **block**(p. 66) class with was designed with the aim of representing and compressing simulation grids.

`#include <MultilevelSplineBlock.h>`

Inheritance diagram for MultilevelSplineBlock< S, C, D >::



Public Member Functions

- **MultilevelSplineBlock** ()

  Constructor making an "invalid" MultilevelSplineBlock. It can not be used directly, but can be assigned to other **blocks**(p. 66).

- **MultilevelSplineBlock** (const Go::Array< **Go::GoGenericGrid**< S+C, double > *, D > &geometry, const std::vector< **Go::GoGenericGrid**< S+C, double > * > &results, double basis_tol)

  Constructor making a MultilevelSplineBlock containing one, single **level**(p. 67)*, the one considered to be 'exact'.*

- **MultilevelSplineBlock** (const **MultilevelSplineBlock**< S, C, D > &)

  copy constructor

- **MultilevelSplineBlock** & **operator=** (const **MultilevelSplineBlock**< S, C, D > &)

  assignment operator (exception safe)

- void **swap** (**MultilevelSplineBlock**< S, C, D > &rhs)

  function to quickly swap two **blocks**(p. 66)

- void **setWriteMode** (bool results=true, bool errors=true, bool all_levels=true, bool use_float=true) const

  Set the writing mode for the **block**(p. 66)*, decinding which components should be written to stream when calling the* **write()**(p. 47) *function.*

- virtual void **write** (std::ostream &os, bool ascii=false) const

  Write the **block**(p. 66) *to an output stream.*

- virtual void **read** (std::istream &is, bool ascii=false)

  Read the **block**(p. 66) *from an input stream. This may or may not be a complete definition of the* **block**(p. 66) *- it depends on which parts were available from the stream. Read* **setWriteMode()**(p. 47) *for more.*

- bool **geometryIsHierarchisable** () const

  This function reports whether **geometry**(p. 66) *approximations can be generated from the information currently present in the* **block**(p. 66).

- bool **resultIsHierarchisable** (int res) const

  This function reports whether approximations for a given **result**(p. 68) *can be generated from the information currently present in the* **block**(p. 66).

- void **makeGeometryHierarchy** (const std::vector< Go::Array< boost::shared_ptr< **Go::GoApproximator**< S+C, double > >, S+C > > &method)

  From a **geometry**(p. 66) **base level**(p. 66) *(supposedly present), generate a "hierarchy" of* **levels**(p. 67) *that approximates this* **base level**(p. 66).

- void **makeResultHierarchy** (const std::vector< Go::Array< boost::shared_ptr< **Go::GoApproximator**< S+C, double > >, S+C > > &method, int resultID)

  From a **result**(p. 68) **base level**(p. 66) *(supposedly present), generate a "hierarchy" of* **levels**(p. 67) *that approximates this* **base level**(p. 66).

- virtual Go::BoundingBox **boundingBox** () const

  Returns a bounding box enclosing the geometric area occupied by the **block**(p. 66).

- virtual void **getCornerPosition** (const bool *max, const double *C_values, double *res) const

  Function to retrieve the position of a block's **corner**(p. 66) *in geometrical space for a given set of* **auxiliary parameters**(p. 66) *(C-parameters).*

- virtual const Go::Array< **Go::GoTensorProductSpline**< S+C, double >, D > **geometrySplines** () const

  Returns the **multivariate**(p. 67) *splines that define this block's* **geometry**(p. 66).

- virtual const **Go::GoTensorProductSpline**< S+C, double > **resultSplines** (int resultID) const

  Returns the **multivariate**(p. 67) *spline describing one of the block's* **results**(p. 68).

- virtual void **setResultLevel** (int resID, int lev)

  Sets the **active level**(p. 66) *for the* **result**(p. 68) *indexed 'resID' to 'lev'.*

- virtual void **setGeometryLevel** (int lev)

  Sets the **active geometry level**(p. 66) *to 'lev'.*

- virtual bool **setResultPrecision** (int resID, double prec)

  If the **block**(p. 66) *contains information regarding the intrinsic error for each* **result**(p. 68) **level**(p. 67)*, this function allows for automatic selection of* **level**(p. 67) *for a given* **result**(p. 68)*, based on an error criterion.*

- virtual bool **setGeometryPrecision** (double prec)

  If the **block**(p. 66) *contains information regarding the intrinsic error for each* **geometry**(p. 66) **level**(p. 67)*, this function allows for automatic selection of* **level**(p. 67) *for the* **geometry**(p. 66)*, based on an error criterion.*

- virtual int **numGeometryLevels** () const

Return the number of **geometry**(p. 66) **levels**(p. 67) *that this* **block**(p. 66) *contains.*

- virtual int **numResultLevels** (int result) const

  Return the number of **levels**(p. 67) *contained in this* **block**(p. 66) *to represent a specified* **result**(p. 68).

- virtual int **getResultLevel** (int resID) const

  Get the index of the currently **active level**(p. 66) *for the* **result**(p. 68) *indexed 'resID'.*

- virtual int **getGeometryLevel** () const

  Get the index of the currently **active geometry level**(p. 66).

- virtual int **numResults** () const

  Returns the number of **results**(p. 68) *that this* **block**(p. 66) *contains.*

- virtual std::vector< boost::shared_ptr< Go::GeomObject > > **outline** (double *aux_par_values) const

  Returns the **outline**(p. 67) *of the* **block**(p. 66).

- const Go::Array< **Go::GoBorrowedMVGrid**< S+C, double >, D > **getGeometryError** () const

  Returns the grids containing the geometric error between the currently **active level**(p. 66) *and the* **base level**(p. 66).

- const **Go::GoBorrowedMVGrid**< S+C, double > **getResultError** (int resID) const

  Returns the grid containing the error between the currently **active level**(p. 66) *and the* **base level**(p. 66) *for the specified* **result**(p. 68).

- const std::vector< double > & **getOriginalKnotvector** (int tdir) const

  Return the knotvector for the spline expressing the "original model", or the **base level**(p. 66).

- Go::Array< std::pair< double, double >, D > **getGeometryRanges** () const

  Get the boundaries (minimum and maximum values) for the volume occupied by the **block**(p. 66) *in* **geometric space**(p. 66).

- std::pair< double, double > **getResultRange** (int res) const

  Get the minimum and maximum value for the specified **result**(p. 68).

- Go::Array< double, D > **getMaxGeometryError** () const

  Return the maximum value of the absolute **geometry**(p. 66) *error between the currently* **active geometry level**(p. 66) *and the* **base level**(p. 66), *for each of the D spatial dimensions.*

- double **getMaxResultError** (int res) const

  Return the maximum value of the absolute **result**(p. 68) *error between the currently* **active level**(p. 66) *and the* **base level**(p. 66), *for a given* **result**(p. 68).

7.4.1 Detailed Description

template<int S, int C, int D> class MultilevelSplineBlock< S, C, D >

This is a **multilevel**(p. 67) **block**(p. 66) class with was designed with the aim of representing and compressing simulation grids.

It contains functionalty for generating several **levels**(p. 67) of detail based on a given, "exact" representation. The exact representation is given when an object of this class is first constructed, using scalar grids (**Go::GoGenericGrid**) to specify the geometric position and **result**(p. 68) values. The MultilevelSplineBlock represents these grid internally as *linear splines*, interpolating values between grid nodes linearly. When generating approximations, other spline **orders**(p. 68) can be used.

Independently of the **level**(p. 67) of detail currently active, the user can always access the original knotvector (the one generated in the constructor in order to define the linear spline), using the **getOriginalKnotvector()**(p. 40) function. This allows the user to determine for which parameter values (the knots of the knotvector) he should evaluate his spline representation in order to retrieve the original gridpoints.

Definition at line 43 of file MultilevelSplineBlock.h.

7.4.2 Constructor & Destructor Documentation

7.4.2.1 template<int S, int C, int D> MultilevelSplineBlock< S, C, D >::MultilevelSplineBlock (const Go::Array< Go::GoGenericGrid< S+C, double > *, D > & *geometry*, const std::vector< Go::GoGenericGrid< S+C, double > * > & *results*, double *basis_tol*) [inline]

Constructor making a MultilevelSplineBlock containing one, single **level**(p. 67), the one considered to be 'exact'.

The geometric position of the grid nodes, as well as the scalar **results**(p. 68) assigned to them, are specified by a collection of **Go::GoGenericGrid** s. These grids should always be of the same *shape*, ie. along a given index, all the grids should have the same number of gridpoints.

Parameters:

geometry an array of D grids specifying the spatial position of the grid nodes in each of the D spatial directions.

results a vector of grids, each specifying the value of a certain scalar field defined on the grid in each gridpoint.

basis_tol a value concerning the tolerance to be used when working with knotvectors, to define whether two knots are equal or not. Should be set low; an acceptable value is often 1.0e-8 or smaller.

Definition at line 23 of file MultilevelSplineBlock_templates.h.

7.4.3 Member Function Documentation

7.4.3.1 template<int S, int C, int D> Go::BoundingBox MultilevelSplineBlock< S, C, D >::boundingBox () const [inline, virtual]

Returns a bounding box enclosing the geometric area occupied by the **block**(p. 66).

Returns:
the a bounding box enclosing the geometric area occupied by the **block**(p. 66)

Implements **Block< S, C, D >** (p. 16).

Definition at line 865 of file MultilevelSplineBlock_templates.h.

7.4.3.2 template<int S, int C, int D> bool MultilevelSplineBlock< S, C, D >::geometryIsHierarchisable () const [inline]

This function reports whether **geometry**(p. 66) approximations can be generated from the information currently present in the **block**(p. 66).

This class contains functionality for generating a user-defined hierarchy of **levels**(p. 67) *of detail* . The requirement, however, is that there is a **base level**(p. 66) present in the **block**(p. 66), from which the other **levels**(p. 67) can be generated. Such a **base level**(p. 66) is supposed to be the 'exact' representation of the **block**(p. 66), and is usually set when the **block**(p. 66) is constructed. However, it may be that the current **block**(p. 66) does not contain this **base level**(p. 66), notably because the **block**(p. 66) has been **read()**(p. 44) from a stream that did not provide it. In that case, the **levels**(p. 67) contained in the **block**(p. 66) cannot be changed.

If there is a **base level**(p. 66) for **geometry**(p. 66) present in the current **block**(p. 66), then it is possible to generate approximating **levels**(p. 67), and this function will return 'true'. In the opposite case, the return value will be 'false'.

Returns:
true or false depending on whether approximating **levels**(p. 67) can be generated or not.

Definition at line 1335 of file MultilevelSplineBlock_templates.h.

Referenced by MultilevelSplineBlock< S, C, D >::makeGeometryHierarchy().

7.4.3.3 template<int S, int C, int D> const Go::Array< Go::GoTensorProductSpline< S+C, double >, D > MultilevelSplineBlock< S, C, D >::geometrySplines () const [inline, virtual]

Returns the **multivariate**(p. 67) splines that define this block's **geometry**(p. 66).

There are D such splines, each expressing the position of the **block**(p. 66) in one spatial dimension, as a function of the S+C parameters. (For instance, in 3D, we would have one spline expressing the x-coordinate, another expressing the y-coordinate and a third one expressing the z-coordinate).

Returns:
A D-sized array containing the **multivariate**(p. 67) splines describing the **geometry**(p. 66) of the **block**(p. 66) in each of the spatial dimensions.

Implements **Block< S, C, D >** (p. 17).

Definition at line 925 of file MultilevelSplineBlock_templates.h.

7.4.3.4 template<int S, int C, int D> void MultilevelSplineBlock< S, C, D >::getCornerPosition (const bool * *max*, const double * *C_values*, double * *res*) const [inline, virtual]

Function to retrieve the position of a block's **corner**(p. 66) in geometrical space for a given set of **auxiliary parameters**(p. 66) (C-parameters).

The block's **corners**(p. 66) are defined as its **0-interfaces**(p. 67) (see **Block connectivity**(p. 2)). A **corner**(p. 66) is characterized by all the **spatial parameters**(p. 68) taking up an extremal value (either 0 or 1). The *max* argument is used to specify which **spatial parameters**(p. 68) are 0 and which are 1. It should point to an array of S bools. If *max*[i] = `true`, it means that for the requested **corner**(p. 66), the **spatial parameter**(p. 68) *i* is set to 1, else it is set to 0. The spatial coordinates of the requested **corner**(p. 66) are written to the memory location pointed to by *res*. This area should of course be big enough to store D `double` s.

Parameters:

max Pointer to an array of S `bool` s, specifying whether the corresponding **spatial parameter**(p. 68) is *maximum* (1) or *minimum* (0) at this **corner**(p. 66). A value of `true` means *maximum*.

C_values Pointer to an array of C `double` s, specifying the values of the **auxiliary parameters**(p. 66) to use when evaluating the position of the **corner**(p. 66) corners.

res pointer to the memory location where the **corner**(p. 66) position (D `double` s) will be written. It is the user's responsibility to allocate enough memory.

Implements **Block< S, C, D >** (p. 17).

Definition at line 896 of file MultilevelSplineBlock_templates.h.

7.4.3.5 template<int S, int C, int D> const Go::Array< Go::GoBorrowedMVGrid< S+C, double >, D > MultilevelSplineBlock< S, C, D >::getGeometryError () const [inline]

Returns the grids containing the geometric error between the currently **active level**(p. 66) and the **base level**(p. 66).

The error grids have the same shape (number of nodes for each index) as the original "base-grids" that was first used to define the **geometry**(p. 66) of this **block**(p. 66). Each node in the original grid therefore has an associated value in the error grid, describing the error between the currently **active level**(p. 66) and the original grid (the **base level**(p. 66)).

Note:

Whether or not this function works does not depend on the presence of the **base level**(p. 66) itself, but rather on the presence of the error grid generated when the **active level**(p. 66) was first established. This error grid is not necessarily present if this **block**(p. 66) has been **read()**(p. 44) from a stream rather than being generated "from scratch" with the **makeGeometryHierarchy()**(p. 42) and **makeResultHierarchy()**(p. 42) functions. In any case, if the requested error grid is not present, an exception will be thrown.
The returned error grids do not own their coefficients, they only share them with the error grids internally stored in the **block**(p. 66).

Returns:

An array of D grids, each expressing the error in the corresponding spatial dimension between the currently **active level**(p. 66) and the original base object.

Definition at line 1253 of file MultilevelSplineBlock_templates.h.

7.4.3.6 template<int S, int C, int D> int MultilevelSplineBlock< S, C, D >::getGeometryLevel () const [inline, virtual]

Get the index of the currently **active geometry level**(p. 66).

Returns:
the index of the currently **active geometry level**(p. 66)

Implements **MultilevelBlock< S, C, D >** (p. 30).

Definition at line 994 of file MultilevelSplineBlock_templates.h.

7.4.3.7 template<int S, int C, int D> Go::Array< std::pair< double, double >, D > MultilevelSplineBlock< S, C, D >::getGeometryRanges () const [inline]

Get the boundaries (minimum and maximum values) for the volume occupied by the **block**(p. 66) in **geometric space**(p. 66).

...in other words, return the "bounding box" of the **block**(p. 66).

Note:
The values returned are for the "exact representation" of the **block**(p. 66)'s **geometry**(p. 66), as specified by its **base level**(p. 66). The actual bounding box of the currently **active level**(p. 66) may differ slightly.

Returns:
a Go::Array with D entries of `std::pair<double, double>`, where each entry represent one dimension in **geometric space**(p. 66). The first value in the `std::pair` is the minimum value of the **block**(p. 66)'s position in space along this dimension, the second value is the maximum value.

Definition at line 1191 of file MultilevelSplineBlock_templates.h.

7.4.3.8 template<int S, int C, int D> Go::Array< double, D > MultilevelSplineBlock< S, C, D >::getMaxGeometryError () const [inline]

Return the maximum value of the absolute **geometry**(p. 66) error between the currently **active geometry level**(p. 66) and the **base level**(p. 66), for each of the D spatial dimensions.

Note:
Even in the case that the **block**(p. 66) does not contain a complete errorgrid for the **active level**(p. 66) (so that the **getGeometryError()**(p. 38) function cannot be called), this function can still be used to retrieve the error's maximum value.

Returns:
A Go::Array containing D values, each representing, for one dimension, the maximum error between the currently **active level**(p. 66) and the **base level**(p. 66).

Definition at line 1211 of file MultilevelSplineBlock_templates.h.

7.4.3.9 template<int S, int C, int D> double MultilevelSplineBlock< S, C, D >::getMaxResultError (int *res*) const [inline]

Return the maximum value of the absolute **result**(p. 68) error between the currently **active level**(p. 66) and the **base level**(p. 66), for a given **result**(p. 68).

Note:

Even in the case that the **block**(p. 66) does not contain a complete errorgrid for the **active level**(p. 66) (so that the **getResultError()**(p. 40) function cannot be called), this function can still be used to retrieve the error's maximum value.

Parameters:

res the index of the **result**(p. 68) for which we are requesting the extremal error value on the **active level**(p. 66).

Returns:

The value of the maxiumum error between the current **level**(p. 67) and the **base level**(p. 66), for the specified **result**(p. 68).

Definition at line 1219 of file MultilevelSplineBlock_templates.h.

7.4.3.10 template<int S, int C, int D> const std::vector< double > & MultilevelSplineBlock< S, C, D >::getOriginalKnotvector (int *tdir*) const [inline]

Return the knotvector for the spline expressing the "original model", or the **base level**(p. 66).

This function will always work, regardless of whether the **base level**(p. 66) is actually present in this **block**(p. 66) or not. The knotvector can be used to determine the parameter values for which we need to evaluate the splines (on any given **level**(p. 67)) to "reconstruct" the position and values of the nodes in the original grid.

Note:

This knotvector is used to describe a spline basis of **order**(p. 68) 2. That means that the first and last knot in the vector are there for border purposes, and could be ignored when evaluating "gridpoints". The number of gridpoints in the original grid, for a given parameter, is equal to the length of the corresponding knotvector minus two.
The knotvector is the same whether we consider **geometry**(p. 66) or any of the **results**(p. 68).

Parameters:

tdir specify which knotvector we want to have returned. Each of the S+C parameters have their own knotvector.

Returns:

the knotvector for the specified parameter for the **base level**(p. 66) spline.

Definition at line 1283 of file MultilevelSplineBlock_templates.h.

7.4.3.11 template<int S, int C, int D> const Go::GoBorrowedMVGrid< S+C, double > MultilevelSplineBlock< S, C, D >::getResultError (int *resID*) const [inline]

Returns the grid containing the error between the currently **active level**(p. 66) and the **base level**(p. 66) for the specified **result**(p. 68).

The error grid has the same shape (number of nodes for each index) as the original "base-grid" that was first used to define this **result**(p. 68) for this **block**(p. 66). Each node in the original grid therefore has an associated value in the error grid, describing the error between the currently **active level**(p. 66) and the original grid (the **base level**(p. 66)).

Note:

Whether or not this function works does not depend on the presence of the **base level**(p. 66) itself, but rather on the presence of the error grid generated when the **active level**(p. 66) was first established. This error grid is not necessarily present if this **block**(p. 66) has been **read()**(p. 44) from a stream rather than being generated "from scratch" with the **makeGeometryHierarchy()**(p. 42) and **makeResultHierarchy()**(p. 42) functions. In any case, if the requested error grid is not present, an exception will be thrown.

The returned error grid does not own its coefficients, it only shares them with the error grid internally stored in the **block**(p. 66).

Parameters:

resID the index for the **result**(p. 68) for which we want to have the error grid. Valid values for resID is from 0 to **numResults()**(p. 44) - 1.

Definition at line 1270 of file MultilevelSplineBlock_templates.h.

7.4.3.12 template<int S, int C, int D> int MultilevelSplineBlock< S, C, D >::getResultLevel (int *resID*) const [inline, virtual]

Get the index of the currently **active level**(p. 66) for the **result**(p. 68) indexed 'resID'.

Parameters:

resID the **result**(p. 68) for which we want to know the **active level**(p. 66)

Returns:

the index of the **active level**(p. 66) for the specified **result**(p. 68)

Implements **MultilevelBlock< S, C, D >** (p. 30).

Definition at line 982 of file MultilevelSplineBlock_templates.h.

7.4.3.13 template<int S, int C, int D> std::pair< double, double > MultilevelSplineBlock< S, C, D >::getResultRange (int *res*) const [inline]

Get the minimum and maximum value for the specified **result**(p. 68).

Note:

The values returned are for the "exact representation" of the requested **result**(p. 68), as specified by its **base level**(p. 66) in the **block**(p. 66). The actual extremal values of the currently **active level**(p. 66) may differ slightly.

Parameters:

res the index of the **result**(p. 68) for which we are requesting the extremal values.

Returns:

A pair of `double` s representing respectively the minimum and the maximum value for the specified **result**(p. 68) scalar field.

Definition at line 1199 of file MultilevelSplineBlock_templates.h.

7.4.3.14 template<int S, int C, int D> void MultilevelSplineBlock< S, C, D >::makeGeometryHierarchy (const std::vector< Go::Array< boost::shared_ptr< Go::GoApproximator< S+C, double > >, S+C > > & *method*) [inline]

From a **geometry**(p. 66) **base level**(p. 66) (supposedly present), generate a "hierarchy" of **levels**(p. 67) that approximates this **base level**(p. 66).

If such **levels**(p. 67) are already defined, they will be removed. If there is no base **level**(p. 67) present in the **block**(p. 66) (see **geometryIsHierarchisable()**(p. 37)), an exception will be thrown. The **levels**(p. 67) will be defined by spline functions approximating the original grid at **base level**(p. 66). These spline functions, of the class **Go::GoTensorProductSpline**, will be approximated using techniques specified by objects deriving from the **Go::GoApproximator** class. For more information about specifying approximation techniques, read the documentation for that class, as well as the one for the **Go::GoTensorProductSpline::fit()** function.

Note:

Each **level**(p. 67) in the hierarchy should be progressive more detailed (more accurate) than the preceding one. This conforms to the idea of "hierarchy", where there is a natural 'ordening' of the **levels**(p. 67) from "coarse" to "fine". This ordering is not verified by this function in any way, but the user is encouraged to observe it. The **setGeometryPrecision()**(p. 46) function assumes that the **levels**(p. 67) are ordered in this way - if this is not the case, that function will not work properly.
Upon generation of each **level**(p. 67), the knotvectors of the defining splines will all be rescaled to the range $[0, 1]$.

Parameters:

method a vector specifying how to generate each **level**(p. 67) in the hierarchy. Each entry in the vector correspond to one **level**(p. 67); the first entry in the vector should describe the coarsest **level**(p. 67). The details on constructing each **level**(p. 67) is contained in a Go::Array of S+C elements, where each element refer to how to approximate the **geometry**(p. 66) splines (all D of them) for the corresponding parameter. This is described by a (**shared pointer**(p. 68) to a) **Go::GoApproximator** object. Read the documentation on this object to understand how to specify the approximation to be made.

Definition at line 1002 of file MultilevelSplineBlock_templates.h.

References MultilevelSplineBlock< S, C, D >::geometryIsHierarchisable().

7.4.3.15 template<int S, int C, int D> void MultilevelSplineBlock< S, C, D >::makeResultHierarchy (const std::vector< Go::Array< boost::shared_ptr< Go::GoApproximator< S+C, double > >, S+C > > & *method*, int *resultID*) [inline]

From a **result**(p. 68) **base level**(p. 66) (supposedly present), generate a "hierarchy" of **levels**(p. 67) that approximates this **base level**(p. 66).

If such **levels**(p. 67) are already defined, they will be removed. If there is no **base level**(p. 66) present in the **block**(p. 66) for this **result**(p. 68) (see **resultIsHierarchisable()**(p. 45)), an exception will be thrown. The **levels**(p. 67) will be defined by spline functions approximating the original grid at **base level**(p. 66). These spline functions, of the class **Go::GoTensorProductSpline**, will be approximated using techniques specified by objects deriving from the **Go::GoApproximator** class. For more information about specifying approximation techniques, read the documentation for that class, as well as the one for the **Go::GoTensorProductSpline::fit()** function.

Note:

Each **level**(p. 67) in the hierarchy should be progressive more detailed (more accurate) than the preceding one. This conforms to the idea of "hierarchy", where there is a natural 'ordening' of the **levels**(p. 67) from "coarse" to "fine". This orderingf is not verified by this function in any way, but the user is encouraged to observe it. The **setResultPrecision()**(p. 46) function assumes that the **levels**(p. 67) are ordered in this way - if this is not the case, that function will not work properly.

Upon generation of each **level**(p. 67), the knotvectors of the defining splines will all be rescaled to the range $[0, 1]$.

Parameters:

method a vector specifying how to generate each **level**(p. 67) in the hierarchy. Each entry in the vector correspond to one **level**(p. 67); the first entry in the vector should describe the coarsest **level**(p. 67). The details on constructing each **level**(p. 67) is contained in a Go::Array of S+C elements, where each element refer to how to approximate the **result**(p. 68) spline for the corresponding parameter. This is described by a (**shared pointer**(p. 68) to a) **Go::GoApproximator** object. Read the documentation on this object to understand how to specify the approximation to be made.

resultID the index of the **result**(p. 68) for which we want to generate the hierarchy. Valid values are from 0 to **numResults()**(p. 44) - 1.

Definition at line 1058 of file MultilevelSplineBlock_templates.h.

References MultilevelSplineBlock< S, C, D >::resultIsHierarchisable().

7.4.3.16 template<int S, int C, int D> int MultilevelSplineBlock< S, C, D >::numGeometryLevels () const [inline, virtual]

Return the number of **geometry**(p. 66) **levels**(p. 67) that this **block**(p. 66) contains.

Returns:

the number of **geometry**(p. 66) **levels**(p. 67) that this **block**(p. 66) contains

Implements **MultilevelBlock< S, C, D >** (p. 31).

Definition at line 1161 of file MultilevelSplineBlock_templates.h.

7.4.3.17 template<int S, int C, int D> int MultilevelSplineBlock< S, C, D >::numResultLevels (int *result*) const [inline, virtual]

Return the number of **levels**(p. 67) contained in this **block**(p. 66) to represent a specified **result**(p. 68).

Parameters:

ref result "result" the index of the **result**(p. 68) for which we want to know the number of **levels**(p. 67)

Returns:

the number of **levels**(p. 67) for the specified **result**(p. 68)

Implements **MultilevelBlock< S, C, D >** (p. 31).

Definition at line 1169 of file MultilevelSplineBlock_templates.h.

7.4.3.18 template<int S, int C, int D> int MultilevelSplineBlock< S, C, D >::numResults () const [inline, virtual]

Returns the number of **results**(p. 68) that this **block**(p. 66) contains.

The **results**(p. 68) will always be refered to by using their index number 0 to **numResults()**(p. 44) - 1

Returns:
the number of **results**(p. 68) that this **block**(p. 66) contains

Implements **Block< S, C, D >** (p. 18).

Definition at line 1182 of file MultilevelSplineBlock_templates.h.

7.4.3.19 template<int S, int C, int D> std::vector< boost::shared_ptr< Go::GeomObject > > MultilevelSplineBlock< S, C, D >::outline (double * *aux_par_values*) const [inline, virtual]

Returns the **outline**(p. 67) of the **block**(p. 66).

The **outline**(p. 67) is defined as the union of all the block's **1-interfaces**(p. 67) (edges), which is collected, expressed as Go-objects (usually Go::SplineCurve s) and returned as the **result**(p. 68) of the function call. This function can be used for all instantiations of the **Block**(p. 15) template, provided that S > 0. The (fixed) values for the **auxiliary parameters**(p. 66) are specified by the user in an C-sized array pointed to by the argument aux_par_values.

Parameters:
aux_par_values points to an array of C *double* s, defining the values of the **auxiliary parameters**(p. 66) for which we want the **outline**(p. 67) of the **block**(p. 66).

Returns:
a vector of Go-objects (usually Go::SplineCurves) expressing the **outline**(p. 67) of the **block**(p. 66) for the specified values for the **auxiliary parameters**(p. 66).

Implements **Block< S, C, D >** (p. 19).

Definition at line 1231 of file MultilevelSplineBlock_templates.h.

7.4.3.20 template<int S, int C, int D> void MultilevelSplineBlock< S, C, D >::read (std::istream & *is*, bool *ascii* = false) [inline, virtual]

Read the **block**(p. 66) from an input stream. This may or may not be a complete definition of the **block**(p. 66) - it depends on which parts were available from the stream. Read **setWriteMode()**(p. 47) for more.

Parameters:
is the input stream
ascii if set to 'true', the contents will be read in ASCII format, if not, the contents will be read in BINARY format.

Implements **Block< S, C, D >** (p. 19).

Definition at line 556 of file MultilevelSplineBlock_templates.h.

References Go::GoGenericGrid< M, T >::read_BINARY(), Go::GoTensorProductSpline< M, T >::read_BINARY(), and Go::GoTensorProductSpline< M, T >::setDataPointer().

7.4.3.21 template<int S, int C, int D> bool MultilevelSplineBlock< S, C, D >::resultIsHierarchisable (int *res*) const [inline]

This function reports whether approximations for a given **result**(p. 68) can be generated from the information currently present in the **block**(p. 66).

This class contains functionality for generating a user-defined hierarchy of **levels**(p. 67) *of detail* . The requirement, however, is that there is a **base level**(p. 66) present in the **block**(p. 66), from which the other **levels**(p. 67) can be generated. Such a **base level**(p. 66) is supposed to be the 'exact' representation of the **block**(p. 66), and is usually set when the **block**(p. 66) is constructed. However, it may be that the current **block**(p. 66) does not contain this **base level**(p. 66), notably because the **block**(p. 66) has been **read()**(p. 44) from a stream that did not provide it. In that case, the **levels**(p. 67) contained in the **block**(p. 66) cannot be changed.

If there is a **base level**(p. 66) for *the* indicated **result**(p. 68) present in the current **block**(p. 66), then it is possible to generate approximating **levels**(p. 67), and this function will return 'true'. In the opposite case, the return value will be 'false'.

Parameters:

res the index of the **result**(p. 68) for which we want to know whether it is possible to generate approximations or not.

Returns:

true or false depending on whether approximating **levels**(p. 67) can be generated or not.

Definition at line 1294 of file MultilevelSplineBlock_templates.h.

Referenced by MultilevelSplineBlock< S, C, D >::makeResultHierarchy().

7.4.3.22 template<int S, int C, int D> const Go::GoTensorProductSpline< S+C, double > MultilevelSplineBlock< S, C, D >::resultSplines (int *resultID*) const [inline, virtual]

Returns the **multivariate**(p. 67) spline describing one of the block's **results**(p. 68).

Parameters:

resultID The index number of the requested **result**(p. 68). Valid values are from 0 to **numResults()**(p. 44) - 1.

Returns:

A **multivariate**(p. 67) spline describing the requested **result**(p. 68) scalar field

Implements **Block< S, C, D >** (p. 19).

Definition at line 945 of file MultilevelSplineBlock_templates.h.

7.4.3.23 template<int S, int C, int D> void MultilevelSplineBlock< S, C, D >::setGeometryLevel (int *lev*) [inline, virtual]

Sets the **active geometry level**(p. 66) to 'lev'.

Parameters:

lev the number of the **level**(p. 67) we want to set as 'active'. Valid values are from 0 to **numGeometryLevels()**(p. 43) - 1.

Implements **MultilevelBlock< S, C, D >** (p. 31).

Definition at line 971 of file MultilevelSplineBlock_templates.h.

7.4.3.24 template<int S, int C, int D> bool MultilevelSplineBlock< S, C, D >::setGeometryPrecision (double *prec*) [inline, virtual]

If the **block**(p. 66) contains information regarding the intrinsic error for each **geometry**(p. 66) **level**(p. 67), this function allows for automatic selection of **level**(p. 67) for the **geometry**(p. 66), based on an error criterion.

The error of the chosen **level**(p. 67) should be inferior to the given precision, while still choosing a **level**(p. 67) containing as little information as possible (as low on detail as possible). If there is no sufficiently precise **level**(p. 67), the most accurate **level**(p. 67) is selected, and 'false' is returned. On the other hand, if there *are* sufficiently precise **levels**(p. 67) present, the least detailed of these will be chosen, and 'true' is returned. If the **block**(p. 66) type does not this kind of automatic **level**(p. 67) selection, an exception should be thrown.

Parameters:

prec the maximum tolerated error (as compared with the exact representation).

Returns:

'true' if it managed to find a suitable **level**(p. 67), 'false' if no sufficiently accurate **level**(p. 67) could be found.

Implements **MultilevelBlock< S, C, D >** (p. 31).

Definition at line 1130 of file MultilevelSplineBlock_templates.h.

7.4.3.25 template<int S, int C, int D> void MultilevelSplineBlock< S, C, D >::setResultLevel (int *resID*, int *lev*) [inline, virtual]

Sets the **active level**(p. 66) for the **result**(p. 68) indexed 'resID' to 'lev'.

Parameters:

resID the index of the **result**(p. 68) for which we want to set the **active level**(p. 66). This index should be between 0 and **numResults()**(p. 44) - 1.

lev the number of the **level**(p. 67) that we want to set as 'active' for this **result**(p. 68). This should be between 0 and **numResultLevels()**(p. 43) - 1, where the argument to **numResultLevels()**(p. 43) is 'resID'.

Implements **MultilevelBlock< S, C, D >** (p. 32).

Definition at line 956 of file MultilevelSplineBlock_templates.h.

7.4.3.26 template<int S, int C, int D> bool MultilevelSplineBlock< S, C, D >::setResultPrecision (int *resID*, double *prec*) [inline, virtual]

If the **block**(p. 66) contains information regarding the intrinsic error for each **result**(p. 68) **level**(p. 67), this function allows for automatic selection of **level**(p. 67) for a given **result**(p. 68), based on an error criterion.

The error of the chosen **level**(p. 67) should be inferior to the given precision, while still choosing a **level**(p. 67) containing as little information as possible (as low on detail as possible). If there is no

sufficiently precise **level**(p. 67), the most accurate **level**(p. 67) is selected, and 'false' is returned. On the other hand, if there *are* sufficiently precise **levels**(p. 67) present, the least detailed of these will be chosen, and 'true' is returned. If the **block**(p. 66) type does not this kind of automatic **level**(p. 67) selection, an exception should be thrown.

Parameters:

resID the **result**(p. 68) for which we want to set the **active level**(p. 66)

prec the maximum tolerated error (as compared with the exact representation).

Returns:

'true' if it managed to find a suitable **level**(p. 67), 'false' if no sufficiently accurate **level**(p. 67) could be found.

Implements **MultilevelBlock< S, C, D >** (p. 32).

Definition at line 1108 of file MultilevelSplineBlock_templates.h.

7.4.3.27 template<int S, int C, int D> void MultilevelSplineBlock< S, C, D >::setWriteMode (bool *results* = true, bool *errors* = true, bool *all_levels* = true, bool *use_float* = true) const [inline]

Set the writing mode for the **block**(p. 66), decinding which components should be written to stream when calling the **write()**(p. 47) function.

This function is provided because the user does not necessarily want to write everything contained in the **block**(p. 66) to the stream (file or other). For instance, when writing to file, it may be that she only wants to write the compressed version, without taking care of the estimated errors, etc.

Parameters:

results - if 'true', then the **block**(p. 66)'s **results**(p. 68) will be written

errors - if 'true', then all error estimates (generated when establishing the **block**(p. 66)'s **levels**(p. 67) of detail) will be written.

all_levels - if 'true', information for all **levels**(p. 67) will be written, if not, then only the information pertaining to the current **level**(p. 67) is written.

use_float - if 'true', the **block**(p. 66)'s grid values / spline coefficients will be written using `float`, in order to save space. In the opposite case, the information will be written using `double` precision.

Note:

With this function, we set the internal *write mode* state. This state is mutable, and can be changed even on `const` **blocks**(p. 66).

Definition at line 1322 of file MultilevelSplineBlock_templates.h.

7.4.3.28 template<int S, int C, int D> void MultilevelSplineBlock< S, C, D >::write (std::ostream & *os*, bool *ascii* = false) const [inline, virtual]

Write the **block**(p. 66) to an output stream.

How much of the information that is actually written can be regulated with the **setWriteMode()**(p. 47) function.

Parameters:

os the output stream

ascii if set to 'true', the contents will be written in ASCII format, if not, the contents will be written in BINARY format (default).

Implements **Block< S, C, D >** (p. 20).

Definition at line 173 of file MultilevelSplineBlock_templates.h.

References Go::GoGenericGrid< M, T >::write_BINARY(), and Go::GoTensorProductSpline< M, T >::write_BINARY().

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

- MultilevelSplineBlock.h
- MultilevelSplineBlock_templates.h

7.5 BlockEnum::ParamConfig< S > Class Template Reference

An array holding a total of S different **ParamState**(p. 14) s.

`#include <BlockEnum.h>`

Public Member Functions

- **ParamConfig** ()

  Default constructor, setting all S **ParamState**(p. 14) *s to* RUN.

- bool **operator==** (const **ParamConfig** &rhs)

  Equality operator, returns true if all S **ParamState**(p. 14) *s of this object are equal to the corrsponding ones found in the 'rhs' object.*

- void **clear** ()

  Resets all S **ParamState**(p. 14) *s to* RUN.

- void **dump** (std::ostream &os)

  Dumps a textual description of the **ParamConfig**(p. 49) *to an output stream.*

7.5.1 Detailed Description

template<int S> class BlockEnum::ParamConfig< S >

An array holding a total of S different **ParamState**(p. 14) s.

This class is convenient for grouping together the **ParamState**(p. 14) s that defines a certain **interface**(p. 67) of a S-block. Read the section on **Block Connectivity**(p. 2) for more.

Definition at line 65 of file BlockEnum.h.

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

- BlockEnum.h
- BlockEnum_templates.h

7.6 BlockEnum::ParamConfigEnumeration< S > Class Template Reference

Array of S^3 entries, each holding a **ParamConfig**(p. 49) set upon construction of the array. It describes all the S^3 **(S-m)-interfaces**(p. 68) that an S-block has.

`#include <BlockEnum.h>`

Public Member Functions

- **ParamConfigEnumeration** ()

  Constructor. Fills each of the array's entries with the **ParamConfig**(p. 49) *for a certain* **(S-m)-interface**(p. 68).

- void **dump** (std::ostream &os)

  Dumps a textual descrition of the **ParamConfigEnumeration**(p. 50) *to an output stream.*

- int **entryPoint** (int k)

  This function return the first entry of the array that contains an **(S-k)-interface**(p. 68) *(an* **interface**(p. 67) *with k fixed parameters).*

7.6.1 Detailed Description

template<int S> class BlockEnum::ParamConfigEnumeration< S >

Array of S^3 entries, each holding a **ParamConfig**(p. 49) set upon construction of the array. It describes all the S^3 **(S-m)-interfaces**(p. 68) that an S-block has.

The entries are automatically set upon construction of this object so that they cover all the different **(S-m)-interfaces**(p. 68) that exists for an S-block. The first entry is the **(S-0)-interface**(p. 68) (the "interior" of the **block**(p. 66)), the $2S$ following entries represent the **(S-1)-interfaces**(p. 68), etc. The 0-**interfaces**(p. 67) arrives last in the array.

This array is used to enumerate all different **(S-m)-interfaces**(p. 68) of a **block**(p. 66), and can for instance be used to loop through all the **interfaces**(p. 67) (by looping through the entries of the array).

Definition at line 107 of file BlockEnum.h.

7.6.2 Constructor & Destructor Documentation

7.6.2.1 template<int T> BlockEnum::ParamConfigEnumeration< T >::ParamConfigEnumeration () [inline]

Constructor. Fills each of the array's entries with the **ParamConfig**(p. 49) for a certain **(S-m)-interface**(p. 68).

Together, the entries cover all the possible **interfaces**(p. 67) for an S-block.

Definition at line 84 of file BlockEnum_templates.h.

References BlockEnum::ParamState.

7.6.3 Member Function Documentation

7.6.3.1 template<int T> int BlockEnum::ParamConfigEnumeration< T >::entryPoint (int *k*) [inline]

This function return the first entry of the array that contains an **(S-k)-interface**(p. 68) (an **interface**(p. 67) with k fixed parameters).

It supposes that the entries of the array are left as they were defined by the constructor of this object, ie. that the user has not changed them in any way.

Definition at line 75 of file BlockEnum_templates.h.

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

- BlockEnum.h
- BlockEnum_templates.h

7.7 PlotableGeometry< S, D > Struct Template Reference

Represent the **geometry**(p. 66) for a fragment of a given **block**(p. 66).

`#include <BlockStructure.h>`

Public Attributes

- Go::Array< **Go::GoTensorProductSpline**< S, double >, D > **coords**

  The D splines defining the **geometry**(p. 66) *of the part of the* **Block**(p. 15) *represented with this object. There is one spline for each spatial dimension.*

- SubblockPrmDomain< S > **domain**

  Specify for which parameter domain this PlotableGeometry is valid. 'domain' is an array of S STL-pairs of `double`*, defining the minimum and maximum parameter value for the S diferent parameters.*

7.7.1 Detailed Description

template<int S, int D> struct PlotableGeometry< S, D >

Represent the **geometry**(p. 66) for a fragment of a given **block**(p. 66).

This class is only used as a component for defining the **PlotableSubblock**(p. 54).

Definition at line 60 of file BlockStructure.h.

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

- BlockStructure.h

7.8 PlotableResult< S > Struct Template Reference

Represent a **result**(p. 68) scalar field for a fragment of a given **block**(p. 66).

`#include <BlockStructure.h>`

Public Attributes

- int **result_number**

  the block's index of the **result**(p. 68) *represented by the PlotableResult object*

- **Go::GoTensorProductSpline**< S, double > **spline**

  the spline describing the **result**(p. 68) *inside the parametric domain specified by 'domain'*

- SubblockPrmDomain< S > **domain**

  Specify for which parameter domain this PlotableResult is valid. 'domain' is an array of S STL-pairs of `double`*, defining the minimum and maximum parameter value for the S different parameters.*

7.8.1 Detailed Description

template<int S> struct PlotableResult< S >

Represent a **result**(p. 68) scalar field for a fragment of a given **block**(p. 66).

This class is only used as a component for defining the **PlotableSubblock**(p. 54).

Definition at line 40 of file BlockStructure.h.

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

- BlockStructure.h

7.9 PlotableSubblock< S, D > Struct Template Reference

This class represent a "fragment" of a **block**(p. 66), ie. it can be used to evaluate a certain part of the block's **geometry**(p. 66) and (optionally) **results**(p. 68) for any value of the **spatial parameters**(p. 68) that fall within the specified *domains*.

`#include <BlockStructure.h>`

Public Attributes

- int **block_number**

  the index of the **block**(p. 66) *that this PlotableSubblock represents a part of.*

- **PlotableGeometry**< S, D > **geometry**

  Representation of (a fragment of) the **geometry**(p. 66).

- std::vector< **PlotableResult**< S > > **results**

  Representation of (fragments of) **results**(p. 68).

- Go::Array< std::pair< bool, bool >, S > **extremity_covered**

  Array with one entry per **spatial parameter**(p. 68). *It reports whether each the two* **(S-1)-interfaces**(p. 68) *specified by locking the corresponding parameter into its MIN or MAX position are covered by another* **subblock**(p. 69) *in the set, or if* this **subblock**(p. 69) *should be used to evaluate the concerned* **interface**(p. 67).

- Go::Array< std::pair< bool, bool >, S > **borders_neighbour**

  Array with one entry per **spatial parameter**(p. 68). *It reports whether each of the two* **(S-1)-interfaces**(p. 68) *specified by locking the corresponding parameter into its MIN or MAX position are shared by a neighbour (another* **block**(p. 66)*).*

7.9.1 Detailed Description

template<int S, int D> struct PlotableSubblock< S, D >

This class represent a "fragment" of a **block**(p. 66), ie. it can be used to evaluate a certain part of the block's **geometry**(p. 66) and (optionally) **results**(p. 68) for any value of the **spatial parameters**(p. 68) that fall within the specified *domains*.

The domain for the **geometry**(p. 66) and for each of the **results**(p. 68) do not necessarily have to be equal, so the user has to check individually for the **geometry**(p. 66) and for each **result**(p. 68) whether this PlotableSubblock is "evaluable" at the parameter values she has. This is rather cumbersome, and it is recommended that all handling of **block**(p. 66) evaluation is handed over to the utility functions sampleBlockGeometry() and sampleBlockResult(). A PlotableSubblock representation of one or more **blocks**(p. 66) is obtained by calling the **BlockStructure::getPlotableRep()**(p. 23) function. After the PlotableSubblock s have been obtained, the user could benefit from the above mentioned utility functions to sample the concerned **blocks**(p. 66) for *any* parametric values (the utility functions browse through a supplied list of PlotableSubblock until they find the one covering the requested parametric value).

Definition at line 91 of file BlockStructure.h.

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

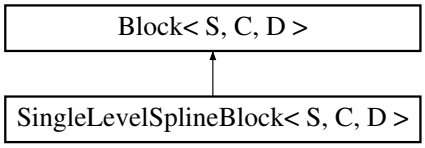
- BlockStructure.h

7.10 SingleLevelSplineBlock< S, C, D > Class Template Reference

This is a **single-level**(p. 67) **block**(p. 66) class that does not support **results**(p. 68), only **geometry**(p. 66).

`#include <SingleLevelSplineBlock.h>`

Inheritance diagram for SingleLevelSplineBlock< S, C, D >::



Public Member Functions

- **SingleLevelSplineBlock** (const double *coefs, const int *rowlengths, const int *order, bool k_reg)

  This constructor takes an array of grid values, and generates a **multivariate**(p. 67) *spline of a specified* **order**(p. 68). *The grid points is used as control points. The knotvectors will be uniform and may or may not be k-regular. The knotvector will always run from 0 to 1.*

- **SingleLevelSplineBlock** (const **SingleLevelSplineBlock** &)

  Copy constructor.

- virtual Go::BoundingBox **boundingBox** () const

  Returns a bounding box enclosing the geometric area occupied by the **block**(p. 66).

- virtual void **getCornerPosition** (const bool *max, const double *C_values, double *res) const

  Function to retrieve the position of a block's **corner**(p. 66) *in geometrical space for a given set of* **auxiliary parameters**(p. 66) *(C-parameters).*

- virtual const Go::Array< **Go::GoTensorProductSpline**< S+C, double >, D > **geometrySplines** () const

  Returns the **multivariate**(p. 67) *splines that define this block's* **geometry**(p. 66).

- virtual const **Go::GoTensorProductSpline**< S+C, double > **resultSplines** (int resultID) const

  Returns the **multivariate**(p. 67) *spline describing one of the block's* **results**(p. 68).

- Go::SplineSurface **surfInterface** (int num) const

  *This function is only specified for S=3. It returns a Go::SplineSurface representation of one of the 2-***interfaces**(p. 67) *of the cube, as defined by its* **geometry**(p. 66).

- virtual std::vector< boost::shared_ptr< Go::GeomObject > > **outline** (double *aux_par_values) const

  Returns the **outline**(p. 67) *of the* **block**(p. 66).

- virtual int **numResults** () const

  Returns the number of **results**(p. 68) *that this* **block**(p. 66) *contains.*

- virtual void **write** (std::ostream &os, bool ascii=false) const

  Write the **block**(p. 66) *to an output stream.*

- virtual void **read** (std::istream &is, bool ascii=false)

  Read the **block**(p. 66) *from an input stream.*

7.10.1 Detailed Description

template<int S, int C, int D> class SingleLevelSplineBlock< S, C, D >

This is a **single-level**(p. 67) **block**(p. 66) class that does not support **results**(p. 68), only **geometry**(p. 66).

It was mostly written for testing purposes, and also to illustrate how the user can tailor her own **block**(p. 66) type to her needs.

Definition at line 28 of file SingleLevelSplineBlock.h.

7.10.2 Constructor & Destructor Documentation

7.10.2.1 template<int S, int C, int D> SingleLevelSplineBlock< S, C, D >::SingleLevelSplineBlock (const double * *coefs*, const int * *rowlengths*, const int * *order*, bool *k_reg*) [inline]

This constructor takes an array of grid values, and generates a **multivariate**(p. 67) spline of a specified **order**(p. 68). The grid points is used as control points. The knotvectors will be uniform and may or may not be k-regular. The knotvector will always run from 0 to 1.

Note:

(Ownership issue): All the coefficients will be copied into an internal array that the SingleLevelSplineBlock has ownership of.

Parameters:

coefs pointer to an array storing all the grid's coefficients. They are stored so that each D consecutive values are the coordinates for one gridpoint. The grid is supposed to be **multiindexed**(p. 67) with S+C indexes. The first index should have the lowest **stride**(p. 68), and the last index should have the highest. This is maybe best illustrated with an **example**: let us consider the case where S=2, C=0 and D = 3. With these values for S and D, we represent a grid that is a 2-**manifold**(p. 67) (a surface) embedded in 3D space. The grid is indexed on i and j, and each gridpoint p_{ij} has the 3D-coordinates (x_{ij}, y_{ij}, z_{ij}). Let us further say that i takes values from 0 to $I-1$, and j takes values from 0 to $J-1$. The storage of the gridpoints in the array pointed to by *coefs* should be on the form:

$$x_{00}, y_{00}, z_{00}, x_{10}, y_{10}, z_{10}, ..., x_{I0}, y_{I0}, z_{I0}, x_{01}, y_{01}, z_{01}, ..., x_{IJ}, y_{IJ}, z_{IJ}$$

rowlengths pointer to an array of S+C integers, specifying the rowlengths of the grid (number of values that each index takes). In the example used above, the array pointed to should contain $[I, J]$.

order should point to an array of S+C integers, specifying the spline **order**(p. 68) for each parameter.

k_reg if this argument is set to 'true', then the generated knotvectors will be k-regular.

Definition at line 50 of file SingleLevelSplineBlock_templates.h.

7.10.3 Member Function Documentation

7.10.3.1 template<int S, int C, int D> Go::BoundingBox SingleLevelSplineBlock< S, C, D >::boundingBox () const [inline, virtual]

Returns a bounding box enclosing the geometric area occupied by the **block**(p. 66).

Returns:
the a bounding box enclosing the geometric area occupied by the **block**(p. 66)

Implements **Block< S, C, D >** (p. 16).

Definition at line 218 of file SingleLevelSplineBlock_templates.h.

References Go::GoGenericGrid< M, T >::maxElem(), and Go::GoGenericGrid< M, T >::minElem().

7.10.3.2 template<int S, int C, int D> const Go::Array< Go::GoTensorProductSpline< S+C, double >, D > SingleLevelSplineBlock< S, C, D >::geometrySplines () const [inline, virtual]

Returns the **multivariate**(p. 67) splines that define this block's **geometry**(p. 66).

There are D such splines, each expressing the position of the **block**(p. 66) in one spatial dimension, as a function of the S+C parameters. (For instance, in 3D, we would have one spline expressing the x-coordinate, another expressing the y-coordinate and a third one expressing the z-coordinate).

Returns:
A D-sized array containing the **multivariate**(p. 67) splines describing the **geometry**(p. 66) of the **block**(p. 66) in each of the spatial dimensions.

Implements **Block< S, C, D >** (p. 17).

Definition at line 268 of file SingleLevelSplineBlock_templates.h.

7.10.3.3 template<int S, int C, int D> void SingleLevelSplineBlock< S, C, D >::getCornerPosition (const bool * *max*, const double * *C_values*, double * *res*) const [inline, virtual]

Function to retrieve the position of a block's **corner**(p. 66) in geometrical space for a given set of **auxiliary parameters**(p. 66) (C-parameters).

The block's **corners**(p. 66) are defined as its **0-interfaces**(p. 67) (see **Block connectivity**(p. 2)). A **corner**(p. 66) is characterized by all the **spatial parameters**(p. 68) taking up an extremal value (either 0 or 1). The *max* argument is used to specify which **spatial parameters**(p. 68) are 0 and which are 1. It should point to an array of S bools. If *max*[i] = `true`, it means that for

the requested **corner**(p. 66), the **spatial parameter**(p. 68) i is set to 1, else it is set to 0. The spatial coordinates of the requested **corner**(p. 66) are written to the memory location pointed to by *res*. This area should of course be big enough to store D `double` s.

Parameters:

max Pointer to an array of S `bool` s, specifying whether the corresponding **spatial parameter**(p. 68) is *maximum* (1) or *minimum* (0) at this **corner**(p. 66). A value of `true` means *maximum*.

C_values Pointer to an array of C `double` s, specifying the values of the **auxiliary parameters**(p. 66) to use when evaluating the position of the **corner**(p. 66) corners.

res pointer to the memory location where the **corner**(p. 66) position (D `double` s) will be written. It is the user's responsibility to allocate enough memory.

Implements **Block< S, C, D >** (p. 17).

Definition at line 240 of file SingleLevelSplineBlock_templates.h.

7.10.3.4 template<int S, int C, int D> virtual int SingleLevelSplineBlock< S, C, D >::numResults () const [inline, virtual]

Returns the number of **results**(p. 68) that this **block**(p. 66) contains.

The **results**(p. 68) will always be refered to by using their index number 0 to **numResults()**(p. 59) - 1

Returns:

the number of **results**(p. 68) that this **block**(p. 66) contains

Implements **Block< S, C, D >** (p. 18).

Definition at line 116 of file SingleLevelSplineBlock.h.

7.10.3.5 template<int S, int C, int D> std::vector< boost::shared_ptr< Go::GeomObject > > SingleLevelSplineBlock< S, C, D >::outline (double * *aux_par_values*) const [inline, virtual]

Returns the **outline**(p. 67) of the **block**(p. 66).

The **outline**(p. 67) is defined as the union of all the block's **1-interfaces**(p. 67) (edges), which is collected, expressed as Go-objects (usually Go::SplineCurve s) and returned as the **result**(p. 68) of the function call. This function can be used for all instantiations of the **Block**(p. 15) template, provided that S > 0. The (fixed) values for the **auxiliary parameters**(p. 66) are specified by the user in an C-sized array pointed to by the argument aux_par_values.

Parameters:

aux_par_values points to an array of C *double* s, defining the values of the **auxiliary parameters**(p. 66) for which we want the **outline**(p. 67) of the **block**(p. 66).

Returns:

a vector of Go-objects (usually Go::SplineCurves) expressing the **outline**(p. 67) of the **block**(p. 66) for the specified values for the **auxiliary parameters**(p. 66).

Implements **Block< S, C, D >** (p. 19).

Definition at line 311 of file SingleLevelSplineBlock_templates.h.

7.10.3.6 template<int S, int C, int D> void SingleLevelSplineBlock< S, C, D >::read (std::istream & *is*, bool *ascii* = false) [virtual]

Read the **block**(p. 66) from an input stream.

Parameters:

is the input stream where the **block**(p. 66) is read from

ascii if the user sets this argument to *true*, then the **block**(p. 66) will be read in ASCII format, else it will be read in BINARY format.

Implements **Block< S, C, D >** (p. 19).

Definition at line 189 of file SingleLevelSplineBlock_templates.h.

7.10.3.7 template<int S, int C, int D> const Go::GoTensorProductSpline< S+C, double > SingleLevelSplineBlock< S, C, D >::resultSplines (int *resultID*) const [inline, virtual]

Returns the **multivariate**(p. 67) spline describing one of the block's **results**(p. 68).

Parameters:

resultID The index number of the requested **result**(p. 68). Valid values are from 0 to **numResults()**(p. 59) - 1.

Returns:

A **multivariate**(p. 67) spline describing the requested **result**(p. 68) scalar field

Implements **Block< S, C, D >** (p. 19).

Definition at line 333 of file SingleLevelSplineBlock_templates.h.

7.10.3.8 template<int C, int D> Go::SplineSurface SingleLevelSplineBlock< C, D >::surfInterface (int *num*) const [inline]

This function is only specified for S=3. It returns a Go::SplineSurface representation of one of the 2-**interfaces**(p. 67) of the cube, as defined by its **geometry**(p. 66).

There are six sides of the cube, numbered 0 to five. The number of the desired side is given as argument to the function, which in turn returns the Go::SplineSurface representation.

Parameters:

num choosing which 2-**interface**(p. 67) (side) of the **block**(p. 66) to generate a surface from. Valid values are:

Side 0: first parameter locked at *minimum* value (0)

Side 1: first parameter locked at *maximum* value (1)

Side 2: second parameter locked at *minimum* value (0)

Side 3: second parameter locked at *maximum* value (1)

Side 4: third parameter locked at *minimum* value (0)

Side 5: third parameter locked at *maximum* value (1)

Returns:

A Go::SplineSurface representing the desired 2-**interface**(p. 67) of the 3-block

Definition at line 276 of file SingleLevelSplineBlock_templates.h.

7.10.3.9 template<int S, int C, int D> void SingleLevelSplineBlock< S, C, D >::write (std::ostream & *os*, bool *ascii* = false) const [virtual]

Write the **block**(p. 66) to an output stream.

Parameters:

os the output stream where the **block**(p. 66) will be written

ascii if the user sets this argument to *true*, then the **block**(p. 66) will be written in ASCII format, else it will be written in BINARY format.

Implements **Block< S, C, D >** (p. 20).

Definition at line 161 of file SingleLevelSplineBlock_templates.h.

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

- SingleLevelSplineBlock.h
- SingleLevelSplineBlock_templates.h

Chapter 8

Structured Blocks Page Documentation

8.1 Model that has become cracked after heavy compression

8.2 Proofs

We consider an **S-block**(p. 68), parameterized by the unit cube in R^S by a **homeomorphism**(p. 67). We want to determine the number of **(S-m)-interfaces**(p. 68) that the **block**(p. 66) has.

An **(S-m)-interface**(p. 68) corresponds to fixing m of the S parameters to their minimum or maximum value, ie. 0 or 1. The number of ways we can select m parameters from S possible is $\binom{S}{m}$. For a given selection of m parameters, where each of them can take on one of two states (its minimum or its maximum value), the number of different configurations is 2^m. The total number of **(S-m)-interfaces**(p. 68) is therefore $2^m\binom{S}{m}$.

8.3 3D interface illustration

8.4 Glossary

- **Active level**

  When a **block**(p. 66) *contains several levels of detail for a certain* **result**(p. 68) *or* **geometry**(p. 66)*, there's always one that is considered the 'active' one, the one currently used. When calling* **Block::geometrySplines()**(p. 17) *or* **Block::resultSplines()**(p. 19)*, it will be the splines for the currently active level that are returned. See also* **Level**(p. 67)*.*

- **Auxiliary parameter**

  See **Spatial parameters**(p. 68)

- **Base level**

  The base level of a **block**(p. 66)*'s* **geometry**(p. 66) *or* **result**(p. 68) *is a representation supposed to be exact (contain all details). It from the base level that all other* **levels**(p. 67) *(approximations) are generated, and without access to the base level, such a generation is not possible.*

- **Block**(p. 15)

  A block is an entity representing a region of space with 0 or more scalar fields defined on it. The region of space occupied by the block is **homeomorph**(p. 67) *with the unit cube in* R^S*, where S is the number of* **spatial parameters**(p. 68) *of the block. This* **homeomorphism**(p. 67) *is refered to as the block's* **geometry**(p. 66)*. Moreover, each of the scalar fields defined on this region are refered to as the block's* **results**(p. 68)*. In addition to the S parameters on which the homemorphism is defined, the block's* **geometry**(p. 66) *and* **results**(p. 68) *can also depend on C* **auxiliary parameters**(p. 66)*. The base class* **Block**(p. 15) *is used to store and manipulate* **blocks**(p. 66)*.*

- **Blockstructure**

  A collection of **blocks**(p. 66) *that logically belongs together is called a blockstructure.* **Blocks**(p. 66) *in a blockstructure should obey certain rules: Their* **geometry**(p. 66) *should all be of the same dimensionality (1-*__manifolds__(p. 67)*, 2-*__manifolds__(p. 67)*, etc.) No two* **Blocks**(p. 66) *in the structure should overlap the same region of space. Moreover, the* **Blocks**(p. 66) *can share* **interfaces**(p. 67)*, but in the case they do, the* **complete interface**(p. 67) *should be shared. The class* **BlockStructure**(p. 21) *is used to store and manipulate blockstructures.*

- **Corner**

  A corner of a **block**(p. 66) *is by definition one of the* **block**(p. 66)*'s* **0-interfaces**(p. 67)*. In other words, it is a point on the* **block**(p. 66)*'s* **geometry**(p. 66) *where all* **spatial parameters**(p. 68) *takes an extremal value. The number of different corners on a* **block**(p. 66) *with S* **spatial parameters**(p. 68) *is* 2^S*.*

- **Geometric space**

  The geometric space is the D-dimensional space in which the **blocks**(p. 66)*' geometries are defined. See also* **parametric space**(p. 67)*.*

- **Geometry**

  The geometry of a **block**(p. 66) *is informally defined as the region occupied by (or represented by) the* **block**(p. 66) *in geometric space. Formally, we can refer to the geometry of a* **block**(p. 66) *as being the* **homeomorphism**(p. 67) *between* **parametric space**(p. 67) *and the* **manifold**(p. 67) *it represents in geometric space.*

- **Homeomorphism**

  Definition that comes from topology. Two geometric objects are called **homeomorphic** *if there exists a continuous, bijective function mapping points from the first object to the second. Moreover, this function must have a continuous inverse. Such a function is called a homeomorphism.*

- **Interface**

  A **block**(p. 66)*'s interfaces are those regions on the* **block**(p. 66)*'s* **manifold**(p. 67) *where one or more of the* **block**(p. 66)*'s* **spatial parameters**(p. 68) *are fixed to its minimum or maximum value. This translates as separate parts of the* **manifold's**(p. 67) **boundary**. *For a* **block**(p. 66) *with S* **spatial parameters**(p. 68)*, a region on the* **manifold**(p. 67) *where m of these parameters are fixed is called a* **(S-m)-interface**(p. 68). *Mathematically it consitutes a (S-m)-*manifold(p. 67). *By extension, we can define the* **block**(p. 66)*'s S-interface (m = 0, no fixed parameters) as being the* **interior** *of the* **manifold**(p. 67).

- **Level**

  A level is a representation of the **block**(p. 66)*'s* **geometry**(p. 66) *or one of it's* **results**(p. 68). *In a* **multilevel block**(p. 67)*, there might be several levels representing the same* **result**(p. 68) *or* **geometry**(p. 66). *Each level might differ in the amount of detail it includes in describing the* **result**(p. 68) *or* **geometry**(p. 66). *The most detailed level is called* **base level**(p. 66)*, which is supposed to be the "original" or "accurate" representation, to which all the other levels are considered "approximative levels".*

- **Manifold**

  Intuitively, the notion of manifold in this documentation is a bounded, S-dimensional volume with a certain shape, embedded in a D-dimensional **geometric space**(p. 66). *Mathematically speaking, a topological S-manfold is a separated space where every point has an open neighbourhood* **homeomorphic**(p. 67) *to an open subset of E^n (Euclidian n-space). The* **interior** *of a manifold is defined as the set of points having an open neighbourhood* **homeomorphic**(p. 67) *to E^n, and the* **boundary** *of a manifold is defined as the complement of the interior. The boundary of an S-manifold is always an (S-1)-manifold.*

- **Multilevel block**

  A **block**(p. 66) *containing more than one representation (*level(p. 67)*) for its* **results**(p. 68) *and/or its* **geometry**(p. 66).

- **Multivariate**

  " $\sim$ which involves more than one variable" . *In this documentation, we speak about* **multivariate splines** *(splines taking more than one parameter, defined as a tensor product of several univariate splines). We also use the term 'multivariate' or 'multiindexed' when speaking about grids with more than one index.*

- **Outline**

  Given an **S-block**(p. 68)*, we define it's outline as the ensemble of points lying on the block's* **1-interfaces**(p. 67). *This can be intuitively interpreted as a set of curves tracing out the "edges" of the* **manifold**(p. 67) *represented by the* **block**(p. 66).

- **Parametric space**

  The parametric space we consider here is the unit cube in R^M. Our **block**(p. 66)*s' defining splines takes are parametrized by this ensemble. For convenience, we make the following distinguishment: We decompose the space R^M into $R^S \bigotimes R^C$, where $S+C=M$ and $S \leq D$, the dimension of the* **geometric space**(p. 66) *on which our* **block**(p. 66)*'s* **geometry**(p. 66) *is defined. The S parameters taking values in R^S are called the* **block**(p. 66)*'s* **spatial parameters**(p. 68)*, and the C parameters taking values in R^C are called the* **block**(p. 66)*'s*

auxiliary parameters(p. 66). *The* **spatial parameters**(p. 68) *are supposed to be connected with the* **block**(p. 66)*'s* **geometry**(p. 66) *through an* **homeomorphism**(p. 67).

- **Plotable**

  A **plotable** *representation of a* **block**(p. 66)*'s* **geometry**(p. 66) *(or* **result**(p. 68)*) is in its simplest form the splines defining the* **geometry**(p. 66) *(or* **result**(p. 68)*), as represented by the currently* **active level**(p. 66). *However, we often want to make adjustments to the* **block**(p. 66)*s before plotting them, in order to assure continuity between two or more neighbouring* **Blocks**(p. 66). *In that case, the spline from the currently* **active level**(p. 66) *might have to be split up in several smaller splines, each covering a smaller partition of the original* **manifold**(p. 67). *In this case, the plotable representation of this* **block**(p. 66) *is given as a collection of smaller, modified splines. In this library, the above happens when the user calls* **BlockStructure::getPlotableRep()**(p. 23). *The* **BlockStructure**(p. 21) *will split up splines when necessary, and return the plotable representation as a STL-vector of* **PlotableSubblock**(p. 54).

- **Result**

  In the context of this library, a result is a scalar field that is defined on a region of space represented by the **geometry**(p. 66) *of a* **block**(p. 66). *A* **block**(p. 66) *always contain exactly one* **geometry**(p. 66) *and an arbitrary number (including 0) of results.*

- **S-block**

  A block with S **spatial parameters**(p. 68). *Must lie in a* **geometric space**(p. 66) *with dimension greater or equal to S.*

- **(S-m)-interface**

  See **Interface**(p. 67).

- **shared pointer**

  The shared pointers used in this library are the boost::shared_ptr<>. They are reference counted pointers where the object pointed to is automatically deleted when the last shared_ptr pointing to it goes out of scope.

- **Spatial parameter**

  See **parametric space**(p. 67)

- **Spline order**

  A spline is a piecewise polynomial where each polynomial piece has the same degree d. The **order** *of the spline is equal to $d + 1$.*

- **stride**

  Stride, the way we refer to that concept in this documentation, is defined as the distance (difference in memory address) between two elements in an array for two consecutive values of an index. For a single-indexed array, the values are usually stored consecutively in memory, which yields a stride of 1. For a bi-indexed array (a "matrix"), indexed by i and j, each running from 1 to respectively I and J, only one of either i or j can have a minimal stride of 1, the other typically has a stride equal to the total range of the other. For instance, the element (i, j) in the above mentioned matrix could be physically indexed in memory as $memory_{start} + j * I + i$, which yields a stride of 1 for i and a stride of I for j. Generally, a multiindexed array (usually) has stride 1 for one of its indexes, the other indexes having strides equal to the product of the ranges of those indexes with lower strides. When optimizing performance-critical programs, it is important to take the different strides of multiindexed arrays into consideration.

- **subblock**

  As the word suggest, a subblock represents a part of a **block**(p. 66). *It is only parametrized on part of the parametric domain used by it's "parent" (the* **block**(p. 66) *itself) and only covers the corresponding part of the* **manifold**(p. 67) *in* **geometric space**(p. 66). *In this library, we subdivide an* **S-block**(p. 68) *into S^3 subblocks in the following way: Each of the S* **spatial parameters**(p. 68) *are considered to have three regions, one from 0 to ξ, one between ξ and $1-\xi$, and one from $1-\xi$ to 1. ξ is here a "small number" (usually corresponding to a single knot interval in the* **block**(p. 66)*'s* **base level**(p. 66) *for the given parameter), and is in any case less than 0.5, so that none of the three domains will ever be empty. The three regions represent respectively the start, the middle and the end of the parametric domain, and in this way we can define a total of S^3 "domain fragments" that each can be represented by one subblock.*

  The reason we divide a **block**(p. 66) *into subblock is so that we can locally modify it near the* **interfaces**(p. 67) *it shares with neighbours in order to assure continuity. The division into subblocks prevents local modifications (like knot insertion or degree raising) of the defining splines from affecting the whole* **block**(p. 66).

  Not all subblocks needs to be 'declared'. A **block**(p. 66) *that does not share any of its* **interfaces**(p. 67) *with neighbours will only have declared it's "central" subblock (corresponding to the "middle region" of each parameter), but the central subblock will extend to cover the parameter domain of the whole* **block**(p. 66). *This can also happen locally; subblocks that does not need to be declared for continuity reasons can remain undeclared, and their parametric domain will be covered by "extending" the nearest declared subblock.*