PSCF+

v1.0

Generated by Doxygen 1.8.17

Contents:

1 Overview

Main Page (Up) Installation (Next)

PSCF+ is a software package for solving the polymer self-consistent field (SCF) theory in continuum. It is based on the nice GPU framework of `PSCF` (which is only for the "standard" or known as the Edwards-Helfand model, *i.e.*, incompressible melts of continuous Gaussian chains with the δ-function interactions, commonly used in polymer field theories) originally developed by Prof. David Morse and co-workers, but is improved with better numerical methods, less GPU memory usage and more flexible algorithms, and is extended to various discrete-chain models. Similar to the C++/CUDA version of PSCF, PSCF+ described here is written primarily in C++ with GPU accelerated code in CUDA.

Same as the C++/CUDA version of PSCF, PSCF+ is applicable to mixtures containing arbitrary acyclic copolymers, and preserves all of the nice features already implemented in the former, including the use of `cuFFT` on GPU, the use of Anderson mixing (which is performed on GPU) combined with a variable-cell scheme to simultaneously solve the nonlinear SCF equations and find the bulk periodicity for the ordered phases formed by block copolymer self-assembly (which speeds-up the calculation by an order of magnitude), and the extensive documentation produced by `Doxygen`. Their differences and expected advantages of the latter include:

- PSCF is only applicable to the "standard" model, while PSCF+ can also be applied to various discrete-chain models with finite-range non-bonded interactions as commonly used in molecular simulations, thus providing the mean-field reference results for such simulations; see `Models.pdf` for more details.
- For the "standard" model, PSCF+ uses the Richardson-extrapolated pseudo-spectral methods (denoted by REPS-*K* with *K*=0,1,2,3,4) to solve the modified diffusion equations (which is the crux of SCF calculations), while PSCF uses only REPS-1. A larger *K*-value gives more accurate result at larger computational cost; see `REPS.pdf` for more details.

- For 3D spatially periodic ordered phases such as those formed by block copolymer self-assembly, while PSCF uses fast Fourier transforms (FFTs) between a uniform grid in the real space and that in the reciprocal space, for Pmmm supergroup PSCF+ uses the discrete cosine transform instead of FFTs to take advantage of the (partial) symmetry of an ordered phase to reduce the number of grid points, thus both speeding up the calculations and reducing the memory usage; see *Y. Qiang and W. Li,* **Macromolecules 53,** 9943 (2020) for more details.
- In SCF calculations the (one-end-integrated) forward and backward propagators q and $q^\dagger$ of each block usually take the largest memory usage, but the GPU memory is often limited. While in PSCF the size of these propagators is MN_s, where M denotes the number of grid points in real space and N_s the number of contour discretization points on a continuous chain (or the number of segments on a discrete chain), in PSCF+ the "slice" algorithm proposed by Li and Qiang can be used to reduce the size of q to $M\sqrt{N_s}$ and that of $q^\dagger$ to just M, thus greatly reducing the GPU memory usage at the cost of computing q twice; see SavMem.pdf for more details.
- Since SCF equations are highly nonlinear, having a good initial guess is very important in practice as it determines not only which final solution (corresponding to a phase in block copolymer self-assembly) can be obtained but also how many iteration steps the solver (*e.g.*, the Anderson mixing) takes to converge these equations. PSCF+ uses automated calculation along a path (ACAP), where the converged solution at a neighboring point is taken as the initial guess at the current point in the parameter space. While this is similar to the "SWEEP" command in PSCF, the key for ACAP to be successful and efficient is that it automatically adjusts the step size along the path connecting the two points, instead of using a fixed step size as in the "SWEEP" command. In PSCF+, ACAP is further combined with the phase-boundary calculation between two specified phases, making the construction of phase diagrams in 2D very efficient. See ACAP.pdf for more details.
- The approach used by PSCF to solve the SCF equations (for an incompressible system) does not allow any athermal species in the system, which has no non-bonded interactions with all other species. This problem is solved in PSCF+; see SlvSCF.pdf for more details.

PSCF+ is free, open-source software. It is distributed under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation, either version 3 of the License or (at your option) any later version. PSCF+ is distributed without any warranty, without even the implied warranty of merchantability or fitness for a particular purpose. See the LICENSE file or the gnu web page for details.

Main Page (Up) Installation (Next)

2 Installation

Overview (Prev) User Guide (Next)

The following pages give instructions for obtaining, configuring and compiling PSCF+.

Contents:

Overview (Prev) Main Page (Up) User Guide (Next)

2.1 System Requirements

Installation (Up) Source Code (Next)

The PSCF+ package provides programs that are designed to run on a desktop, laptop or cluster with an NVIDIA GPU.

PSCF+ is distributed only as source code, and must be compiled by the user. All source code is written in ANSI 2011 C++ language standard with CUDA. Code that is written for an NVIDIA graphics processing unit (GPU).

Compilation of PSCF+ is controlled by a system of UNIX makefiles and a series of shell script.

In order to compile all of the programs in the PSCF+ package, the system on which the code is compiled must have:

- a "git" version control client
- a C++ compiler
- a python interpreter
- the GNU Scientific Library (GSL)
- An NVIDIA graphics card
- A CUDA compiler
- the CUFFT GPU-accelerated fast Fourier transform library.

A git client is needed to obtain (clone) the source code, which is maintained in a git repository on the github.com server. A python interpreter is needed during compilation (but not during execution) because the build system that compiles the PSCF source codes uses a few python scripts that are provided with the package. The GNU scientific library is used by several programs within the package for linear algebra operations. The CUFFT library, which is used extensively in SCF calculations here, is provided with recent versions of the CUDA development environment.

Installation (Up) Source Code (Next)

2.2 Source Code

System Requirements (Prev) Installation (Up) Environment Variables (Next)

The source code for PSCF+ is hosted on the `github` server, as project qwcsu/pscfplus.
The source code can be obtained by using a git version-control manager to clone the public `git repository`. The following instructions assume that a "git" client has been installed on your computer.

To obtain a working copy of the PSCF+ git repository, you should first change directory (cd) to the directory you want to contain the pscfplus/ directory. From there, then enter the command

```
> git clone --recursive https://github.com/qwcsu/pscfplus.git
```

This should create a complete working copy of the PSCF+ source code in a new subdirectory named "pscfplus/" of the directory from which you invoked the above command.

Herefafter, we assume that the root directory of the PSCF+ working copy is named pscfplus/. References to paths that do not start explicitly with a prefix "pscfplus/" should be understood to be relative paths, relative to this directory.

System Requirements (Prev) Installation (Up) Environment Variables (Next)

2.3 Environment Variables

Source Code (Prev) Installation (Up) Compilation (Next)

To compile PSCF+ in a unix environment, before compiling any code, the user should modify the following unix environment variables:

- Add the pscfplus/bin directory to the unix $PATH shell environment variable (the shell command search path). By default, executable file created by the PSCF+ build system is installed in the pscfplus/bin directory. The directory in which these files are located must be added to the users $PATH variable in order to allow the unix shell to find the executable file when it is invoked by name in a command executed from any other directory.
- Add the pscfplus/lib/python directory to the $PYTHONPATH environment variable (the python module search path). The pscfplus/scripts/python directory contains a python script that is used by the build system during compilation to generate information about dependencies among C++ files. This directory must be added to the PYTHONPATH variable in order to allow the python interpreter to find this file.

To make these changes using a bash shell, add some variant of the following lines to your the .profile or .bash_profile file in your user home directory:

```
PSCFPLUS_DIR=${HOME}/pscfplus
export PATH=${PATH}:/${PSCFPLUS_DIR}/bin
export PYTHONPATH=${PYTHONPATH}:/${PSCFPLUS_DIR}/scripts/python
```

The value of PSCFPLUS_DIR should be set to the path to the pscfpp root directory (i.e., the root of the directory tree created by cloning the pscfpp git repository). In the above fragment, as an example, it is assumed that this is a subdirectory named pscfplus within the users home directory.

After adding an appropriate variant of these lines to .profile or .bash_profile, log out, log back in, and then enter "echo $PATH" and "echo $PYTHONPATH" to make sure that these variables have been set correctly.

Source Code (Prev) Installation (Up) Compilation (Next)

2.4 Compilation

Environment Variables (Next) Installation (Up) User Guide (Next)

Below is an instructions for compiling the PSCF+ program with examples. The directions given here assume that you have cloned the PSCF+ repository and installed all required dependencies, and that the root directory of the repository is named pscfplus/.

Instructions:

- **Set environment variables:** Modify the users $PATH and $PYTHONPATH unix environment variables, as discussed here.
- **Navigate to root directory:** Change directory (cd) to the pscfplus/ root directory.

- **Setup:** Invoke the "setup" script from the pscfplus/ root directory. Enter the command
```
> ./setup
```
to setup the build system with default compiler options. Alternatively, invoke the setup with a filename argument for non-default compiler options.

- **Change directory to the build directory:** Change directory (cd) to the pscfplus/bld subdirectory, by entering "cd bld" from the root directory.

- **Compile the PSCF+ program of given model:** From pscflus/bld, enter
```
> bash compile.sh [-B CHN] [-N NBP] [-C] [-D] [-K K]
```
This will create generate a large number of intermediate object (∗.o), dependency (∗.d) and library (∗.a) files in subdirectories of the pscf+/bld directory, and install executable named "pg" in the pscfplus/bin directory. The options in the above command are as follows:
  - CHN: Specifying the model of chain connectivity (by default it is the continuous Gaussian chain)
```
DGC: discrete Gaussian chain
FJC: freely jointed chain
```
  - NBP: Specifying the form of non-bonded pair potential (by default it is the δ-function potential)
```
G: Gaussian potential
DPD: dissipative particle dynamics potential
SS: soft-sphere potential
```
  - `-C`: Specifying a compressible system (by default the system is incompressible)
  - `-D`: Specifying the use of the discrete cosine transform of type II between the real and reciprocal space (by default the fast Fourier transform is used)
  - `-K`: Specifying the *K*-value of the REPS-*K method* (by default the REPS-1 method is used)

Examples:

- **Compilation of "standard" model:** To compile the “standard” model (i.e., incompressible melts of continuous Gaussian chains with the Dirac -function repulsion) using REPS-1 method and fast Fourier transform (same as used in PSCF), users can simply use the following command:
```
bash compile.sh
```
- **Compilation of DPDC model:** To compile the DPDC model (i.e., compressible melts of discrete Gaussian chains with the dissipative particle dynamics potential) using fast Fourier transform, users can use the following command:
```
bash compile.sh -B DGC -C -N DPD
```
- To get the list of the aboved options, users can use
```
bash compile.sh -h
```

Source Code (Prev) Installation (Up) User Guide (Next)

3 User Guide

Installation (Prev) 4 Developer Information (Next)

Contents:

Installation (Prev)   Main Page (Up)   4 Developer Information (Next)

3.1 Invoking an Executable

User Guide (Prev)   Parameter Files (Next)

3.1.1 Command line usage

Here is an example of command line usage of PSCF+ program for single phase SCF calculation:
```
pg3d -e
```

We can see that the command line usage has the format
```
executable [-e]
```

In the above example, pg3d is the name of executable for 3D periodic microstructures. Similarly, pg1d and pg2d are for 1D and 2D periodic microstructures, respectively. -e activates echoing of the parameter file to standard output, which is optional.

3.1.2 Command line usage

Single phase SCF calculation requires two input files for :

- a parameter file: param
- a command file: command

under the working directory, and their names have to be "param" and "command" respectively.

Here is another example of command line usage of PSCF+ program finding a phase boundry point of two phases using Ridders' method or constructing a a phase boundry (line) of two phases via Newton's method and ACAP:
```
pg3d3d -e
```

In the above example, pg3d3d is the name of executable as well, finding a phase boundry point or constructing a a phase boundry (line) of two 3D phases. Similar, pg2d3d is to find a phase boundry point or to construct a a phase boundry (line) of 2D and 3D phases. For two phases calculations, the executables are pg1d2d, pg1d3d, pg2d3d, pg2d2d, and pg3d3d.

Two phases SCF calculation requires three input files for :

- two parameter files: param1 and param2
- a command file: command

under the working directory, and their names have to be "param1", "param2" and "command" respectively.

PSCF+ program uses similar formats for the parameter and command files. The contents and formats of these two types of file are discussed briefly below and in more detail in several separate pages (see Parameter Files , Command Files).

When the program is executed, the parameter file is read first. The parameter file is used to initialize the state of the program and allocate memory.

The command file is read and interpreted after the parameter file. The command file is a script that contains a list of commands that are interpreted and executed in sequence. This script controls the program flow after initialization.

User Guide (Up) Parameter Files (Next)

3.2 Parameter Files

Invoking an Executable (Prev) Command Files (Next)

The parameter file structure is adapted from C++/CUDA version of PSCF. In PSCF+, Parameter files for differ model systems contain somewhat different elements.

- **Parameter File Structure**

  The structure of a parameter file for a model system using continuous Gaussian chain is shown below.

```
System{
  Mixture{
    nMonomer   ...
    monomers   ...  ...
               ...  ...
    nPolymer   ...
    Polymer{
      nBlock  ...
      nVertex ...
      blocks  ...  ... ... ...  ...
              ...  ... ... ...  ...
      phi     ...
    }
    [ns]       ...
  }
  Interaction{
     chi ... ... ...
     [kappa    ...]
     sigma     ...
  }
  unitCell ...
  mesh ....
  groupName ...
  AmIterator{
     maxItr  ...
     epsilon ...
     maxHist ...
     isMinimized ...
  }
}
```

The purpose of each subblock and parameter in the main System block is as follows:

- **Mixture**: Description of molecular species and composition of the mixture.
  - **nMonomer**: Number of monomer types consisting of block copolymers
  - **monomers**: Description of monomer type. The first parameter is unique integer index for monomer type starting from 0; the second parameter specifies the statistical segment length.
  - **nPolymer**: Number of polymer species.
  - **Polymer**: Description of polymer structure, including number of blocks (nBlock), number of vertice (nVertex), and the connection of the blocks (blocks). There are five parameters used for description of block connection in continuous Gaussian chain: the fisrt parameter is unique integer index for each block starting from 0, the second and the third parameters are indice of two vertices of the block, the forth parameter is the monomer index of the block, and the last one is the length of the block. (See example below)
  - **phi**: Volume fraction of the polymer species
  - **ns**: Chain discretization used for continuous Gaussian chain. Omitted when Discrete chain is used.
- **Interaction**:
  - **chi**: Values of Flory-Huggins (χ) interaction parameters. The first two parameters are the indice monomer types, and the third one is the value of χN.
  - **kappa**: (optional) Compressibility parameter N/kappa. Omitted when system is incompressible.
  - **sigma**: The interaction range. 0 for Dirac δ-function interaction.
- **mesh**: Description of mesh used for spatial discretization, which is a list of D integer numbers with D being the dimensionality of the system.
- **groupName**: Name of the crystallographic space group .
- **AmIterator**: parameters required by the iterator.
  - **maxItr**: The maximum number of iterations.
  - **epsilon**: The criterion of convergence for SCF equations.
  - **maxHist**: The maximum size of history matrix used in Anderson mixing.
  - **isMinimized**: boolean parameter specifies whether the stress of unit cell is minimized while solving the SCF equations.

The structure of a parameter file for a model system using discrete-chain is shown below.

```
System{
  Mixture{
    nMonomer   ...
    monomers   ...  ...
               ...  ...
    nPolymer   ...
    DPolymer{
      nBond  ...
      nVertex ...
      bonds   ...  ... ... ...  ...  ...
              ...  ... ... ...  ...  ...
      phi     ...
    }
    ns         ...
  }
  Interaction{
     chi ... ... ...
     [kappa    ...]
     sigma     ...
  }
  unitCell ...
  mesh ....
  groupName ...
  AmIterator{
     maxItr  ...
     epsilon ...
     maxHist ...
```

```
    isMinimized ...
  }
}
```

There are two differences from system model using continuous Gaussian chain:

- **DPolymer**: Description of polymer structure, including number of bonds (nBond), number of vertice (nVertex), and the connection of the bonds (bonds). There are six parameters used for description of block connection in discrete chain: the fisrt parameter is unique integer index for each bond starting from 0, the second and the third parameters are indice of two vertices of the bond, the forth and fifth parameters are the monomer indices of the bond on the two vertices, and the last one is the number of segments on the bond. (See example below)
- ns is no longer needed for discrete chain model.

- ***Example for "Standard" model***

```
System{
  Mixture{
    nMonomer  2
    monomers   0   A   1.0
               1   B   1.0
    nPolymer  1
    Polymer{
      nBlock  2
      nVertex 3
      blocks  0  0  0  1  2.50000000E-01
              1  1  1  2  7.50000000E-01
      phi     1.0
    }
    ns        128
  }
  Interaction{
    chi  1   0   20.0
    sigma 0.0
  }
  unitCell cubic      4.6662857614e+00
  mesh      64   64  64
  groupName         I_m_-3_m
  AmIterator{
    maxItr 5000
    epsilon 1e-9
    maxHist 20
    isMinimized 1
  }
}
```

- ***Example for DPDC model***

```
System{
  Mixture{
    nMonomer  2
    monomers   0   A   2.0
               1   B   1.0
    nPolymer  1
    DPolymer{
      nBond  3
      nVertex 4
      bonds   0  0  1  0  0   2
              1  2  3  1  1   8
              2  1  2  0  1   0
      phi     1.0
    }
  }
  Interaction{
    chi  1   0   40
    kappa 157.07963267948966
    sigma 0.89442719099992
  }
  mesh     64    64   64
  groupName         I_m_-3_m
  AmIterator{
    maxItr 5000
    epsilon 1e-8
    maxHist 20
    isMinimized 0
  }
}
```

Invoking an Executable (Prev) User Guide (Up) Command Files (Next)

3.3 Command Files

Parameter Files (Prev) Field Files (Next)

The command file contains a sequence of commands that are read and executed in sequence. Each command begins with an all upper case label. Some commands take one or more parameters, which must appear after the upper case label, separated by whitespace. The program stops execution and returns control to the operating system when it encounters the command string 'FINISH' on a line by itself.

Read an omega field in basis format named omega.bf under the directory out/:

```
READ_W_BASIS            out/omega.bf
```

Read an omega field in real-space grid format named omega.rf under the directory out/:

```
READ_W_RGRID            out/omega.rf
```

Read an omega field in fourier-space grid format named omega.kf under the directory out/:

```
READ_W_KGRID            out/omega.kf
```

Write an omega field in basis format named omega.bf under the directory out/:

```
WRITE_W_BASIS           out/omega.bf
```

Write an omega field in real-space grid format named omega.rf under the directory out/:

```
WRITE_W_RGRID           out/omega.rf
```

Write an omega field in fourier-space grid format named omega.kf under the directory out/:

```
WRITE_W_KGRID           out/omega.kf
```

Read a volume fraction field in basis format named omega.bf under the directory out/:

```
READ_C_BASIS            out/phi.bf
```

Read a volume fraction field in real-space grid format named omega.rf under the directory out/:

```
READ_C_RGRID            out/phi.rf
```

Read a volume fraction field in fourier-space grid format named omega.kf under the directory out/:

```
READ_C_KGRID            out/phi.kf
```

Write a volume fraction field in basis format named omega.bf under the directory out/:

```
WRITE_C_BASIS           out/phi.bf
```

Write a volume fraction field in real-space grid format named omega.rf under the directory out/:

```
WRITE_C_RGRID           out/phi.rf
```

Write a volume fraction field in fourier-space grid format named omega.kf under the directory out/:

```
WRITE_C_KGRID           out/phi.kf
```

Solve SCF equations iteratively with given initial guess (omege field should be read before):

```
ITERATE
```

Automated calculation along a path changing χN value of monomer 0 and 1 from χN=20 to χN=40 with initial stepsize 0.2, largest stepsize 2.0, smallest stepsize 0.01, and stepsize scaling factor 1.1. The output will be logged in the file named log under the dicrectory out/:

```
CHI_PATH                20  30  0  1  0.2  2.0  0.01  1.1  out/log
```

Automated calculation along a path changing statistical segment length b of monomer 0 from b=1.0 to b=3.0 with initial stepsize 0.05, largest stepsize 0.2, smallest stepsize 0.01, and stepsize scaling factor 1.05. The output will be logged in the file named log under the dicrectory out/:

```
b_PATH                 1.0  3.0  0  0.05  0.2  0.01  1.05  out/log
```

Automated calculation along a path changing statistical segment length b of monomer 0 from b=1.0 to b=3.0 with initial stepsize 0.05, largest stepsize 0.2, smallest stepsize 0.01, and stepsize scaling factor 1.05. The output will be logged in the file named log under the dicrectory out/:

```
b_PATH                 1.0  3.0  0  0.05  0.2  0.01  1.05  out/log
```

Given χN=25, find the bA value (statistical segment length of monomer 0) that the BCC sphere phase has the same Helmholtz free energy with FCC where the difference bewteen these two phases is smaller than 10E-3. bcc/out/omega.bf and fcc/out/omega.bf are two initial guesses for the BCC and FCC phases, respectively. 1.25 is the initial guess of the bA value for the boundry point. The new omega fields of these two phases are written to bcc/out/omega_new.bf and fcc/out/omega_new.bf respectively. The input and output files are for omega fields in basis format only so far.

```
chi_b                25.0  0  1.25  10E-3  bcc/out/omega.bf  fcc/out/omega.bf  bcc/out/omega_new.bf
       fcc/out/omega_new.bf
```

Given statistical segment length of monomer 0 bA=1.25, find the χN value of monomer 0 and monomer 1 that the BCC sphere phase has the same Helmholtz free energy with FCC where the difference bewteen these two phases is smaller than 10E-3. bcc/out/omega.bf and fcc/out/omega.bf are two initial guesses for the BCC and FCC phases, respectively.25 is the initial guess of the χN value for the boundry point. The new omega fields of these two phases are written to bcc/out/omega_new.bf and fcc/out/omega_new.bf respectively. The input and output files are for omega fields in basis format only so far.

```
b_chi                1.25  0  1  25  10E-3  bcc/out/omega.bf  fcc/out/omega.bf bcc/out/omega_new.bf
       fcc/out/omega_new.bf
```

Automated calculation along a path changing statistical segment length b of monomer 0 from b=1.25 to b=3.0 and finding the corresponding χN values of monomer 0 and 1 that the BCC sphere phase has the same Helmholtz free energy with FCC where the difference bewteen these two phases is smaller than 10E-3. bcc/out/omega.bf and fcc/out/omega.bf are two initial guesses for the BCC and FCC phases, respectively. 0, 0.05, 0.2, 0.01, and 1.05 are initial stepsize, largest stepsize, smallest stepsize, and stepsize scaling factor, respectively. The output will be logged in the file named log under the dicrectory out/ 1.25 is the initial guess of the bA value for the first boundry point. The new omega fields of these two phases are written to bcc/out/omega_new.bf and fcc/out/omega_new.bf respectively. The input and output files are for omega fields in basis format only so far.

```
b_chi_curve          1.25  0  1  25  10E-3  3.0  0  0.05  0.2  0.01  1.05  out/log  bcc/out/omega.bf
       fcc/out/omega.bf bcc/out/omega_new.bf  fcc/out/omega_new.bf
```

Parameter Files (Prev) User Guide (Up) Field Files (Next)

3.4 Field Files

Command Files (Prev) 4 Developer Information (Next)

The files of omega field and volume fraction field in PSCF+ share the same format with C++/CUDA version of PSCF. please see `https://dmorse.github.io/pscfpp-man/` for details.

Command Files (Prev) User Guide (Up) 4 Developer Information (Next)

4 Developer Information

User Guide (Prev)

Contents:

User Guide (Prev) Main Page (Up)

4.1 4.1 Directory Structure

4 Developer Information (Prev) 4.2 Coding Standards (Next)

All source files of PSCF+ are in the pscfplus/src/ directory tree. The header and source file for each class are in the same directory. The name of each C++ file is the same as the class name, followed by an extension to indicate file type. We use extension .h to indicate a header file, .tpp to indicate the implementation of a class template, .cpp to indicate a C++ source file, and .cu to indicate a CUDA source file. All class names and corresponding file names are upper space camel (like Util::Vector or Pscf::Basis).
The source code in pscfpp/src is divided among two top-level namespaces, named Util and Pscf.
The Util namespace contains a collection of utilities for scientific computation that is also used by other projects. All code in the Util namespace is contained in the src/util directory. This directory contains the contents of a separate github git repository (repository dmorse/util) that is imported into the pscfpp as a submodule.
The Pscf namespace contains all C++ and CUDA code that is specific to the PSCF+ project. The Pscf namespace contains several enclosed namespaces that each contain code that is used only by one program or set of closely related programs.
The main subdirectories of src/ are:

- src/util/ contains code of utilities for scientific computation that is also used by other projects.
- src/pscf/ contains basic classes in the Pscf namespace for polymer self-consistent field calculations, which is accessible to all PSCF+ programs.
- src/pspg/ contains CUDA code of utilities in the Pscf::Pspg namespace for scientific computation, including encapsulation of fast fourier transform, fast cosine transform, and input/output of omega/phi fields storing in GPU RAM.
- src/pgc/ contains implementation of model system using continuous Gaussian chain model in the Pscf::Pspg::Continuous namespace
- src/pgd/ contains implementation of model system using discrete chain model, including discrete Gaussian and freely-jointed chain, in the Pscf::Pspg::Discrete.

4 Developer Information (Prev) Main Page (Up) 4.2 Coding Standards (Next)

4.2 4.2 Coding Standards

4.1 Directory Structure (Prev) build_page (Next)

- Naming Conventions:

CamelCase: Use camelCase for variable and function names. Start with a lowercase letter, and capitalize the first letter of each subsequent word within the name. For example: myVariable, calculateTotalCost(). PascalCase: Use PascalCase for class names. Start with an uppercase letter, and capitalize the first letter of each subsequent word within the name. For example: MyClass. UPPER_CASE_WITH_UNDERSCORES: Use uppercase letters and underscores to name constants and macros. For example: PI, MAX_VALUE. Namespace Names: Namespace names should follow the same rules as other identifiers. Typically, they are in lowercase, and underscores can be used to separate words. For example: my_namespace.

- File Names: File names should generally be in lowercase and use underscores to separate words. For example: my_file_name.cpp.
- Formatting: The code of PSCF+ are reformatted using Prettier, which a free plug-in supported by many editors. Here is the configuration file for Prettier:

```
{
  "printWidth": 80,
  "tabWidth": 4,
  "useTabs": false,
  "semi": true,
  "singleQuote": false,
```

```
  "trailingComma": "none",
  "bracketSpacing": true,
  "endOfLine": "lf"
}
- printWidth: Sets the maximum line length.
- tabWidth: The number of spaces per tab.
- useTabs: Whether to use tabs or spaces for indentation.
- semi: Whether to add semicolons at the end of statements.
- singleQuote: Whether to prefer single quotes over double quotes.
- trailingComma: Whether to add a trailing comma in multi-line arrays or objects.
- bracketSpacing: Controls whether spaces are added inside object braces.
- endOfLine: Defines the line ending format, useful for maintaining consistency across different OSes
```

Please see `Prettier documentation` for details.

4.1 Directory Structure (Prev) Main Page (Up) build_page (Next)

5 Module Index

5.1 Modules

Here is a list of all modules:

6 Namespace Index

6.1 Namespace List

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

7 Hierarchical Index

7.1 Class Hierarchy

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

8 Class Index

8.1 Class List

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

9 File Index

9.1 File List

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

10 Module Documentation

10.1 Programs

10.1.0.1 Programs

- pgc1d_page
- pgc2d_page
- pgc3d_page
- pgd1d_page
- pgd2d_page
- pgd3d_page

10.2 Accumulators

Statistical operations (e.g., averages or autocorrelations) on a data sequence.

Classes

- class Util::AutoCorr< Data, Product >

    Auto-correlation function for one sequence of Data values.
- class Util::AutoCorrArray< Data, Product >

    Auto-correlation function for an ensemble of sequences.
- class Util::AutoCorrelation< Data, Product >

    Auto-correlation function, using hierarchical algorithm.
- class Util::AutoCorrStage< Data, Product >

    Hierarchical auto-correlation function algorithm.
- class Util::Average

    Calculates the average and variance of a sampled property.
- class Util::AverageStage

    Evaluate average with hierarchical blocking error analysis.
- class Util::Distribution

    A distribution (or histogram) of values for a real variable.
- class Util::IntDistribution

    A distribution (or histogram) of values for an int variable.
- class Util::MeanSqDispArray< Data >

    Mean-squared displacement (MSD) vs.
- class Util::RadialDistribution

    Distribution (or histogram) of values for particle separations.
- class Util::SymmTensorAverage

    Calculates averages of all components of a Tensor-valued variable.
- class Util::TensorAverage

    Calculates averages of all components of a Tensor-valued variable.

10.2.1 Detailed Description

Statistical operations (e.g., averages or autocorrelations) on a data sequence.

10.3 Serialization

Classes

- class Util::BinaryFileIArchive

  Saving archive for binary istream.
- class Util::BinaryFileOArchive

  Saving / output archive for binary ostream.
- class Util::MemoryIArchive

  Input archive for packed heterogeneous binary data.
- class Util::MemoryOArchive

  Save archive for packed heterogeneous binary data.
- class Util::Serializable

  Abstract class for serializable objects.
- class Util::TextFileIArchive

  Loading archive for text istream.
- class Util::TextFileOArchive

  Saving archive for character based ostream.
- class Util::XdrFileIArchive

  Loading / input archive for binary XDR file.
- class Util::XdrFileOArchive

  Saving / output archive for binary XDR file.

Functions

- template<class Archive , typename T >
  void Util::serialize (Archive &ar, T &data, const unsigned int version)

  Serialize one object of type T.
- template<class Archive , typename T >
  void Util::serializeEnum (Archive &ar, T &data, const unsigned int version=0)

  Serialize an enumeration value.
- template<class Archive , typename T >
  void Util::serializeCheck (Archive &ar, T &data, const char *label="")

  Save a value, or save and check correctness on loading.

10.3.1 Detailed Description

Serialization of C++ objects to/from file or memory.
The code in this module provides a system for serializing sequences of C++ objects to a file or to random access memory. A serialization of an object stores the full internal state and allows the object to be reconstructed. The design is loosely based on that of the Boost serialization library, http://www.boost.org/doc/libs/1_48_0/libs/serialization/doc/index.html but is much simpler (and less powerful) than the Boost library.

10.3.2 Archives

An archive stores serialized data, either in a file or in RAM. The definition of an archive used here is very similar to that used in the Boost serialization library. An archive class may model either a saving / output archive, to which data is saved, or a loading / input archive, from which data is loaded. By convention, the names of saving/output archive classes end with the string OArchive and the names of loading/input archive classes end with the string IArchive.
Different archive classes store serialized objects in different forms. For example, TextFileOArchive and TextFileIArchive are saving and loading archive classes, respectively, that are wrappers for ofstream or ifstream file stream objects in which data is stored in a character representation. BinaryFileOArchive and BinaryFileIArchive are saving/output and loading / input archives that store data in a binary format. MemoryOArchive and MemoryIArchive are saving and loading archives that stored data in binary form in a block of random-access memory.

10.3.3 Overloaded IO operators

Objects may be saved to a saving archive or loaded from a loading archive using overloaded operators, using the same syntax as that of the Boost library. Each saving archive class must define method templates that overload the $<<$ (insertion) and & operators. These overloaded operators must be equivalent, and must save an object to the archive. If ar is an instance of a saving archive, such as BinaryFileOArchive, the expressions
```
ar « data;
ar &  data;
```
are thus equivalent, and both save the state of variable data into archive ar. Each loading archive class must instead define template methods to overload the $>>$ (extractor) and & operator, which must be equivalent, and which must load an object from the archive. If ar is an instance of a loading archive, such as BinaryFileIArchive, then the expressions
```
ar » data;
ar &  data;
```
are equivalent, and both load the state of variable data from archive ar.

10.3.4 Serialize Functions

Objects of type T can be saved to or loaded from an instance of a class Archive if and only if the compiler can find a function named serialize with the signature
```
void serialize(Archive& ar, T& data, unsigned int version)
```
Here, "version" is an integer index that indicates the version of the archive. This version id is normally given by an integer member of the archive class. The operator & for a class Archive is normally implemented by a method template
```
template <typename T>
void Archive::operator & (T& data);
{  serialize(*this, data, version_); }
```
that simply calls the appropiate serialize method. Here, version_ is an integer member of the Archive class that stores the archive version id. Similar templates must be provided for the $<<$ or $>>$ operator.

Each archive class provides serialize functions for all of the built-in C/C++ types, as well as few other common data types such as std::string. Definitions of the serialize function for saving archive types must save (write) data, and those for loading archive types must load (read) data.

Instances of user-defined classes may also be serialized if an appropriate serialize function can be found by the compiler. Serialization of instances of a class T may be enabled by defining either:

- A global serialize function template, with a signature
```
template <class Archive>
inline void serialize(Archive& ar, T& data, const unsigned int version);
```
- A serialize method template in class T, with a signature
```
template <class Archive>
void T::serialize(Archive& ar, const unsigned int version);
```
  Note that, in either case, the archive type is normally a template parameter, so that the same serialize function can work with multiple types of archives.

In order to use this system, it is worth understanding how the compiler finds an appropriate serialize method. When the C++ compiler needs a serialize method for a particular archive type Archive and data type T, it will look first for a function serialize(Archive&, T&, unsigned int) with exactly the required signature, and then for an appropriate template. Such functions are provided for each archive classes for all of the built-in C/C++ types, and are always used to serialize such types. For class types, their is normally no such non-template function, and so the compiler will look for an appropriate template, giving priority to templates in which fewer of the function parameters have types given by template arguments, rather than explicit types. If the compiler has access to a global serialize function template for class T with the signature described above, in which the archive type is a template parameter but the data type T is explicit, it will use this. If no such global serialize function template is found, the compiler will try to compile the following generic template,
```
template <class Archive, typename T>
inline void serialize(Archive& ar, T& data, const unsigned int version)
{  data.serialize(ar, version); }
```
which is defined in the file src/util/serialize.h. This template simply calls the serialize method of class T, and so will not compile if no such method exists. The compiler can thus use, in decreasing order of priority: 1) An explicit serialize function for type T and a specific archive type, 2) A serialize function template for a specific type T in which the archive type is a template parameter, or 3) A serialize method of class T in which the archive type is a template parameter. If none of these are accessible for class T, compilation will fail for any code that attempts to serialize an instance of class T.

The use of a single operator & to represent both output (when applied to a saving archive) and input (when applied to a loading archive), makes it possible to write a single serialize function template for each class that specifies how to order save or load instances of that class, by specifying the order in which members of the class are serialized. For example, consider the following definition of a simple complex number class:

```
class Complex  {
public:
   A(double real, double imag) : real_(real), imag_(imag) {}
   template <class Archive>
   void serialize(Archive& ar, unsigned int version)
   {
      ar & real_;
      ar & imag_:
   }
private:
   double real_;
   double imag_;
}
```

The serialize method template provides instructions for the order in which to either save the two floating point members of the class to a saving archive, or to load them from a loading archive. The use of a template in which the archive type is a parameter allows a single serialize method to be used with any type of saving or loading archive.
The most serious disadvantage of this system is that, if the serialize method is defined by a template, it cannot also be a virtual method. As a result, the serialize method template for a class cannot be accessed polymorphically, via a pointer or reference to a base class. This limitation becomes a problem in designs in which some objects are accessed only via base class pointers. The Serializable abstract base class, discussed below, partially solves this problem, by replacing the serialize method template by a pair of virtual save() and load() methods.

10.3.5 Serializable Classes

Serializable is an abstract base class that provides an alternate interface for serializing objects, using virtual functions rather than method templates. Each subclass of Serializable must define virtual save() and load() methods with the following signatures:

```
virtual void save(Serializable::OArchive& ar);
virtual void load(Serializable::IArchive& ar);
```

The typenames Serializable::OArchive and Serializable::IArchive are typedefs that define a pair of archive classes to be used for serialization.
The advantage of using virtual functions is that it allows these methods to be accessed polymorphically, via base class pointers or references. The disadvantage is that it requires the hard-coding of a single type type of saving and loading archive. To retain some flexibility, these saving and loading types are defined in the Serializable class by a pair of typedefs. This allows the type of archives used with Serializable objects to be changed throughout the code by changing these two typedefs and recompiling.
In practice, a serialize method or function template should be defined for relatively simple, non-polymorphic classes, but polymorhpic classes that are normally accessed via base class pointers need to be derived from Serializable, and must implement save and load methods.

10.3.6 Function Documentation

10.3.6.1 serialize() `template<class Archive , typename T >`

```
void Util::serialize (
            Archive & ar,
            T & data,
            const unsigned int version )  [inline]
```

Serialize one object of type T.
Default implementation calls serialize method of data object. Can be overridden by any explicit specialization.

Parameters

ar	archive object

Parameters

data	object to be serialized
version	archive version id

Definition at line 29 of file serialize.h.
Referenced by Util::MemoryOArchive::operator&(), and Util::MemoryOArchive::operator<<().

10.3.6.2 serializeEnum() `template<class Archive , typename T >`

```
void Util::serializeEnum (
            Archive & ar,
            T & data,
            const unsigned int version = 0 )  [inline]
```

Serialize an enumeration value.

Parameters

ar	archive object
data	object to be serialized
version	archive version id

Definition at line 42 of file serialize.h.
Referenced by Pscf::serialize().

10.3.6.3 serializeCheck() `template<class Archive , typename T >`

```
void Util::serializeCheck (
            Archive & ar,
            T & data,
            const char * label = "" )  [inline]
```

Save a value, or save and check correctness on loading.

Parameters

ar	archive object
data	object to be serialized
label	label C-string for object.

Definition at line 64 of file serialize.h.
References UTIL_THROW.

10.4 Container Templates

Modules

- Object Arrays

  Array containers that store objects by value, and related iterators.
- Pointer Arrays

  Array containers that store pointers to objects, and related iterators.
- Matrix Containers

  Two-dimensional array containers that store by objects value.
- Linked List

  A simple linked list implementation and associated iterator.
- Iterators

  Iterators for use with particular containers.

10.4.1 Detailed Description

Container and iterator class templates.
This module contains a set of simple container templates, some of which are similar to containers provided by the C++ standard library. Bounds checking of indices for all array containers can be turned on (for safety) or off (for speed) by defining or not defining the UTIL_DEBUG preprocessor macro.

10.4.2 Array and Matrix Containers

Containers templates whose name contains the string 'Array' are one dimensional array containers, much like C arrays. All such containers overload the subscript [] operator so as to return an object by reference, using the same syntax as a C array or a std::vector: If A is an array, then A[i] is a reference to the ith element of A.
Container templates whose name contains the string 'Matrix' are two dimensional arrays. These overload the (int, int) operator to access elements: If M is a Matrix, then M(i, j) is a reference to the element in column j of row i of A.

10.4.3 Container Name Prefixes

The names of many containers have prefixes before the word Array or Matrix that indicates policies for memory allocation and management.
Containers templates whose name begins with the letter 'D' (such as DArray, DSArray, DPArray, and DMatrix) use dynamically allocated memory. The declaration "DArray<int> A" declares a dynamically allocated array of integers. Memory must be explicitly allocated for these containers by calling the "allocate" method after the container is instantiated and before it is used. Dynamically allocated containers can only be allocated once and are not resizable. Attempting to allocate a container more than once is as an error, and causes an Exception to be thrown.
Containers templates whose name begins with a letter 'F' (such as FArray, FSArray, FPArray, and FMatrix) are fixed size containers. The capacity of each such container is determined at compile time by a template parameter or parameters. Thus, for example,

```
FArray<int, 4> A;
```

declares a fixed size array of four integers, much like the declaration "int V[4]" of a fixed size C array.
The letter "S" in the names of DSArray and FSArray indicate that these are "sized" arrays. These arrays have a variable logical size that is less than or equal to the physical capacity. The logical size is the current number of elements, which are always stored contiguously from index 0 to index size - 1. Accessing an element with index greater than or equal to size is an error, and will cause an Exception to be thrown if debugging is enabled. The capacity of an array is the number of elements for which memory has been allocated. The size of such an array is initially set to zero, and elements are added sequentially by the append() method, which adds a new element at the end of the array and increments the size counter. Once the size reaches the array capacity, attempting to append another element will cause an Exception to be thrown.
Array containers whose name includes the prefix G are sized arrays with a capacity that can grow (G="growable") as needed as elements are appended. The "GArray" template thus implements a dynamic array very similiar to the

standard library std::vector. Automatic resizing changes the address of the beginning of the array, and invalidates all iterators and pointers to elements.

10.4.4 Pointer Arrays

Container templates whose name contains the prefix "P" are pointer arrays. A pointer array is a container that stores pointers to objects that are instantiated outside of the array, rather than storing actual objects. The containers DPArray and FPArray are dynamically allocated fixed size pointer arrays, respectively. The GPArray array is a growable pointer array, which can grow without bound. The pointer arrays are all similar to "sized" arrays in that they have a logical size that must be less than or equal to their capacity, and in that elements must can be added to the end of an initially empty array by function "append(T&)". Pointer arrays use the same interface for the subscript operator (which returns a reference) and the append function (which takes a reference parameter) as that used by the sized and object arrays. A pointer array of type DPArray$<$ T $>$ is thus different from a sized array of pointers, of type DSArray$<$T$*>$, because DPArray$<$ T $>$ overloads the [] operator to return a reference to an object of type T that is referenced by a private pointer, whereas the subscript operator for a DSArray$<$T$*>$ returns an actual pointer.

10.5 Object Arrays

Array containers that store objects by value, and related iterators.

Classes

- class Util::Array< Data >

    Array container class template.
- class Util::ArrayIterator< Data >

    Forward iterator for an Array or a C array.
- class Util::ConstArrayIterator< Data >

    Forward const iterator for an Array or a C array.
- class Util::DArray< Data >

    Dynamically allocatable contiguous array template.
- class Util::DSArray< Data >

    Dynamically allocated array with variable logical size.
- class Util::FArray< Data, Capacity >

    A fixed size (static) contiguous array template.
- class Util::FSArray< Data, Capacity >

    A fixed capacity (static) contiguous array with a variable logical size.
- class Util::GArray< Data >

    An automatically growable array, analogous to a std::vector.
- class Util::GridArray< Data >

    Multi-dimensional array with the dimensionality of space.
- class Util::Pair< Data >

    An array of exactly 2 objects.
- class Util::RArray< Data >

    An Array that acts as a reference to another Array or C array.
- class Util::RingBuffer< Data >

    Class for storing history of previous values in an array.

10.5.1 Detailed Description

Array containers that store objects by value, and related iterators.
The Array containers that do not have a P prefix are one-dimensional array containers that store objects by value. These all overload the subscript [] operator to provide access to elements as references.
The DArray and FArray containers are simple wrappers for dynamically allocated and fixed-size C arrays, respectively. The DSArray and FSArray containers are dynamically and statically allocated arrays, respectively, that have both a fixed capacity but a variable logical size, with contiguous elements. The GArray container is a growable sized array, similar to a std::vector. Destructors for these arrays all delete the associated C array of objects.
An RArray $<$ T $>$ is an Array that is intended to be used as an alias for, or a shallow copy of, a target DArray, FArray or C array. An RArray contains a copy of the array address and capacity of the target array, where are copied by the RArray::associate() method. Like other array containers, an RArray overloads the [] operator to provide access to elements as references. The destructor of an RArray does not delete the associated C array.
A RingBuffer is a cylic buffer array for which the append() method adds elements to the end of a sequence if the buffer is not full, and overwrites the oldest element if it is.

10.6 Pointer Arrays

Array containers that store pointers to objects, and related iterators.

Classes

- class Util::ArraySet< Data >

  A container for pointers to a subset of elements of an associated array.
- class Util::ArrayStack< Data >

  A stack of fixed capacity.
- class Util::ConstPArrayIterator< Data >

  Forward iterator for a PArray.
- class Util::DPArray< Data >

  A dynamic array that only holds pointers to its elements.
- class Util::FPArray< Data, Capacity >

  Statically allocated pointer array.
- class Util::GPArray< Data >

  An automatically growable PArray.
- class Util::GStack< Data >

  An automatically growable Stack.
- class Util::PArray< Data >

  An array that only holds pointers to its elements.
- class Util::PArrayIterator< Data >

  Forward iterator for a PArray.
- class Util::SSet< Data, Capacity >

  Statically allocated array of pointers to an unordered set.

10.6.1 Detailed Description

Array containers that store pointers to objects, and related iterators.
The one-dimensional array with names that contain a prefix "P" all store pointers to objects. This module also includes associated iterators.
The DPArray and FPArray class templates are dynamically and statically allocated pointer arrays, respectively. A GPArray is a growable pointer array.
Each DPArray < T >, FArray <T, N>, or GPArray<T> container has a private C array of T∗ pointers. These containers all overload the the [] operator so as to return a T& reference, rather than a T∗ pointer. The append method takes a T& reference as a parameter. The destructor for a pointer array deletes the underlying array of T∗ pointers, but not the T objects to which they point.
An ArrayStack < T > container is a finite capacity stack that is implemented as a dynamically allocated array of T∗ pointers. Objects can be pushed onto or popped off the top of the stack using the push(T&) and pop() methods. An ArrayStack can be allocated only once, and cannot be resized.
An SSet< T > is a container that holds pointers to an unordered set of T objects. It provides fast addition and removal of single elements, and fast iteration through all elements. The indexing of elements is arbitrary and mutable, and may change when an element is deleted.
An ArraySet < T > is a container that holds pointers to a subset of the elements of an associated array of T objects. The indexing of elements within an ArraySet container is arbitrary and mutable.

10.7 Matrix Containers

Two-dimensional array containers that store by objects value.

Classes

- class Util::DMatrix< Data >

    Dynamically allocated Matrix.
- class Util::DRaggedMatrix< Data >

    Dynamically allocated RaggedMatrix.
- class Util::FMatrix< Data, M, N >

    Fixed Size Matrix.
- class Util::Matrix< Data >

    Two-dimensional array container template (abstract).
- class Util::RaggedMatrix< Data >

    A 2D array in which different rows can have different lengths.

10.7.1 Detailed Description

Two-dimensional array containers that store by objects value.
Matrix containers overload the () operator to return elements by refernce. If A is a matrix object A(i, j) returns element i, j of the matrix.
The Matrix base class defines a conventional matrix, in which all rows are the same length. The DMatrix and FMatrix subclasses use dynamically allocated and fixed-size arrays, respectively.
The RaggedMatrix base class and DRaggedMatrix subclass define two-dimensional containers in which different rows can have different lengths, though the list of row dimensions can be specified only once.

10.8 Linked List

A simple linked list implementation and associated iterator.

Classes

- class Util::List< Data >

  Linked list class template.
- class Util::ListArray< Data >

  An array of objects that are accessible by one or more linked List objects.
- class Util::Node< Data >

  Linked List Node, class template.

10.8.1 Detailed Description

A simple linked list implementation and associated iterator.

10.9 Iterators

Iterators for use with particular containers.

Classes

- class Util::ArrayIterator< Data >

    Forward iterator for an Array or a C array.
- class Util::ConstArrayIterator< Data >

    Forward const iterator for an Array or a C array.
- class Util::ConstPArrayIterator< Data >

    Forward iterator for a PArray.
- class Util::ListIterator< Data >

    Bidirectional iterator for a List.
- class Util::PArrayIterator< Data >

    Forward iterator for a PArray.

10.9.1 Detailed Description

Iterators for use with particular containers.

10.10 Output Format

Utilities to simplify formatted C++ stream output.

Classes

- class Util::Bool

    Wrapper for an bool value, for formatted ostream output.
- class Util::Dbl

    Wrapper for a double precision number, for formatted ostream output.
- class Util::Format

    Base class for output wrappers for formatted C++ ostream output.
- class Util::Int

    Wrapper for an int, for formatted ostream output.
- class Util::Lng

    Wrapper for a long int, for formatted ostream output.
- class Util::Str

    Wrapper for a std::string, for formatted ostream output.

Functions

- template<typename Type >
  void Util::write (std::ostream &out, Type data)

    Function template for output in a standard format.

10.10.1 Detailed Description

Utilities to simplify formatted C++ stream output.
This module provides wrapper classes that can simplify formatted output of the primitive data types with controllable field width and floating point precision.

10.10.2 Classes

The classes Int, Lng, Dbl, Bool, and Str are wrappers for outputting the data types int, long double, bool, and std::string, respectively. An inserter (<<) operator is defined for each such wrapper class that produces formatted output of the enclosed data with a controllable field width and (for Dbl) precision. Each wrapper class has a member variable of the associated data type and an integer field width member. The Dbl class also has an integer precision member, to control floating point precision.
Example: We wish to output the elements of two double precision precision array named "A" and "B" in two column with a minimum field width of 20 characters for elements of A, with 10 digit precision, and 10 characters for elements of B, with 6 digit precision. The following code accomplishes this:

```
double A[10], B[10];
// ... code that assigns values to elements of A and B ...
for (int i=0; i< 10; ++i) {
   std::cout « Dbl(A[i], 20, 10) « Dbl(B[i], 10, 6) « std::endl;
}
```

The Dbl constructor used in this snippet has the interface Dbl::Dbl(double value, int width, int precision). The use of wrapper classes allows one to control output format using an an interface that is more compact than the C++ iostream interace, and only slightly more verbose than that of the C fprint function.
Two or more constructors are provide for each wrapper class. Each class has a constructor that requires only the value of of the variable, while others require the value and field width or (as in the above example) the value, width and precision. If a field width or precision is not specified as a parameter to the constructor, it may be set after construction using setter functions.

When no value is specified for the field width or (for Dbl) the precision, default values are used. The default width and precision for all data types are given by Format::defaultWidth() and Format::defaultPrecision(). These default values may be modified using the static methods Format::setDefaultWidth() and Format::setDefaultPrecision().
Example: Suppose we wish to output the two column array described in the previous example, but are willing to use a 15 column field an 7 digits of precision for both columns. This could also be accomplished as follows:

```
double A[10], B[10];
Format::setDefaultWidth(15);
Format::setDefaultPrecision(7);
for (int i=0; i< 10; ++i) {
   std::cout « Dbl(A[i]) « Dbl(B[i]) « std::endl;
}
```

The setDefaultWidth() and setDefaultPrecision() functions are not needed if one is happy with the initial default settings, which are a width of 20 characters and a precision of 12.

10.10.3 Function Template

The write() function template provides a generic interface for formatting ostream output, which can be used within a class or function template to output data for which the type is a template parameter. The wrapper classes cannot be used directly in this situation, because they require that an object of the appropriate wrapper class be specified explicitly. To output a variable data to an ostream out, one calls write(out, data). An explicit specialization of write() is provided for each data type for which there exists a wrapper class. Each explicit specialization uses the corresponding wrapper class internally to format the output. Thus, if variable data is an int, write(out, data) is equivalent to out $<<$ Int(data). For other data types, for which there exists no wrapper class, write(out, data) is equivalent out $<<$ data.

10.10.4 Function Documentation

10.10.4.1 write() `template<typename Type >`

```
void Util::write (
             std::ostream & out,
             Type data )  [inline]
```

Function template for output in a standard format.
The write function template is designed to simplify formatted stream output of variables within class and function template, when the typename of a variable is a template parameter.
The primary template implementation simply invokes the insertion $<<$ operator for the specified type. For types controlled by the primary template (i.e., those for which no explicit specialization is provided) the expression write(out, data) is equivalent to out $<<$ data.
Explicit specializations of this method are provided for int, long, double, bool, and string. Each of these uses an appropriate wrapper class (Int, Lng, Dbl, Bool, or Str) to format output. For example, if data is an int, write(out, data) is equivalent to out $<<$ Int(data). The width and (if appropriate) precision are controlled by Format::defaultWidth() and Format::defaultWidth().
Definition at line 80 of file write.h.

10.11 Mathematics

Mathematical constants and utilities.

Classes

- class Util::Binomial

  Class for binomial coefficients (all static members)
- class Util::CardinalBSpline

  A cardinal B-spline basis function.
- class Util::Constants

  Mathematical constants.
- class Util::Polynomial< T >

  A Polynomial (i.e,.
- class Util::Rational

  A Rational number (a ratio of integers).

Functions

- bool Util::feq (double x, double y, double eps=1.0E-10)

  Are two floating point numbers equal to within round-off error?
- int Util::gcd (int a, int b)

  Compute greatest common divisor (gcd) of two integers.

10.11.1 Detailed Description

Mathematical constants and utilities.

10.11.2 Function Documentation

10.11.2.1 feq()

```
bool Util::feq (
    double x,
    double y,
    double eps = 1.0E-10 )  [inline]
```

Are two floating point numbers equal to within round-off error?
Returns true if eps > fabs(x-y)*c/(fabs(x)+fabs(y)+c), c=1.05E-5.

Parameters

x	first real argument
y	second real argument
eps	maximum tolerance for nominally "equal" values

Returns

true if equal to within tolerance, false otherwise

Definition at line 27 of file feq.h.
Referenced by Util::RadialDistribution::loadParameters(), Util::Distribution::loadParameters(), and Util::Distribution::serialize().

10.11.2.2 gcd()

```
int Util::gcd (
    int a,
    int b ) [inline]
```

Compute greatest common divisor (gcd) of two integers.

Uses Euclidean algorithm to compute gcd. Always returns a non-negative integer. If one argument is zero, the absolute value of the other is returned. Returns zero if and only if both integers are zero.

Parameters

a	1st integer
b	2nd integer

Returns

greatest common divisor of a and b

Definition at line 30 of file gcd.h.

10.12 Miscellaneous Utilities

Classes

- class Util::Bit

  Represents a specific bit location within an unsigned int.
- class Util::Exception

  A user-defined exception.
- class Util::FileMaster

  A FileMaster manages input and output files for a simulation.
- class Util::FlagSet

  A set of boolean variables represented by characters.
- class Util::Log

  A static class that holds a log output stream.
- class Util::Memory

  Provides method to allocate array.
- class Util::Notifier< Event >

  Abstract template for a notifier (or subject) in the Observer design pattern.
- class Util::Observer< Event >

  Abstract class template for observer in the observer design pattern.
- class Util::Setable< T >

  Template for a value that can be set or declared null (i.e., unknown).
- class Util::Timer

  Wall clock timer.

Functions

- std::string Util::toString (int n)

  Return string representation of an integer.
- int Util::rStrip (std::string &string)

  Strip trailing whitespace from a string.
- void Util::checkString (std::istream &in, const std::string &expected)

  Extract string from stream, and compare to expected value.
- bool Util::getLine (std::istream &in, std::stringstream &line)

  Read the next line into a stringstream.
- bool Util::getNextLine (std::istream &in, std::string &line)

  Read the next non-empty line into a string, strip trailing whitespace.
- bool Util::getNextLine (std::istream &in, std::stringstream &line)

  Read next non-empty line into a stringstream, strip trailing whitespace.
- template< typename D , typename B , typename M >
  ptrdiff_t Util::memberOffset (D &object, M B::*memPtr)

  Template for calculating offsets of data members.

10.12.1 Detailed Description

Miscellaneous utility classes and functions.

10.12.2 Function Documentation

10.12.2.1 toString() `std::string Util::toString ( int n )`

Return string representation of an integer.

Parameters

n	integer to be converted.

Definition at line 52 of file ioUtil.cpp.

10.12.2.2 rStrip() `int Util::rStrip ( std::string & string )`

Strip trailing whitespace from a string.

Parameters

string	string (stripped upon return).

Returns

length of stripped string.

Definition at line 18 of file ioUtil.cpp.
Referenced by Util::getNextLine(), and Util::XmlAttribute::match().

10.12.2.3 checkString() `void Util::checkString ( std::istream & in, const std::string & expected )`

Extract string from stream, and compare to expected value.

Exceptions

Exception	if input value differs from expected value.

Parameters

in	input stream
expected	expected value of string read from stream

Definition at line 37 of file ioUtil.cpp.
References Util::Log::file(), and UTIL_THROW.

10.12.2.4 getLine() `bool Util::getLine ( std::istream & in, std::stringstream & line )`

Read the next line into a stringstream.
Variant of std::getline(). Does not strip trailing whitespace.

Parameters

in	input stream from which to read.
line	stringstream containing line, on output.

Definition at line 62 of file ioUtil.cpp.

10.12.2.5 getNextLine() [1/2]

```
bool Util::getNextLine (
            std::istream & in,
            std::string & line )
```

Read the next non-empty line into a string, strip trailing whitespace.
Variant of std::getline() that skips empty lines.

Parameters

in	input stream from which to read.
line	string with next non-empty line, on output.

Returns

true if not end-of-file, false if end-of-file.

Definition at line 79 of file ioUtil.cpp.
References Util::rStrip().

10.12.2.6 getNextLine() [2/2]

```
bool Util::getNextLine (
            std::istream & in,
            std::stringstream & line )
```

Read next non-empty line into a stringstream, strip trailing whitespace.
Variant of std::getline() that skips empty lines and uses stringstream.

Parameters

in	input stream from which to read.
line	stringstream containing next non-empty line, on output.

Returns

true if not end-of-file, false if end-of-file.

Definition at line 100 of file ioUtil.cpp.
References Util::rStrip().

10.12.2.7 memberOffset()

```
template<typename D , typename B , typename M >
ptrdiff_t Util::memberOffset (
            D & object,
            M B::* memPtr )
```

Template for calculating offsets of data members.
Types: D - derived class B - base class M - member type
Definition at line 27 of file Offset.h.

10.13 Managers and Factories

Classes

- class Util::Factory< Data >

  Factory template.
- class Util::Manager< Data >

  Template container for pointers to objects with a common base class.

10.13.1 Detailed Description

A Manager is a container for items with a common base class.
A Manager is is a subclass of ParamComposite that manages a list of subclasses of a base class, which it creates using an associated Factory.

10.14 Parameter File IO

Classes

- class Util::Begin

  Beginning line of a composite parameter block.
- class Util::Blank

  An empty line within a parameter file.
- class Util::CArray2DParam< Type >

  A Parameter associated with a 2D built-in C array.
- class Util::CArrayParam< Type >

  A Parameter associated with a 1D C array.
- class Util::DArrayParam< Type >

  A Parameter associated with a DArray container.
- class Util::DMatrixParam< Type >

  A Parameter associated with a 2D built-in C array.
- class Util::DSymmMatrixParam< Type >

  A Parameter associated with a symmetric DMatrix.
- class Util::End

  End bracket of a ParamComposite parameter block.
- class Util::FArrayParam< Type, N >

  A Parameter associated with a FArray container.
- class Util::Label

  A label string in a file format.
- class Util::OptionalLabel

  An optional Label string in a file format.
- class Util::ParamComponent

  Abstract base class for classes that input and output parameters to file.
- class Util::ParamComposite

  An object that can read multiple parameters from file.
- class Util::Parameter

  A single variable in a parameter file.
- class Util::ScalarParam< Type >

  Template for a Parameter object associated with a scalar variable.

10.14.1 Detailed Description

Classes used to read parameters from a parameter file. Any class that must read values of member variables from a file should be derived from ParamComposite, which provides methods for reading and writing a parameter file, using a programmatically defined file format.

ParamComponent is an abstract base class. The classes ParamComposite, Parameter, Begin, End, and Blank are derived directly from ParamComponent. Parameter, Begin, End, and Blank are "leaf" notes it a tree structure.

Each subclasses of Parameter represents a parameter associated with a different type of C++ object. Such subclasses include class templates ScalarParam, CArrayParam, DArrayParam, FArrayParam, CArray2DParam and MatrixParam. The template ScalarParam represents any parameter that is associated with either a primitive C type or a user type for which their exist overloaded "<<" and ">>" file IO operators. The templates CArrayParam, DArrayParam, FArrayParam, CArray2DParam, and MatrixParam difine parameter file formats for different types of 1D and 2D arrays.

10.15 Random

Random numbers and processes.

Classes

- class Util::Ar1Process

    Generator for a discrete AR(1) Markov process.
- class Util::Random

    Random number generator.

10.15.1 Detailed Description

Random numbers and processes.

10.16 Signals (Observer Pattern)

Classes

- class Util::IFunctor< T >

  Interface for functor that wraps a void function with one argument (abstract).
- class Util::MethodFunctor< Object, T >

  Functor that wraps a one-argument class member function.
- class Util::Signal< T >

  Notifier (or subject) in the Observer design pattern.

10.16.1 Detailed Description

Classes used to implement the observer design pattern. A Signal maintains a list of registered "observers" and "notifies" each observer when Signal::notify() is called, by calling a specific method of each observer object. Observers are stored internally as a list pointers to IFunctor objects, each of which can be called using an overloaded () operator. Each Functor is created as an instance of the MethodFunctor<T>, which stores a pointer to a T object and to pointer to a method of class T, and which uses the () operator to call a specific method of a specific object.

The Signal, IFunctor, and MethodFunctor classes are all templates that take an optional parameter T that represents the typename of of a parameter that should be passed to the notify method of the Signal<T>, which then passes it to the void (const T&) operator of the IFunctor<T>. In each template, setting typename T to the the default value of T=void invokes a explicit specialization in which the void Signal<>::notify() and void IFunctor<>::operator () take no parameters. An instance of Signal<> is thus a signal that notifies observers by calling methods that take no arguments, while a Signal<T> is a signal that notifies observers by calling methods with a signature void (const &T). MethodFunctor takes two template parameters: MethodFunctor<ObserverClass, typename T=void> is a subclass of IFunctor<T> for which the (const T&) operator calls a specific void (const T&) methodof an observer of type class ObserverObject.

10.17 Space (Vector, Tensor)

Classes that represent spatial Vectors, Tensors, etc.

Classes

- class Util::Grid

  A grid of points indexed by integer coordinates.
- class Util::IntVector

  An IntVector is an integer Cartesian vector.
- class Util::Tensor

  A Tensor represents a Cartesian tensor.
- class Util::Vector

  A Vector is a Cartesian vector.

Variables

- const int Util::Dimension = 3

  Dimensionality of space.
- const int Util::DimensionSq = Dimension∗Dimension

  Square of Dimensionality of space.

10.17.1 Detailed Description

Classes that represent spatial Vectors, Tensors, etc.

10.17.2 Variable Documentation

10.17.2.1 Dimension `const int Util::Dimension = 3`

Dimensionality of space.

Definition at line 19 of file Dimension.h.

Referenced by Util::GridArray< Data >::allocate(), Util::SymmTensorAverage::clear(), Util::TensorAverage::clear(), Util::Tensor::commitMpiType(), Util::Grid::dimension(), Pscf::Mesh< D >::dimension(), Util::Tensor::dyad(), Util::Tensor::identity(), Util::Grid::isInGrid(), Util::GridArray< Data >::isInGrid(), Util::SymmTensorAverage::loadParameters(), Util::TensorAverage::loadParameters(), Util::TensorAverage::operator()(), Util::Tensor::operator()(), Util::operator<<(), Util::Tensor::operator=(), Util::operator==(), Util::operator>>(), Util::Vector::operator[](), Util::IntVector::operator[](), Util::Grid::position(), Util::GridArray< Data >::position(), Util::product(), Util::Grid::rank(), Util::SymmTensorAverage::readParameters(), Util::TensorAverage::readParameters(), Util::SymmTensorAverage::sample(), Util::TensorAverage::sample(), Util::SymmTensorAverage::serialize(), Util::TensorAverage::serialize(), Util::Tensor::setColumn(), Util::Grid::setDimensions(), Util::SymmTensorAverage::setNSamplePerBlock(), Util::TensorAverage::setNSamplePerBlock(), Util::Tensor::setRow(), Util::Grid::shift(), Util::GridArray< Data >::shift(), Util::Tensor::symmetrize(), Util::SymmTensorAverage::SymmTensorAverage(), Util::Tensor::Tensor(), Util::TensorAverage::TensorAverage(), Util::Tensor::trace(), and Util::Tensor::transpose().

10.17.2.2 DimensionSq `const int Util::DimensionSq = Dimension*Dimension`

Square of Dimensionality of space.

Definition at line 26 of file Dimension.h.

Referenced by Util::Tensor::add(), Util::Tensor::divide(), Util::Tensor::identity(), Util::Tensor::multiply(), Util::Tensor::operator∗=(), Util::Tensor::operator+=(), Util::Tensor::operator-=(), Util::Tensor::operator/=(), Util::operator<<(), Util::Tensor::operator=(), Util::operator==(), Util::operator>>(), Util::Tensor::serialize(), Util::Tensor::subtract(), Util::Tensor::Tensor(), and Util::Tensor::zero().

10.18 Util namespace

Group of modules containing classes in the Util namespace.

Modules

- Accumulators

  Statistical operations (e.g., averages or autocorrelations) on a data sequence.
- Serialization
- Container Templates
- Output Format

  Utilities to simplify formatted C++ stream output.
- Mathematics

  Mathematical constants and utilities.
- Miscellaneous Utilities
- Managers and Factories
- Parameter File IO
- Random

  Random numbers and processes.
- Signals (Observer Pattern)
- Space (Vector, Tensor)

  Classes that represent spatial Vectors, Tensors, etc.
- Xml Tag Parsers

10.18.1 Detailed Description

Group of modules containing classes in the Util namespace.

10.19 Xml Tag Parsers

Classes

- class Util::XmlAttribute

  Parser for an XML attribute.
- class Util::XmlBase

  Base class for classes that parse XML markup tags.
- class Util::XmlEndTag

  Parser for an XML end tag.
- class Util::XmlStartTag

  Parser for an XML start tag.
- class Util::XmlXmlTag

  Parser for an XML file declaration tag (first line in file).

10.19.1 Detailed Description

Classes to verify and parse XML markup tags.

10.20 Chemical Structure

Classes that describe chemical structure of polymers and solvents.

Classes

- class Pscf::BlockDescriptor

  A linear homopolymer block within a block copolymer.
- class Pscf::BondDescriptor

  A linear bond (including block-bond and joint-bond) within a block copolymer.
- class Pscf::Monomer

  Descriptor for a monomer or particle type.
- class Pscf::Species

  Base class for a molecular species (polymer or solvent).
- class Pscf::Vertex

  A junction or chain end in a block polymer.

10.20.1 Detailed Description

Classes that describe chemical structure of polymers and solvents.

10.21 Crystallography

Classes that describe crystallographic information.

Classes

- class Pscf::Basis< D >

  Symmetry-adapted basis for pseudo-spectral scft.
- class Pscf::SpaceGroup< D >

  Crystallographic space group.
- class Pscf::SpaceSymmetry< D >

  A SpaceSymmetry represents a crystallographic space group symmetry.
- struct Pscf::TWave< D >

  Simple wave struct for use within Basis construction.
- struct Pscf::TWaveNormComp< D >

  Comparator for TWave objects, based on TWave::sqNorm.
- struct Pscf::TWaveDftComp< D >

  Comparator for TWave objects, based on TWave::indicesDft.
- struct Pscf::TWaveBzComp< D >

  Comparator for TWave objects, based on TWave::indicesBz.
- class Pscf::UnitCell< D >

  Base template for UnitCell<D> classes, D=1, 2 or 3.
- class Pscf::UnitCell< 1 >

  1D crystal unit cell.
- class Pscf::UnitCell< 2 >

  2D crystal unit cell.
- class Pscf::UnitCell< 3 >

  3D crystal unit cell.
- class Pscf::UnitCellBase< D >

  Base class template for a crystallographic unit cell.

Functions

- std::string Pscf::makeGroupFileName (int D, std::string groupName)

  Generates the file name from a group name.
- template< int D >
  std::ostream & Pscf::operator<< (std::ostream &out, const SpaceGroup< D > &g)

  Output stream inserter operator for a SpaceGroup<D>.
- template< int D >
  std::istream & Pscf::operator>> (std::istream &in, SpaceGroup< D > &g)

  Input stream extractor operator for a SpaceGroup<D>.
- template< int D >
  bool Pscf::operator== (const SpaceSymmetry< D > &A, const SpaceSymmetry< D > &B)

  Are two SpaceSymmetry objects equivalent?
- template< int D >
  bool Pscf::operator!= (const SpaceSymmetry< D > &A, const SpaceSymmetry< D > &B)

  Are two SpaceSymmetry objects not equivalent?
- template< int D >
  SpaceSymmetry< D > Pscf::operator* (const SpaceSymmetry< D > &A, const SpaceSymmetry< D > &B)

Return the product A∗B of two symmetry objects.

- template<int D>
  IntVec< D > Pscf::operator∗ (const SpaceSymmetry< D > &S, const IntVec< D > &V)

  Return the IntVec<D> product S∗V of a rotation matrix and an IntVec<D>.
- template<int D>
  IntVec< D > Pscf::operator∗ (const IntVec< D > &V, const SpaceSymmetry< D > &S)

  Return the IntVec<D> product V∗S of an IntVec<D> and a rotation matrix.
- template<int D>
  std::ostream & Pscf::operator<< (std::ostream &out, const SpaceSymmetry< D > &A)

  Output stream inserter for a SpaceSymmetry<D>
- template<int D>
  std::istream & Pscf::operator>> (std::istream &in, SpaceSymmetry< D > &A)

  Input stream extractor for a SpaceSymmetry<D>
- template<int D>
  std::istream & Pscf::operator>> (std::istream &in, UnitCell< D > &cell)

  istream input extractor for a UnitCell<D>.
- template<int D>
  std::ostream & Pscf::operator<< (std::ostream &out, UnitCell< D > const &cell)

  ostream output inserter for a UnitCell<D>.
- template<class Archive , int D>
  void Pscf::serialize (Archive &ar, UnitCell< D > &cell, const unsigned int version)

  Serialize to/from an archive.
- template<int D>
  void Pscf::readUnitCellHeader (std::istream &in, UnitCell< D > &cell)

  Read UnitCell<D> from a field file header (fortran pscf format).
- template<int D>
  void Pscf::writeUnitCellHeader (std::ostream &out, UnitCell< D > const &cell)

  Write UnitCell<D> to a field file header (fortran pscf format).
- std::istream & Pscf::operator>> (std::istream &in, UnitCell< 1 >::LatticeSystem &lattice)

  istream extractor for a 1D UnitCell<1>::LatticeSystem.
- std::ostream & Pscf::operator<< (std::ostream &out, UnitCell< 1 >::LatticeSystem lattice)

  ostream inserter for a 1D UnitCell<1>::LatticeSystem.
- std::istream & Pscf::operator>> (std::istream &in, UnitCell< 2 >::LatticeSystem &lattice)

  istream extractor for a 2D UnitCell<2>::LatticeSystem.
- std::istream & Pscf::operator>> (std::istream &in, UnitCell< 3 >::LatticeSystem &lattice)

  istream extractor for a 3D UnitCell<3>::LatticeSystem.
- std::ostream & Pscf::operator<< (std::ostream &out, UnitCell< 3 >::LatticeSystem lattice)

  ostream inserter for an 3D UnitCell<3>::LatticeSystem.

10.21.1 Detailed Description

Classes that describe crystallographic information.

10.21.2 Function Documentation

10.21.2.1 makeGroupFileName()

```
std::string Pscf::makeGroupFileName (
    int D,
    std::string groupName )
```

Generates the file name from a group name.

Parameters

groupName	standard name of space group
D	dimensionality of space (D=1,2 or 3)
groupName	standard name of space group

Definition at line 25 of file groupFile.cpp.
References UTIL_THROW.
Referenced by Pscf::Basis< D >::makeBasis().

10.21.2.2 operator<<() [1/5]

```
template<int D>
std::ostream& Pscf::operator<< (
        std::ostream & out,
        const SpaceGroup< D > & g )
```

Output stream inserter operator for a SpaceGroup<D>.

Parameters

out	output stream
g	space group

Definition at line 106 of file SpaceGroup.h.
References Pscf::SymmetryGroup< SpaceSymmetry< D > >::size().

10.21.2.3 operator>>() [1/6]

```
template<int D>
std::istream& Pscf::operator>> (
        std::istream & in,
        SpaceGroup< D > & g )
```

Input stream extractor operator for a SpaceGroup<D>.

Parameters

in	input stream
g	space group

Definition at line 127 of file SpaceGroup.h.
References Pscf::SymmetryGroup< SpaceSymmetry< D > >::add(), Pscf::SymmetryGroup< SpaceSymmetry< D > >::clear(), and UTIL_CHECK.

10.21.2.4 operator==()

```
template<int D>
bool Pscf::operator== (
        const SpaceSymmetry< D > & A,
        const SpaceSymmetry< D > & B )
```

Are two SpaceSymmetry objects equivalent?

Parameters

A	first symmetry
B	second symmetry

Returns

True if A == B, false otherwise

Definition at line 357 of file SpaceSymmetry.h.

10.21.2.5 operator"!=() template<int D>

```
bool Pscf::operator!= (
            const SpaceSymmetry< D > & A,
            const SpaceSymmetry< D > & B ) [inline]
```

Are two SpaceSymmetry objects not equivalent?

Parameters

A	first symmetry
B	second symmetry

Returns

True if A != B, false otherwise

Definition at line 305 of file SpaceSymmetry.h.

10.21.2.6 operator*() [1/3] template<int D>

```
SpaceSymmetry< D > Pscf::operator* (
            const SpaceSymmetry< D > & A,
            const SpaceSymmetry< D > & B )
```

Return the product A*B of two symmetry objects.

Parameters

A	first symmetry
B	second symmetry

Returns

product A*B

Definition at line 378 of file SpaceSymmetry.h.
References Pscf::SpaceSymmetry< D >::normalize().

10.21.2.7 operator*() [2/3] template<int D>

```
IntVec< D > Pscf::operator* (
            const SpaceSymmetry< D > & S,
            const IntVec< D > & V )
```

Return the IntVec<D> product S*V of a rotation matrix and an IntVec<D>.
The product is defined to be the matrix product of the rotation matrix and the integer vector S.R * V.

Parameters

S	symmetry operation
V	integer vector

Returns

product S∗V

Definition at line 411 of file SpaceSymmetry.h.

10.21.2.8 operator∗() [3/3]

```
template<int D>
IntVec< D > Pscf::operator* (
            const IntVec< D > & V,
            const SpaceSymmetry< D > & S )
```

Return the IntVec<D> product V∗S of an IntVec<D> and a rotation matrix.

The product is defined to be the matrix product of the integer vector and the space group rotation matrix S.R ∗ V.

Parameters

V	integer vector
S	symmetry operation

Returns

product V∗S

Definition at line 428 of file SpaceSymmetry.h.

10.21.2.9 operator<<() [2/5]

```
template<int D>
std::ostream & Pscf::operator<< (
            std::ostream & out,
            const SpaceSymmetry< D > & A )
```

Output stream inserter for a SpaceSymmetry<D>

Parameters

out	output stream
A	SpaceSymmetry<D> object to be output

Returns

modified output stream

Definition at line 445 of file SpaceSymmetry.h.

10.21.2.10 operator>>() [2/6]

```
template<int D>
std::istream & Pscf::operator>> (
            std::istream & in,
            SpaceSymmetry< D > & A )
```

Input stream extractor for a SpaceSymmetry<D>

Parameters

in	input stream
A	SpaceSymmetry<D> object to be input

Returns

modified input stream

Definition at line 465 of file SpaceSymmetry.h.

10.21.2.11 operator>>() [3/6]

```
template<int D>
std::istream & Pscf::operator>> (
    std::istream & in,
    UnitCell< D > & cell )
```

istream input extractor for a UnitCell<D>.

Parameters

in	input stream
cell	UnitCell<D> to be read

Returns

modified input stream

Definition at line 21 of file UnitCell.tpp.

References Pscf::UnitCellBase< D >::nParameter_, Pscf::UnitCellBase< D >::parameters_, and Pscf::UnitCellBase< D >::setLattice().

10.21.2.12 operator<<() [3/5]

```
template<int D>
std::ostream & Pscf::operator<< (
    std::ostream & out,
    UnitCell< D > const & cell )
```

ostream output inserter for a UnitCell<D>.

Parameters

out	output stream
cell	UnitCell<D> to be written

Returns

modified output stream

Definition at line 34 of file UnitCell.tpp.

References Pscf::UnitCellBase< D >::nParameter_, and Pscf::UnitCellBase< D >::parameters_.

10.21.2.13 serialize()

```
template<class Archive , int D>
void Pscf::serialize (
    Archive & ar,
    UnitCell< D > & cell,
    const unsigned int version )
```

Serialize to/from an archive.

Parameters

ar	input or output archive
cell	UnitCell<D> object to be serialized
version	archive version id

Definition at line 48 of file UnitCell.tpp.
References Pscf::UnitCellBase< D >::nParameter_, Pscf::UnitCellBase< D >::parameters_, and Util::serializeEnum().

10.21.2.14 readUnitCellHeader() `template<int D>`

```
void Pscf::readUnitCellHeader (
            std::istream & in,
            UnitCell< D > & cell )
```

Read UnitCell<D> from a field file header (fortran pscf format).

If the unit cell has a non-null lattice system on entry, the value read from file must match this existing value, or this function throws an exception. If the lattice system is null on entry, the lattice system value is read from file. In either case, unit cell parameters (dimensions and angles) are updated using values read from file.

Parameters

in	input stream
cell	UnitCell<D> to be read

Definition at line 59 of file UnitCell.tpp.
References Pscf::UnitCellBase< D >::nParameter_, Pscf::UnitCellBase< D >::parameters_, Pscf::UnitCellBase< D >::setLattice(), and UTIL_CHECK.

10.21.2.15 writeUnitCellHeader() `template<int D>`

```
void Pscf::writeUnitCellHeader (
            std::ostream & out,
            UnitCell< D > const & cell )
```

Write UnitCell<D> to a field file header (fortran pscf format).

Parameters

out	output stream
cell	UnitCell<D> to be written

Definition at line 88 of file UnitCell.tpp.
References Pscf::UnitCellBase< D >::nParameter_, and Pscf::UnitCellBase< D >::parameters_.
Referenced by Pscf::Pspg::FieldIo< D >::writeFieldHeader().

10.21.2.16 operator>>() [4/6] `std::istream & Pscf::operator>> (`

```
            std::istream & in,
            UnitCell< 1 >::LatticeSystem & lattice )
```

istream extractor for a 1D UnitCell<1>::LatticeSystem.

Parameters

in	input stream

Parameters

lattice	UnitCell<1>::LatticeSystem to be read

Returns

modified input stream

Definition at line 52 of file UnitCell1.cpp.
References UTIL_THROW.

10.21.2.17 operator<<() [4/5]

```
std::ostream & Pscf::operator<< (
        std::ostream & out,
        UnitCell< 1 >::LatticeSystem lattice )
```

ostream inserter for a 1D UnitCell<1>::LatticeSystem.

Parameters

out	output stream
lattice	UnitCell<1>::LatticeSystem to be written

Returns

modified output stream

Definition at line 69 of file UnitCell1.cpp.
References UTIL_THROW.

10.21.2.18 operator>>() [5/6]

```
std::istream & Pscf::operator>> (
        std::istream & in,
        UnitCell< 2 >::LatticeSystem & lattice )
```

istream extractor for a 2D UnitCell<2>::LatticeSystem.

Parameters

in	input stream
lattice	UnitCell<2>::LatticeSystem to be read

Returns

modified input stream

Definition at line 157 of file UnitCell2.cpp.
References UTIL_THROW.

10.21.2.19 operator>>() [6/6]

```
std::istream & Pscf::operator>> (
        std::istream & in,
        UnitCell< 3 >::LatticeSystem & lattice )
```

istream extractor for a 3D UnitCell<3>::LatticeSystem.

Parameters

in	input stream
lattice	UnitCell<3>::LatticeSystem to be read

Returns

modified input stream

Definition at line 330 of file UnitCell3.cpp.
References UTIL_THROW.

10.21.2.20 operator<<() [5/5]

```
std::ostream & Pscf::operator<< (
    std::ostream & out,
    UnitCell< 3 >::LatticeSystem lattice )
```

ostream inserter for an 3D UnitCell<3>::LatticeSystem.

Parameters

out	output stream
lattice	UnitCell<3>::LatticeSystem to be written

Returns

modified output stream

Definition at line 365 of file UnitCell3.cpp.
References UTIL_THROW.

10.22 Homogeneous Mixtures

Classes to compute properties of spatially homogeneous mixtures.

Classes

- class Pscf::Homogeneous::Clump

    Collection of all monomers of a single type in a molecule.

- class Pscf::Homogeneous::Mixture

    A spatially homogeneous mixture.

- class Pscf::Homogeneous::Molecule

    Descriptor of a molecular species in a homogeneous mixture.

10.22.1 Detailed Description

Classes to compute properties of spatially homogeneous mixtures.

10.23 Mathematics

Miscellaneous mathematical utility classes.

Classes

- class Pscf::IntVec< D, T >

  An IntVec<D, T> is a D-component vector of elements of integer type T.
- class Pscf::LuSolver

  Solve Ax=b by LU decomposition of A.
- class Pscf::RealVec< D, T >

  A RealVec<D, T> is D-component vector with elements of floating type T.
- class Pscf::TridiagonalSolver

  Solver for Ax=b with tridiagonal matrix A.

10.23.1 Detailed Description

Miscellaneous mathematical utility classes.

10.24 Spatial Mesh

Classes to define a regular grid or mesh.

Classes

- class Pscf::Mesh< D >

    Description of a regular grid of points in a periodic domain.
- class Pscf::MeshIterator< D >

    Base class for mesh iterator class template.

10.24.1 Detailed Description

Classes to define a regular grid or mesh.

10.25 Pscf namespace

Group of modules containing classes in the Pscf namespace.

Modules

- Pscf Common

  Basic classes in the Pscf namespace, accessible to all sub-namespaces.
- GPU-Accelerated Utilities

  Classes of GPU-accelerated mathematical utilities for SCFT calculations.
- Continuous

  Group of classes containing classes in the Continuous namespace.
- Discrete

  Group of classes containing classes in the Discrete namespace.

10.25.1 Detailed Description

Group of modules containing classes in the Pscf namespace.

10.26 Pscf Common

Basic classes in the Pscf namespace, accessible to all sub-namespaces.

Modules

- Chemical Structure

  Classes that describe chemical structure of polymers and solvents.
- Crystallography

  Classes that describe crystallographic information.
- Homogeneous Mixtures

  Classes to compute properties of spatially homogeneous mixtures.
- Mathematics

  Miscellaneous mathematical utility classes.
- Spatial Mesh

  Classes to define a regular grid or mesh.
- Solver Templates

  Templates for classes that solve modified diffusion equations.

10.26.1 Detailed Description

Basic classes in the Pscf namespace, accessible to all sub-namespaces.

10.27 Solver Templates

Templates for classes that solve modified diffusion equations.

Classes

- class Pscf::BlockTmpl< TP >

  Class template for a block in a block copolymer.
- class Pscf::MixtureTmpl< TP, TS >

  A mixture of polymer and solvent species.
- class Pscf::PolymerTmpl< Block >

  Descriptor and MDE solver for an acyclic block polymer.
- class Pscf::PropagatorTmpl< TP >

  Template for propagator classes.
- class Pscf::SolventTmpl< TP >

  Template for a class representing a solvent species.
- class Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >

  A mixture of polymer and solvent species.

10.27.1 Detailed Description

Templates for classes that solve modified diffusion equations.
The templates defined in this module are designed to be used as base classes for classes that define a variety of different implementations of self-consistent field theory (SCFT), in which each implementation uses a particular set of algorithms to solve the modified diffusion equation (MDE) in a particular type of geometry.
To define an implementation of SCFT, one must define the following set of solver classes derived from these templates:

- A Propagator class, derived from PropagatorTmpl
- A Block class, derived from BlockTmpl<Propagator>
- A Polymer class, derived from PolymerTmpl<Block>
- A Solvent class, derived from SolventTmpl<Block>
- A Mixture class, derived from MixtureTmpl<Polymer, Solvent>

10.28 Fields

Fields and FFT plans for use with pseudo-spectral algorithm on a GPU.

Classes

- class Pscf::Pspg::DField< Data >

    Dynamic array with aligned data, for use with cufftw library/device code.
- class Pscf::Pspg::FFT< D >

    Fourier transform wrapper for real data.
- class Pscf::Pspg::FFTBatched< D >

    Fourier transform wrapper for real data.
- class Pscf::Pspg::FieldIo< D >

    File input/output operations for fields in several file formats.
- class Pscf::Pspg::RDField< D >

    Field of real single precision values on an FFT mesh on a device.
- class Pscf::Pspg::RDFieldDft< D >

    Fourier transform of a real field on an FFT mesh.

10.28.1 Detailed Description

Fields and FFT plans for use with pseudo-spectral algorithm on a GPU.

10.29 Mathematics

Mathematical utility classes specific to the CUDA implementation.

10.29.1 Detailed Description

Mathematical utility classes specific to the CUDA implementation.

10.30 ThreadGrid

Management of GPU resources and setting of execution configurations.

Functions

- void Pscf::Pspg::ThreadGrid::init ()

  Initialize static variables in Pspg::ThreadGrid namespace.
- void Pscf::Pspg::ThreadGrid::setThreadsPerBlock ()

  Set the number of threads per block to a default value.
- void Pscf::Pspg::ThreadGrid::setThreadsPerBlock (int nThreadsPerBlock)

  Set the number of threads per block to a specified value.
- void Pscf::Pspg::ThreadGrid::setThreadsLogical (int nThreadsLogical)

  Set the total number of threads required for execution.
- void Pscf::Pspg::ThreadGrid::setThreadsLogical (int nThreadsLogical, int &nBlocks)

  Set the total number of threads required for execution.
- void Pscf::Pspg::ThreadGrid::setThreadsLogical (int nThreadsLogical, int &nBlocks, int &nThreads)

  Set the total number of threads required for execution.
- void Pscf::Pspg::ThreadGrid::checkExecutionConfig ()

  Check the execution configuration (threads and block counts).
- int Pscf::Pspg::ThreadGrid::nBlocks ()

  Get the current number of blocks for execution.
- int Pscf::Pspg::ThreadGrid::nThreads ()

  Get the number of threads per block for execution.
- int Pscf::Pspg::ThreadGrid::nThreadsLogical ()

  Return previously requested total number of threads.
- bool Pscf::Pspg::ThreadGrid::hasUnusedThreads ()

  Indicates whether there will be unused threads.

10.30.1 Detailed Description

Management of GPU resources and setting of execution configurations.

10.30.2 Function Documentation

10.30.2.1 init() `void Pscf::Pspg::ThreadGrid::init ( )`

Initialize static variables in Pspg::ThreadGrid namespace.
Definition at line 35 of file ThreadGrid.cu.
References Pscf::Pspg::ThreadGrid::setThreadsPerBlock(), and UTIL_THROW.
Referenced by Pscf::Pspg::ThreadGrid::setThreadsLogical(), and Pscf::Pspg::Continuous::System< D >::System().

10.30.2.2 setThreadsPerBlock() `[1/2]` `void Pscf::Pspg::ThreadGrid::setThreadsPerBlock ( )`

Set the number of threads per block to a default value.
Query the hardware to determine a reasonable number.
Definition at line 48 of file ThreadGrid.cu.
Referenced by Pscf::Pspg::ThreadGrid::init(), and Pscf::Pspg::Continuous::System< D >::setOptions().

10.30.2.3 setThreadsPerBlock() [2/2] void Pscf::Pspg::ThreadGrid::setThreadsPerBlock (int nThreadsPerBlock)

Set the number of threads per block to a specified value.

Parameters

nThreadsPerBlock	the number of threads per block (input)

Definition at line 72 of file ThreadGrid.cu.
References Pscf::Pspg::ThreadGrid::checkExecutionConfig().

10.30.2.4 setThreadsLogical() [1/3] void Pscf::Pspg::ThreadGrid::setThreadsLogical (int nThreadsLogical)

Set the total number of threads required for execution.
Calculate the number of blocks, and calculate threads per block if necessary. Updates static variables.

Parameters

nThreadsLogical	total number of required threads (input)

Definition at line 80 of file ThreadGrid.cu.
References Pscf::Pspg::ThreadGrid::init(), Pscf::Pspg::ThreadGrid::nThreadsLogical(), and UTIL_ASSERT.
Referenced by Pscf::Pspg::Continuous::AmIterator< D >::allocate(), Pscf::Pspg::Continuous::Propagator< D >::allocate(), Pscf::Pspg::Continuous::AmIterator< D >::buildOmega(), Pscf::Pspg::Continuous::Mixture< D >::compute(), Pscf::Pspg::Continuous::AmIterator< D >::computeDeviation(), Pscf::Pspg::Continuous::System< D >::computeFreeEnergy(), Pscf::Pspg::Continuous::Propagator< D >::computeHead(), Pscf::Pspg::Continuous::Block< D >::computeInt(), Pscf::Pspg::Continuous::Polymer< D >::computeJoint(), Pscf::Pspg::Continuous::Mixture< D >::computeStress(), Pscf::Pspg::FFT< D >::forwardTransform(), Pscf::Pspg::FFTBatched< D >::forwardTransform(), Pscf::Pspg::Continuous::Propagator< D >::intQ(), Pscf::Pspg::Continuous::AmIterator< D >::minimizeCoeff(), Pscf::Pspg::Continuous::Block< D >::setDiscretization(), Pscf::Pspg::Continuous::Mixture< D >::setMesh(), Pscf::Pspg::ThreadGrid::setThreadsLogical(), Pscf::Pspg::Continuous::Block< D >::setupSolver(), Pscf::PolymerTmpl< Block< D > >::solve(), Pscf::Pspg::Continuous::Propagator< D >::solveBackward(), and Pscf::Pspg::Continuous::Block< D >::step().

10.30.2.5 setThreadsLogical() [2/3] void Pscf::Pspg::ThreadGrid::setThreadsLogical (int nThreadsLogical, int & nBlocks)

Set the total number of threads required for execution.
Recalculate the number of blocks, and calculate threads per block if necessary. Also updates the nBlocks output parameter.

Parameters

nThreadsLogical	total number of required threads (input)
nBlocks	updated number of blocks (output)

Definition at line 108 of file ThreadGrid.cu.
References Pscf::Pspg::ThreadGrid::nBlocks(), Pscf::Pspg::ThreadGrid::nThreadsLogical(), and Pscf::Pspg::ThreadGrid::setThreadsLogical().

10.30.2.6 setThreadsLogical() [3/3] `void Pscf::Pspg::ThreadGrid::setThreadsLogical ( int nThreadsLogical, int & nBlocks, int & nThreads )`

Set the total number of threads required for execution.

Computes and sets the number of blocks, and sets threads per block if necessary. Updates values of nBlocks and nThreads parameters in output parameters that are passed by value.

Parameters

nThreadsLogical	total number of required threads (input)
nBlocks	updated number of blocks (output)
nThreads	updated number threads per block (output)

Definition at line 116 of file ThreadGrid.cu.

References Pscf::Pspg::ThreadGrid::nBlocks(), Pscf::Pspg::ThreadGrid::nThreads(), Pscf::Pspg::ThreadGrid::n←ThreadsLogical(), and Pscf::Pspg::ThreadGrid::setThreadsLogical().

10.30.2.7 checkExecutionConfig() `void Pscf::Pspg::ThreadGrid::checkExecutionConfig ( )`

Check the execution configuration (threads and block counts).

Check for validity and optimality, based on hardware warp size and streaming multiprocessor constraints.

Definition at line 124 of file ThreadGrid.cu.

References UTIL_THROW.

Referenced by Pscf::Pspg::ThreadGrid::setThreadsPerBlock().

10.30.2.8 nBlocks() `int Pscf::Pspg::ThreadGrid::nBlocks ( )`

Get the current number of blocks for execution.

Definition at line 170 of file ThreadGrid.cu.

Referenced by Pscf::Pspg::Continuous::Propagator< D >::allocate(), Pscf::Pspg::Continuous::System< D >←::computeFreeEnergy(), and Pscf::Pspg::ThreadGrid::setThreadsLogical().

10.30.2.9 nThreads() `int Pscf::Pspg::ThreadGrid::nThreads ( )`

Get the number of threads per block for execution.

Definition at line 173 of file ThreadGrid.cu.

Referenced by Pscf::Pspg::Continuous::System< D >::computeFreeEnergy(), and Pscf::Pspg::ThreadGrid::set←ThreadsLogical().

10.30.2.10 nThreadsLogical() `int Pscf::Pspg::ThreadGrid::nThreadsLogical ( )`

Return previously requested total number of threads.

Definition at line 176 of file ThreadGrid.cu.

Referenced by Pscf::Pspg::ThreadGrid::setThreadsLogical().

10.30.2.11 hasUnusedThreads() `bool Pscf::Pspg::ThreadGrid::hasUnusedThreads ( )`

Indicates whether there will be unused threads.

Returns true iff nThreads*nBlocks != nThreadsLogical.

Definition at line 179 of file ThreadGrid.cu.

10.31 GPU-Accelerated Utilities

Classes of GPU-accelerated mathematical utilities for SCFT calculations.

Modules

- Fields

  Fields and FFT plans for use with pseudo-spectral algorithm on a GPU.
- Mathematics

  Mathematical utility classes specific to the CUDA implementation.
- ThreadGrid

  Management of GPU resources and setting of execution configurations.

Classes

- class Pscf::Pspg::Continuous::System< D >

  Main class in SCFT simulation of one system.

10.31.1 Detailed Description

Classes of GPU-accelerated mathematical utilities for SCFT calculations.

10.32 Continuous

Group of classes containing classes in the Continuous namespace.

Modules

- Iterators

  Iterators for solving the nonlinear self-consistency equations.
- Solvers

  Classes that solve modified diffusion equations for periodic microstrucures using a pseudo-spectral algorithm.

10.32.1 Detailed Description

Group of classes containing classes in the Continuous namespace.

10.33 Iterators

Iterators for solving the nonlinear self-consistency equations.

Classes

- class Pscf::Pspg::Continuous::AmIterator$<$ D $>$

  Anderson mixing iterator for the pseudo spectral method.
- class Pscf::Pspg::Continuous::Iterator$<$ D $>$

  Base class for iterative solvers for SCF equations.

10.33.1 Detailed Description

Iterators for solving the nonlinear self-consistency equations.

10.34 Solvers

Classes that solve modified diffusion equations for periodic microstrucures using a pseudo-spectral algorithm.

Classes

- class Pscf::Pspg::Continuous::Block< D >

  Block within a branched polymer.
- class Pscf::Pspg::Continuous::Mixture< D >

  Solver for a mixture of polymers and solvents.
- class Pscf::Pspg::Continuous::Polymer< D >

  Descriptor and solver for a branched polymer species.
- class Pscf::Pspg::Continuous::Propagator< D >

  MDE solver for one-direction of one block.

10.34.1 Detailed Description

Classes that solve modified diffusion equations for periodic microstrucures using a pseudo-spectral algorithm.

10.35 Discrete

Group of classes containing classes in the Discrete namespace.

Modules

- Iterators

  Iterators for solving the nonlinear self-consistency equations.
- Solvers

  Classes that solve modified diffusion equations for periodic microstrucures using a pseudo-spectral algorithm.

10.35.1 Detailed Description

Group of classes containing classes in the Discrete namespace.

10.36 Iterators

Iterators for solving the nonlinear self-consistency equations.

Classes

- class Pscf::Pspg::Discrete::AmIterator< D >

  Anderson mixing iterator for the pseudo spectral method.
- class Pscf::Pspg::Discrete::Iterator< D >

  Base class for iterative solvers for SCF equations.

10.36.1 Detailed Description

Iterators for solving the nonlinear self-consistency equations.

10.37 Solvers

Classes that solve modified diffusion equations for periodic microstrucures using a pseudo-spectral algorithm.

Classes

- class Pscf::Pspg::Discrete::Bond< D >

    Bond within a branched polymer.
- class Pscf::Pspg::Discrete::DMixture< D >

    Solver for a mixture of polymers (Discrete chain model).
- class Pscf::Pspg::Discrete::DPolymer< D >

    Descriptor and solver for a branched polymer species (Discrete chain model).
- class Pscf::Pspg::Discrete::DPropagator< D >

    CKE solver for one-direction of one bond.

10.37.1 Detailed Description

Classes that solve modified diffusion equations for periodic microstrucures using a pseudo-spectral algorithm.

10.38 Unit Test Framework

A framework for constructing unit tests for other classes.

Classes

- class CommandLine

  Abstraction of a C array of command line arguments.
- class CompositeTestRunner

  A TestRunner comprised of one or more child TestRunners.
- class ParamFileTest

  A UnitTest with a built-in input file.
- class TestRunner

  Abstract base class for classes that run tests.
- class UnitTest

  UnitTest is a base class for classes that define unit tests.
- class UnitTestRunner< UnitTestClass >

  Template for a TestRunner that runs test methods of an associated UnitTest.

10.38.1 Detailed Description

A framework for constructing unit tests for other classes.
The src/test directory contains source code for a simple unit test framework for C++ code. The library contains only header files, in which which all member functions are inlined. There are several simple programs that illustrate library usage in the directory src/test/examples/.

See also

developer_test_page

11 Namespace Documentation

11.1 Pscf Namespace Reference

Classes for polymer self-consistent field theory.

Namespaces

- Pspg

  Classes of GPU-accelerated mathematical utilities for SCFT calculations.

Classes

- class Basis

  Symmetry-adapted basis for pseudo-spectral scft.
- class BlockDescriptor

  A linear homopolymer block within a block copolymer.
- class BlockTmpl

  Class template for a block in a block copolymer.
- class BondDescriptor

  A linear bond (including block-bond and joint-bond) within a block copolymer.
- class ChiInteraction

  Flory-Huggins excess free energy model.
- class DPolymerTmpl
- class Field

  Base class template for a field defined on a spatial grid.
- class Interaction

  Base class for excess free energy models.
- class IntVec

  An IntVec< D, T> is a D-component vector of elements of integer type T.
- class LuSolver

  Solve Ax=b by LU decomposition of A.
- class Mesh

  Description of a regular grid of points in a periodic domain.
- class MeshIterator

  Base class for mesh iterator class template.
- class MixtureTmpl

  A mixture of polymer and solvent species.
- class Monomer

  Descriptor for a monomer or particle type.
- class PolymerTmpl

  Descriptor and MDE solver for an acyclic block polymer.
- class PropagatorTmpl

  Template for propagator classes.
- class RealVec

  A RealVec< D, T> is D-component vector with elements of floating type T.
- class SolventTmpl

  Template for a class representing a solvent species.
- class SpaceGroup

Crystallographic space group.
- class SpaceSymmetry

  A SpaceSymmetry represents a crystallographic space group symmetry.
- class Species

  Base class for a molecular species (polymer or solvent).
- class SymmetryGroup

  Class template for a group of elements.
- class TridiagonalSolver

  Solver for Ax=b with tridiagonal matrix A.
- struct TWave

  Simple wave struct for use within Basis construction.
- struct TWaveBzComp

  Comparator for TWave objects, based on TWave::indicesBz.
- struct TWaveDftComp

  Comparator for TWave objects, based on TWave::indicesDft.
- struct TWaveNormComp

  Comparator for TWave objects, based on TWave::sqNorm.
- class UnitCell

  Base template for UnitCell<D> classes, D=1, 2 or 3.
- class UnitCell< 1 >

  1D crystal unit cell.
- class UnitCell< 2 >

  2D crystal unit cell.
- class UnitCell< 3 >

  3D crystal unit cell.
- class UnitCellBase

  Base class template for a crystallographic unit cell.
- class Vec

  A Vec<D, T><D,T> is a D-component vector with elements of type T.
- class Vertex

  A junction or chain end in a block polymer.

Functions

- std::istream & operator>> (std::istream &in, BlockDescriptor &block)

  istream extractor for a BlockDescriptor.
- std::ostream & operator<< (std::ostream &out, const BlockDescriptor &block)

  ostream inserter for a BlockDescriptor.
- std::istream & operator>> (std::istream &in, BondDescriptor &bond)

  istream extractor for a BlockDescriptor.
- std::istream & operator<< (std::istream &out, const BondDescriptor &bond)

  ostream inserter for a BlockDescriptor.
- std::istream & operator>> (std::istream &in, Monomer &monomer)

  istream extractor for a Monomer.
- std::ostream & operator<< (std::ostream &out, const Monomer &monomer)

  ostream inserter for a Monomer.
- std::istream & operator>> (std::istream &in, Species::Ensemble &policy)

  istream extractor for a Species::Ensemble.

- std::ostream & operator<< (std::ostream &out, Species::Ensemble policy)

    ostream inserter for an Species::Ensemble.

- template<class Archive >
  void serialize (Archive &ar, Species::Ensemble &policy, const unsigned int version)

    Serialize a Species::Ensemble.

- std::string makeGroupFileName (int D, std::string groupName)

    Generates the file name from a group name.

- template<int D>
  IntVec< D > shiftToMinimum (IntVec< D > &v, IntVec< D > d, UnitCell< D > const &cell)

    Returns minimum magnitude image of DFT wavevector.

- template<int D>
  std::ostream & operator<< (std::ostream &out, const SpaceGroup< D > &g)

    Output stream inserter operator for a SpaceGroup<D>.

- template<int D>
  std::istream & operator>> (std::istream &in, SpaceGroup< D > &g)

    Input stream extractor operator for a SpaceGroup<D>.

- template<int D>
  bool operator== (const SpaceSymmetry< D > &A, const SpaceSymmetry< D > &B)

    Are two SpaceSymmetry objects equivalent?

- template<int D>
  bool operator!= (const SpaceSymmetry< D > &A, const SpaceSymmetry< D > &B)

    Are two SpaceSymmetry objects not equivalent?

- template<int D>
  SpaceSymmetry< D > operator∗ (const SpaceSymmetry< D > &A, const SpaceSymmetry< D > &B)

    Return the product A∗B of two symmetry objects.

- template<int D>
  IntVec< D > operator∗ (const SpaceSymmetry< D > &S, const IntVec< D > &V)

    Return the IntVec<D> product S∗V of a rotation matrix and an IntVec<D>.

- template<int D>
  IntVec< D > operator∗ (const IntVec< D > &V, const SpaceSymmetry< D > &S)

    Return the IntVec<D> product V∗S of an IntVec<D> and a rotation matrix.

- template<int D>
  std::ostream & operator<< (std::ostream &out, const SpaceSymmetry< D > &A)

    Output stream inserter for a SpaceSymmetry<D>

- template<int D>
  std::istream & operator>> (std::istream &in, SpaceSymmetry< D > &A)

    Input stream extractor for a SpaceSymmetry<D>

- template<int D>
  std::istream & operator>> (std::istream &in, UnitCell< D > &cell)

    istream input extractor for a UnitCell<D>.

- template<int D>
  std::ostream & operator<< (std::ostream &out, UnitCell< D > const &cell)

    ostream output inserter for a UnitCell<D>.

- template<class Archive , int D>
  void serialize (Archive &ar, UnitCell< D > &cell, const unsigned int version)

    Serialize to/from an archive.

- template<int D>
  void readUnitCellHeader (std::istream &in, UnitCell< D > &cell)

    Read UnitCell<D> from a field file header (fortran pscf format).

- template<int D>
  void writeUnitCellHeader (std::ostream &out, UnitCell< D > const &cell)

  Write UnitCell<D> to a field file header (fortran pscf format).
- std::istream & operator>> (std::istream &in, UnitCell< 1 >::LatticeSystem &lattice)

  istream extractor for a 1D UnitCell<1>::LatticeSystem.
- std::ostream & operator<< (std::ostream &out, UnitCell< 1 >::LatticeSystem lattice)

  ostream inserter for a 1D UnitCell<1>::LatticeSystem.
- std::istream & operator>> (std::istream &in, UnitCell< 2 >::LatticeSystem &lattice)

  istream extractor for a 2D UnitCell<2>::LatticeSystem.
- std::ostream & operator<< (std::ostream &out, UnitCell< 2 >::LatticeSystem lattice)

  ostream inserter for a 2D UnitCell<2>::LatticeSystem.
- std::istream & operator>> (std::istream &in, UnitCell< 3 >::LatticeSystem &lattice)

  istream extractor for a 3D UnitCell<3>::LatticeSystem.
- std::ostream & operator<< (std::ostream &out, UnitCell< 3 >::LatticeSystem lattice)

  ostream inserter for an 3D UnitCell<3>::LatticeSystem.
- template<int D, typename T >
  std::istream & operator>> (std::istream &in, IntVec< D, T > &vector)

  istream extractor for a IntVec<D, T>.
- template<int D, typename T >
  std::ostream & operator<< (std::ostream &out, const IntVec< D, T > &vector)

  ostream inserter for a IntVec<D, T>.
- template<int D, typename T >
  bool operator== (const IntVec< D, T > &v1, const IntVec< D, T > &v2)

  Equality of two IntVec<D> objects.
- template<int D, typename T >
  bool operator== (const IntVec< D, T > &v1, const Vec< D, T > &v2)

  Equality of an IntVec<D> and a Vec<D, T>
- template<int D, typename T >
  bool operator== (const Vec< D, T > &v1, const IntVec< D, T > &v2)

  Equality of an Vec<D, T> and an IntVec<D, T>
- template<int D, typename T >
  bool operator!= (const IntVec< D, T > &v1, const IntVec< D, T > &v2)

  Inequality of two IntVec<D, T> objects.
- template<int D, typename T >
  bool operator!= (const IntVec< D, T > &v1, const Vec< D, T > &v2)

  Inequality of an IntVec<D> and a Vec<D, T>
- template<int D, typename T >
  bool operator!= (const Vec< D, T > &v1, const IntVec< D, T > &v2)

  Inequality of a Vec<D, T> and an IntVec<D, T>
- template<int D, typename T >
  bool operator< (const IntVec< D, T > &v1, const IntVec< D, T > &v2)

  Less than comparison for two IntVec<D, T>s.
- template<int D, typename T >
  bool operator<= (const IntVec< D, T > &v1, const IntVec< D, T > &v2)

  Less than or equal to comparison for two IntVec<D, T>s.
- template<int D, typename T >
  bool operator> (const IntVec< D, T > &v1, const IntVec< D, T > &v2)

  Greater than comparison for two IntVec<D, T>s.

- template<int D, typename T >
  bool operator>= (const IntVec< D, T > &v1, const IntVec< D, T > &v2)

  Greater than or equal to comparison for two IntVec<D, T>s.
- template<int D, typename T >
  std::istream & operator>> (std::istream &in, RealVec< D, T > &vector)

  istream extractor for a RealVec<D, T>.
- template<int D, typename T >
  std::ostream & operator<< (std::ostream &out, const RealVec< D, T > &vector)

  ostream inserter for a RealVec<D, T>.
- template<int D, typename T >
  T dot (Vec< D, T > const &v1, Vec< D, T > const &v2)

  Return dot product of two vectors.
- template<int D, typename T >
  Vec< D, T > operator+ (Vec< D, T > const &v1, Vec< D, T > const &v2)

  Return the sum of two vectors.
- template<int D>
  std::istream & operator>> (std::istream &in, Mesh< D > &mesh)

  Input stream extractor for reading a Mesh<D> object.
- template<int D>
  std::ostream & operator<< (std::ostream &out, Mesh< D > &mesh)

  Output stream inserter for writing a Mesh<D>::LatticeSystem.

11.1.1 Detailed Description

Classes for polymer self-consistent field theory.

11.1.2 Function Documentation

11.1.2.1 operator>>() [1/7] std::istream & Pscf::operator>> (

std::istream & *in*,
BlockDescriptor & *block*)

istream extractor for a BlockDescriptor.

Parameters

in	input stream
block	BlockDescriptor to be read from stream

Returns

modified input stream

Definition at line 53 of file BlockDescriptor.cpp.

11.1.2.2 operator<<() [1/8] std::ostream & Pscf::operator<< (

std::ostream & *out*,
const BlockDescriptor & *block*)

ostream inserter for a BlockDescriptor.

Parameters

out	output stream
block	BlockDescriptor to be written to stream

Returns

modified output stream

Definition at line 66 of file BlockDescriptor.cpp.

11.1.2.3 operator>>() [2/7]

```
std::istream & Pscf::operator>> (
        std::istream & in,
        BondDescriptor & bond )
```

istream extractor for a BlockDescriptor.

Parameters

in	input stream
bond	BondDescriptor to be read from stream

Returns

modified input stream

Definition at line 41 of file BondDescriptor.cpp.

11.1.2.4 operator<<() [2/8]

```
std::istream& Pscf::operator<< (
        std::istream & out,
        const BondDescriptor & bond )
```

ostream inserter for a BlockDescriptor.

Parameters

out	output stream
bond	BondDescriptor to be written to stream

Returns

modified output stream

11.1.2.5 operator>>() [3/7]

```
std::istream & Pscf::operator>> (
        std::istream & in,
        Monomer & monomer )
```

istream extractor for a Monomer.

Parameters

in	input stream
monomer	Monomer to be read from stream

Returns

modified input stream

Definition at line 22 of file Monomer.cpp.

11.1.2.6 operator<<() [3/8]

```
std::ostream & Pscf::operator<< (
    std::ostream & out,
    const Monomer & monomer )
```

ostream inserter for a Monomer.

Parameters

out	output stream
monomer	Monomer to be written to stream

Returns

modified output stream

Definition at line 33 of file Monomer.cpp.

11.1.2.7 operator>>() [4/7]

```
std::istream & Pscf::operator>> (
    std::istream & in,
    Species::Ensemble & policy )
```

istream extractor for a Species::Ensemble.

Parameters

in	input stream
policy	Species::Ensemble to be read

Returns

modified input stream

Definition at line 22 of file Species.cpp.
References UTIL_THROW.

11.1.2.8 operator<<() [4/8]

```
std::ostream & Pscf::operator<< (
    std::ostream & out,
    Species::Ensemble policy )
```

ostream inserter for an Species::Ensemble.

Parameters

out	output stream
policy	Species::Ensemble to be written

Returns

modified output stream

Definition at line 40 of file Species.cpp.
References UTIL_THROW.

11.1.2.9 serialize() `template<class Archive >`

```
void Pscf::serialize (
            Archive & ar,
            Species::Ensemble & policy,
            const unsigned int version )
```

Serialize a Species::Ensemble.

Parameters

ar	archive object
policy	object to be serialized
version	archive version id

Definition at line 127 of file Species.h.
References Util::serializeEnum().

11.1.2.10 shiftToMinimum() `template<int D>`

```
IntVec<D> Pscf::shiftToMinimum (
            IntVec< D > & v,
            IntVec< D > d,
            UnitCell< D > const & cell )
```

Returns minimum magnitude image of DFT wavevector.

Parameters

v	IntVec<D> containing integer indices of wavevector.
d	dimensions of the discrete Fourier transform grid.
cell	UnitCell

11.1.2.11 operator<<() [5/8] `std::ostream & Pscf::operator<< (`

```
            std::ostream & out,
            UnitCell< 2 >::LatticeSystem lattice )
```

ostream inserter for a 2D UnitCell<2>::LatticeSystem.

Parameters

out	output stream
lattice	UnitCell<2>::LatticeSystem to be written

Returns

modified output stream

Definition at line 186 of file UnitCell2.cpp.
References UTIL_THROW.

11.1.2.12 operator>>() [5/7] template<int D, typename T >

```
std::istream& Pscf::operator>> (
            std::istream & in,
            IntVec< D, T > & vector )
```

istream extractor for a IntVec<D, T>.
Input elements of a vector from stream, without line breaks.

Parameters

in	input stream
vector	IntVec<D, T> to be read from stream

Returns

modified input stream

Definition at line 85 of file IntVec.h.

11.1.2.13 operator<<() [6/8] template<int D, typename T >

```
std::ostream& Pscf::operator<< (
            std::ostream & out,
            const IntVec< D, T > & vector )
```

ostream inserter for a IntVec<D, T>.
Output a IntVec<D, T> to an ostream, without line breaks.
Output elements of a vector to stream, without line breaks.

Parameters

out	output stream
vector	IntVec<D, T> to be written to stream

Returns

modified output stream

Definition at line 104 of file IntVec.h.

11.1.2.14 operator==() [1/3] template<int D, typename T >

```
bool Pscf::operator== (
            const IntVec< D, T > & v1,
            const IntVec< D, T > & v2 ) [inline]
```

Equality of two IntVec<D> objects.

Returns

true if v1 == v2, false otherwise.

Definition at line 120 of file IntVec.h.

11.1.2.15 operator==() [2/3] template<int D, typename T >

```
bool Pscf::operator== (
            const IntVec< D, T > & v1,
            const Vec< D, T > & v2 ) [inline]
```

Equality of an IntVec<D> and a Vec<D, T>

Returns

true if v1 == v2, false otherwise.

Definition at line 137 of file IntVec.h.

11.1.2.16 operator==() [3/3] template<int D, typename T >

```
bool Pscf::operator== (
            const Vec< D, T > & v1,
            const IntVec< D, T > & v2 ) [inline]
```

Equality of an Vec<D, T> and an IntVec<D, T>

Returns

true if v1 == v2, false otherwise.

Definition at line 154 of file IntVec.h.

11.1.2.17 operator"!=() [1/3] template<int D, typename T >

```
bool Pscf::operator!= (
            const IntVec< D, T > & v1,
            const IntVec< D, T > & v2 ) [inline]
```

Inequality of two IntVec<D, T> objects.

Returns

true if v1 != v2, false if v1 == v2.

Definition at line 164 of file IntVec.h.

11.1.2.18 operator"!=() [2/3] template<int D, typename T >

```
bool Pscf::operator!= (
            const IntVec< D, T > & v1,
            const Vec< D, T > & v2 ) [inline]
```

Inequality of an IntVec<D> and a Vec<D, T>

Returns

true if v1 == v2, false otherwise.

Definition at line 174 of file IntVec.h.

11.1.2.19 operator"!=() [3/3] template<int D, typename T >

```
bool Pscf::operator!= (
            const Vec< D, T > & v1,
            const IntVec< D, T > & v2 )  [inline]
```

Inequality of a Vec<D, T> and an IntVec<D, T>

Returns

true if v1 == v2, false otherwise.

Definition at line 184 of file IntVec.h.

11.1.2.20 operator<() template<int D, typename T >

```
bool Pscf::operator< (
            const IntVec< D, T > & v1,
            const IntVec< D, T > & v2 )  [inline]
```

Less than comparison for two IntVec<D, T>s.
Elements with lower array indices are treated as more signficant.

Returns

true if v1 < v2, false otherwise.

Definition at line 196 of file IntVec.h.

11.1.2.21 operator<=() template<int D, typename T >

```
bool Pscf::operator<= (
            const IntVec< D, T > & v1,
            const IntVec< D, T > & v2 )  [inline]
```

Less than or equal to comparison for two IntVec<D, T>s.
Elements with lower array indices are more signficant digits.

Returns

true if v1 < v2, false otherwise.

Definition at line 220 of file IntVec.h.

11.1.2.22 operator>() template<int D, typename T >

```
bool Pscf::operator> (
            const IntVec< D, T > & v1,
            const IntVec< D, T > & v2 )  [inline]
```

Greater than comparison for two IntVec<D, T>s.

Returns

true if v1 > v2, false otherwise.

Definition at line 242 of file IntVec.h.

11.1.2.23 operator>=() template<int D, typename T >

```
bool Pscf::operator>= (
            const IntVec< D, T > & v1,
            const IntVec< D, T > & v2 )  [inline]
```

Greater than or equal to comparison for two IntVec<D, T>s.

Returns

true if v1 >= v2, false otherwise.

Definition at line 252 of file IntVec.h.

11.1.2.24 operator>>() [6/7]

```
template<int D, typename T >
std::istream& Pscf::operator>> (
            std::istream & in,
            RealVec< D, T > & vector )
```

istream extractor for a RealVec<D, T>.
Input elements of a vector from stream, without line breaks.

Parameters

in	input stream
vector	RealVec<D, T> to be read from stream

Returns

modified input stream

Definition at line 89 of file RealVec.h.

11.1.2.25 operator<<() [7/8]

```
template<int D, typename T >
std::ostream& Pscf::operator<< (
            std::ostream & out,
            const RealVec< D, T > & vector )
```

ostream inserter for a RealVec<D, T>.
Output a RealVec<D, T> to an ostream, without line breaks.
Output elements of a vector to stream, without line breaks.

Parameters

out	output stream
vector	RealVec<D, T> to be written to stream

Returns

modified output stream

Definition at line 108 of file RealVec.h.

11.1.2.26 dot()

```
template<int D, typename T >
T Pscf::dot (
            Vec< D, T > const & v1,
            Vec< D, T > const & v2 ) [inline]
```

Return dot product of two vectors.

Parameters

v1	first input vector

Parameters

v2	second input vector

Returns

dot product v1.v2

Definition at line 283 of file Vec.h.
References dot(), and Util::setToZero().
Referenced by dot().

11.1.2.27 operator+() `template<int D, typename T >`

```
Vec<D, T> Pscf::operator+ (
            Vec< D, T > const & v1,
            Vec< D, T > const & v2 )  [inline]
```

Return the sum of two vectors.

Parameters

v1	first input vector
v2	second input vector

Returns

sum v1 + v2

Definition at line 302 of file Vec.h.
References Pscf::Vec< D, T >::add().

11.1.2.28 operator>>() [7/7] `template<int D>`

```
std::istream & Pscf::operator>> (
            std::istream & in,
            Mesh< D > & mesh )
```

Input stream extractor for reading a Mesh<D> object.

Parameters

in	input stream
mesh	Mesh<D> object to be read

Returns

modified input stream

Definition at line 136 of file Mesh.tpp.
References Pscf::Mesh< D >::setDimensions(), and UTIL_CHECK.

11.1.2.29 operator<<() [8/8] `template<int D>`

```
std::ostream & Pscf::operator<< (
```

```
std::ostream & out,
Mesh< D > & mesh )
```

Output stream inserter for writing a Mesh<D>::LatticeSystem.

Parameters

out	output stream
mesh	Mesh<D> to be written

Returns

modified output stream

Definition at line 149 of file Mesh.tpp.

11.2 Pscf::Pspg Namespace Reference

Classes of GPU-accelerated mathematical utilities for SCFT calculations.

Namespaces

- Continuous

  Classes for pseudo-spectral algorithm for continuous Gaussian chain.
- Discrete

  Classes for pseudo-spectral algorithm for discrete chain, including discrete Gaussian chain and freely jointed chain.
- ThreadGrid

  Global functions and variables to control GPU thread and block counts.

Classes

- class DField

  Dynamic array with aligned data, for use with cufftw library/device code.
- class FFT

  Fourier transform wrapper for real data.
- class FFTBatched

  Fourier transform wrapper for real data.
- class FieldIo

  File input/output operations for fields in several file formats.
- class HistMat
- class RDField

  Field of real single precision values on an FFT mesh on a device.
- class RDFieldDft

  Fourier transform of a real field on an FFT mesh.
- class Solvent

  Class representing a solvent species.
- class WaveList

11.2.1 Detailed Description

Classes of GPU-accelerated mathematical utilities for SCFT calculations.

11.3 Pscf::Pspg::Continuous Namespace Reference

Classes for pseudo-spectral algorithm for continuous Gaussian chain.

Classes

- class AmIterator

  Anderson mixing iterator for the pseudo spectral method.
- class Block

  Block within a branched polymer.
- class Iterator

  Base class for iterative solvers for SCF equations.
- class Joint
- class Mixture

  Solver for a mixture of polymers and solvents.
- class Polymer

  Descriptor and solver for a branched polymer species.
- class Propagator

  MDE solver for one-direction of one block.
- class System

  Main class in SCFT simulation of one system.

11.3.1 Detailed Description

Classes for pseudo-spectral algorithm for continuous Gaussian chain.

11.4 Pscf::Pspg::Discrete Namespace Reference

Classes for pseudo-spectral algorithm for discrete chain, including discrete Gaussian chain and freely jointed chain.

Classes

- class AmIterator

  Anderson mixing iterator for the pseudo spectral method.
- class Bond

  Bond within a branched polymer.
- class BondTmpl
- class DMixture

  Solver for a mixture of polymers (Discrete chain model).
- class DMixtureTmpl

  A mixture of polymer and solvent species.
- class DPolymer

  Descriptor and solver for a branched polymer species (Discrete chain model).
- class DPropagator

  CKE solver for one-direction of one bond.
- class Iterator

  Base class for iterative solvers for SCF equations.
- class System

11.4.1 Detailed Description

Classes for pseudo-spectral algorithm for discrete chain, including discrete Gaussian chain and freely jointed chain.

11.5 Pscf::Pspg::ThreadGrid Namespace Reference

Global functions and variables to control GPU thread and block counts.

Functions

- void init ()

  Initialize static variables in Pspg::ThreadGrid namespace.
- void setThreadsPerBlock ()

  Set the number of threads per block to a default value.
- void setThreadsPerBlock (int nThreadsPerBlock)

  Set the number of threads per block to a specified value.
- void setThreadsLogical (int nThreadsLogical)

  Set the total number of threads required for execution.
- void setThreadsLogical (int nThreadsLogical, int &nBlocks)

  Set the total number of threads required for execution.
- void setThreadsLogical (int nThreadsLogical, int &nBlocks, int &nThreads)

  Set the total number of threads required for execution.
- void checkExecutionConfig ()

  Check the execution configuration (threads and block counts).
- int nBlocks ()

  Get the current number of blocks for execution.
- int nThreads ()

  Get the number of threads per block for execution.
- int nThreadsLogical ()

  Return previously requested total number of threads.
- bool hasUnusedThreads ()

  Indicates whether there will be unused threads.

11.5.1 Detailed Description

Global functions and variables to control GPU thread and block counts.

11.6 Util Namespace Reference

Utility classes for scientific computation.

Classes

- class Ar1Process

  Generator for a discrete AR(1) Markov process.
- class Array

  Array container class template.
- class ArrayIterator

  Forward iterator for an Array or a C array.
- class ArraySet

A container for pointers to a subset of elements of an associated array.
- class ArrayStack

  A stack of fixed capacity.
- class AutoCorr

  Auto-correlation function for one sequence of Data values.
- class AutoCorrArray

  Auto-correlation function for an ensemble of sequences.
- class AutoCorrelation

  Auto-correlation function, using hierarchical algorithm.
- class AutoCorrStage

  Hierarchical auto-correlation function algorithm.
- class Average

  Calculates the average and variance of a sampled property.
- class AverageStage

  Evaluate average with hierarchical blocking error analysis.
- class Begin

  Beginning line of a composite parameter block.
- class BinaryFileIArchive

  Saving archive for binary istream.
- class BinaryFileOArchive

  Saving / output archive for binary ostream.
- class Binomial

  Class for binomial coefficients (all static members)
- class Bit

  Represents a specific bit location within an unsigned int.
- class Blank

  An empty line within a parameter file.
- class Bool

  Wrapper for an bool value, for formatted ostream output.
- class CardinalBSpline

  A cardinal B-spline basis function.
- class CArray2DParam

  A Parameter associated with a 2D built-in C array.
- class CArrayParam

  A Parameter associated with a 1D C array.
- class Constants

  Mathematical constants.
- class ConstArrayIterator

  Forward const iterator for an Array or a C array.
- class ConstPArrayIterator

  Forward iterator for a PArray.
- class DArray

  Dynamically allocatable contiguous array template.
- class DArrayParam

  A Parameter associated with a DArray container.
- class Dbl

  Wrapper for a double precision number, for formatted ostream output.

- class Distribution

  A distribution (or histogram) of values for a real variable.
- class DMatrix

  Dynamically allocated Matrix.
- class DMatrixParam

  A Parameter associated with a 2D built-in C array.
- class DPArray

  A dynamic array that only holds pointers to its elements.
- class DRaggedMatrix

  Dynamically allocated RaggedMatrix.
- class DSArray

  Dynamically allocated array with variable logical size.
- class DSymmMatrixParam

  A Parameter associated with a symmetric DMatrix.
- class End

  End bracket of a ParamComposite parameter block.
- class Exception

  A user-defined exception.
- class Factory

  Factory template.
- class FArray

  A fixed size (static) contiguous array template.
- class FArrayParam

  A Parameter associated with a FArray container.
- class FileMaster

  A FileMaster manages input and output files for a simulation.
- class FlagSet

  A set of boolean variables represented by characters.
- class FlexPtr

  A pointer that may or may not own the object to which it points.
- class FMatrix

  Fixed Size Matrix.
- class Format

  Base class for output wrappers for formatted C++ ostream output.
- class FPArray

  Statically allocated pointer array.
- class FSArray

  A fixed capacity (static) contiguous array with a variable logical size.
- class GArray

  An automatically growable array, analogous to a std::vector.
- class GPArray

  An automatically growable PArray.
- class Grid

  A grid of points indexed by integer coordinates.
- class GridArray

  Multi-dimensional array with the dimensionality of space.
- class GStack

An automatically growable Stack.

- class IFunctor

  Interface for functor that wraps a void function with one argument (abstract).
- class IFunctor< void >

  Interface for functor that wraps a void function with no arguments (abstract).
- class Int

  Wrapper for an int, for formatted ostream output.
- class IntDistribution

  A distribution (or histogram) of values for an int variable.
- class IntVector

  An IntVector is an integer Cartesian vector.
- class Label

  A label string in a file format.
- class List

  Linked list class template.
- class ListArray

  An array of objects that are accessible by one or more linked List objects.
- class ListIterator

  Bidirectional iterator for a List.
- class Lng

  Wrapper for a long int, for formatted ostream output.
- class Log

  A static class that holds a log output stream.
- class Manager

  Template container for pointers to objects with a common base class.
- class Matrix

  Two-dimensional array container template (abstract).
- class MeanSqDispArray

  Mean-squared displacement (MSD) vs.
- class Memory

  Provides method to allocate array.
- class MemoryCounter

  Archive to computed packed size of a sequence of objects, in bytes.
- class MemoryIArchive

  Input archive for packed heterogeneous binary data.
- class MemoryOArchive

  Save archive for packed heterogeneous binary data.
- class MethodFunctor

  Functor that wraps a one-argument class member function.
- class MethodFunctor< Object, void >

  Functor that wraps a class member function with no arguments.
- class MpiFileIo

  Identifies whether this processor may do file I/O.
- class MpiLoader

  Provides methods for MPI-aware loading of data from input archive.
- class MpiLogger

  Allows information from every processor in a communicator, to be output in rank sequence.

- class MpiStructBuilder

    A MpiStructBuilder objects is used to create an MPI Struct datatype.
- class MpiTraits

    Default MpiTraits class.
- class MpiTraits< bool >

    MpiTraits<bool> explicit specialization.
- class MpiTraits< char >

    MpiTraits<char> explicit specialization.
- class MpiTraits< double >

    MpiTraits<double> explicit specialization.
- class MpiTraits< float >

    MpiTraits<float> explicit specialization.
- class MpiTraits< int >

    MpiTraits<int> explicit specialization.
- class MpiTraits< IntVector >

    Explicit specialization MpiTraits<IntVector>.
- class MpiTraits< long >

    MpiTraits<long> explicit specialization.
- class MpiTraits< long double >

    MpiTraits<long double> explicit specialization.
- class MpiTraits< Rational >

    Explicit specialization MpiTraits<Rational>.
- class MpiTraits< short >

    MpiTraits<short> explicit specialization.
- class MpiTraits< Tensor >

    Explicit specialization MpiTraits<Tensor>.
- class MpiTraits< unsigned char >

    MpiTraits<unsigned char> explicit specialization.
- class MpiTraits< unsigned int >

    MpiTraits<unsigned int> explicit specialization.
- class MpiTraits< unsigned long >

    MpiTraits<unsigned long> explicit specialization.
- class MpiTraits< unsigned short >

    MpiTraits<unsigned short> explicit specialization.
- class MpiTraits< Vector >

    Explicit specialization MpiTraits<Vector>.
- class MpiTraitsNoType

    Base class for MpiTraits with no type.
- class MTRand

    Generates double floating point numbers in the half-open interval [0, 1)
- class MTRand53

    generates 53 bit resolution doubles in the half-open interval [0, 1)
- class MTRand_closed

    Generates double floating point numbers in the closed interval [0, 1].
- class MTRand_int32

    Mersenne Twister random number generator engine.
- class MTRand_open

Generates double floating point numbers in the open interval (0, 1).
- class Node

  Linked List Node, class template.
- class Notifier

  Abstract template for a notifier (or subject) in the Observer design pattern.
- class Observer

  Abstract class template for observer in the observer design pattern.
- class OptionalLabel

  An optional Label string in a file format.
- class Pair

  An array of exactly 2 objects.
- class ParamComponent

  Abstract base class for classes that input and output parameters to file.
- class ParamComposite

  An object that can read multiple parameters from file.
- class Parameter

  A single variable in a parameter file.
- class PArray

  An array that only holds pointers to its elements.
- class PArrayIterator

  Forward iterator for a PArray.
- class Polynomial

  A Polynomial (i.e,.
- class RadialDistribution

  Distribution (or histogram) of values for particle separations.
- class RaggedMatrix

  A 2D array in which different rows can have different lengths.
- class Random

  Random number generator.
- class RArray

  An Array that acts as a reference to another Array or C array.
- class Rational

  A Rational number (a ratio of integers).
- class RingBuffer

  Class for storing history of previous values in an array.
- class ScalarParam

  Template for a Parameter object associated with a scalar variable.
- class ScopedPtr

  A very simple RAII pointer.
- class Serializable

  Abstract class for serializable objects.
- class Setable

  Template for a value that can be set or declared null (i.e., unknown).
- class Signal

  Notifier (or subject) in the Observer design pattern.
- class Signal< void >

  Notifier (or subject) in the Observer design pattern (zero parameters).

- class SSet

  Statically allocated array of pointers to an unordered set.
- class Str

  Wrapper for a std::string, for formatted ostream output.
- class SymmTensorAverage

  Calculates averages of all components of a Tensor-valued variable.
- class Tensor

  A Tensor represents a Cartesian tensor.
- class TensorAverage

  Calculates averages of all components of a Tensor-valued variable.
- class TextFileIArchive

  Loading archive for text istream.
- class TextFileOArchive

  Saving archive for character based ostream.
- class Timer

  Wall clock timer.
- class Vector

  A Vector is a Cartesian vector.
- class XdrFileIArchive

  Loading / input archive for binary XDR file.
- class XdrFileOArchive

  Saving / output archive for binary XDR file.
- class XmlAttribute

  Parser for an XML attribute.
- class XmlBase

  Base class for classes that parse XML markup tags.
- class XmlEndTag

  Parser for an XML end tag.
- class XmlStartTag

  Parser for an XML start tag.
- class XmlXmlTag

  Parser for an XML file declaration tag (first line in file).

Typedefs

- typedef unsigned char Byte

  Define a "Byte" type.

Functions

- float product (float a, float b)

  Product for float Data.
- double product (double a, double b)

  Product for double Data.
- double product (const Vector &a, const Vector &b)

  Dot product for Vector Data.
- double product (const Tensor &a, const Tensor &b)

  Double contraction for Tensor Data.

- complex< float > product (complex< float > a, complex< float > b)

    Inner product for complex<float> Data.
- complex< double > product (complex< double > a, complex< double > b)

    Inner product for complex<double> Data.
- void setToZero (int &value)

    Set an int variable to zero.
- void setToZero (float &value)

    Set a float variable to zero.
- void setToZero (double &value)

    Set a double variable to zero.
- void setToZero (Vector &value)

    Set a Vector variable to zero.
- void setToZero (Tensor &value)

    Set a Vector variable to zero.
- void setToZero (complex< float > &value)

    Set a complex<float> variable to zero.
- void setToZero (complex< double > &value)

    Set a complex<double> variable to zero.
- template<typename T >
  int memorySize (T &data)

    Function template to compute memory size of one object.
- template<class Archive , typename T >
  void serialize (Archive &ar, T &data, const unsigned int version)

    Serialize one object of type T.
- template<class Archive , typename T >
  void serializeEnum (Archive &ar, T &data, const unsigned int version=0)

    Serialize an enumeration value.
- template<class Archive , typename T >
  void serializeCheck (Archive &ar, T &data, const char ∗label="")

    Save a value, or save and check correctness on loading.
- template<typename Data >
  std::istream & operator>> (std::istream &in, Pair< Data > &pair)

    Input a Pair from an istream.
- template<typename Data >
  std::ostream & operator<< (std::ostream &out, const Pair< Data > &pair)

    Output a Pair to an ostream, without line breaks.
- std::istream & operator>> (std::istream &in, Bool &object)

    Input stream extractor for an Bool object.
- std::ostream & operator<< (std::ostream &out, const Bool &object)

    Output stream inserter for an Bool object.
- std::istream & operator>> (std::istream &in, Dbl &object)

    Input stream extractor for an Dbl object.
- std::ostream & operator<< (std::ostream &out, const Dbl &object)

    Output stream inserter for an Dbl object.
- std::istream & operator>> (std::istream &in, Int &object)

    Input stream extractor for an Int object.
- std::ostream & operator<< (std::ostream &out, const Int &object)

    Output stream inserter for an Int object.

- std::istream & operator>> (std::istream &in, Lng &object)

  Input stream extractor for an Lng object.
- std::ostream & operator<< (std::ostream &out, const Lng &object)

  Output stream inserter for an Lng object.
- std::istream & operator>> (std::istream &in, Str &object)

  Input stream extractor for an Str object.
- std::ostream & operator<< (std::ostream &out, const Str &object)

  Output stream inserter for an Str object.
- template<> void write (std::ostream &out, double data)

  Explicit specialization of write for double data.
- template<> void write (std::ostream &out, std::complex< double > data)

  Explicit specialization of write for double data.
- template<> void write (std::ostream &out, int data)

  Explicit specialization of write for int data.
- template<> void write (std::ostream &out, long data)

  Explicit specialization of write for long data.
- template<> void write (std::ostream &out, bool data)

  Explicit specialization of write for bool data.
- template<typename Type >
  void write (std::ostream &out, Type data)

  Function template for output in a standard format.
- template<> void write (std::ostream &out, std::string data)

  Explicit specialization of write for std::string data.
- bool feq (double x, double y, double eps=1.0E-10)

  Are two floating point numbers equal to within round-off error?
- int gcd (int a, int b)

  Compute greatest common divisor (gcd) of two integers.
- template<typename T >
  bool operator== (Polynomial< T > &a, Polynomial< T > &b)

  Equality operator for polynomials.
- template<typename T >
  bool operator!= (Polynomial< T > &a, Polynomial< T > &b)

  Inequality operator for polynomials.
- template<typename T >
  Polynomial< T > operator- (Polynomial< T > const &a)

  Unary negation of polynomial.
- std::ostream & operator<< (std::ostream &out, Rational const &rational)

  Output stream inserter for a Rational.
- Rational operator+ (Rational const &a, Rational const &b)

  Compute sum of two rationals.
- Rational operator+ (Rational const &a, int b)

  Compute sum of rational and integer.
- Rational operator+ (int b, Rational const &a)

  Compute sum of integer and integer.
- Rational operator- (Rational const &a, Rational const &b)

  Compute difference of rationals.
- Rational operator- (Rational const &a, int b)

Compute difference of rational and integer.

- Rational operator- (int b, Rational const &a)

  Compute difference of integer and rational.

- Rational operator* (Rational const &a, Rational const &b)

  Compute product of rationals.

- Rational operator* (Rational const &a, int b)

  Compute product of rational and integer.

- Rational operator* (int b, Rational const &a)

  Compute product of integer and rational.

- Rational operator/ (Rational const &a, Rational const &b)

  Compute quotient of two rationals.

- Rational operator/ (Rational const &a, int b)

  Compute quotient Rational divided by integer.

- Rational operator/ (int b, Rational const &a)

  Compute quotient integer divided by Rational.

- Rational operator- (Rational const &a)

  Unary negation of Rational.

- bool operator== (Rational const &a, Rational const &b)

  Equality operators.

- bool operator== (Rational const &a, int b)

  Equality operator for a Rational and an integer.

- bool operator== (int b, Rational const &a)

  Equality operator for an integer and a Rational.

- bool operator!= (Rational const &a, Rational const &b)

  Inequality operators.

- bool operator!= (Rational const &a, int b)

  Inequality operator for a Rational and an integer.

- bool operator!= (int b, Rational const &a)

  Inequality operator for an integer and a Rational.

- void MpiThrow (Exception &e)

  Function to throw exception in MPI code.

- void initStatic ()

  Guarantee initialization of all static class members in Util namespace.

- int rStrip (std::string &string)

  Strip trailing whitespace from a string.

- void checkString (std::istream &in, const std::string &expected)

  Extract string from stream, and compare to expected value.

- std::string toString (int n)

  Return string representation of an integer.

- bool getLine (std::istream &in, std::stringstream &line)

  Read the next line into a stringstream.

- bool getNextLine (std::istream &in, std::string &line)

  Read the next non-empty line into a string, strip trailing whitespace.

- bool getNextLine (std::istream &in, std::stringstream &line)

  Read next non-empty line into a stringstream, strip trailing whitespace.

- void checkRequiredIstream (std::istream &in)

  Check status of a std::istream just before reading required variable.

- template< typename D , typename B , typename M >
  ptrdiff_t memberOffset (D &object, M B::∗memPtr)

  Template for calculating offsets of data members.
- template< typename D , typename B >
  ptrdiff_t baseOffset (D &object)

  Template for calculating offsets of base class subobjects.
- template<> void send< bool > (MPI::Comm &comm, bool &data, int dest, int tag)

  Explicit specialization of send for bool data.
- template<> void recv< bool > (MPI::Comm &comm, bool &data, int source, int tag)

  Explicit specialization of recv for bool data.
- template<> void bcast< bool > (MPI::Intracomm &comm, bool &data, int root)

  Explicit specialization of bcast for bool data.
- template<> void send< std::string > (MPI::Comm &comm, std::string &data, int dest, int tag)

  Explicit specialization of send for std::string data.
- template<> void recv< std::string > (MPI::Comm &comm, std::string &data, int source, int tag)

  Explicit specialization of recv for std::string data.
- template<> void bcast< std::string > (MPI::Intracomm &comm, std::string &data, int root)

  Explicit specialization of bcast for std::string data.
- template< typename T >
  void send (MPI::Comm &comm, T &data, int dest, int tag)

  Send a single T value.
- template< typename T >
  void recv (MPI::Comm &comm, T &data, int source, int tag)

  Receive a single T value.
- template< typename T >
  void bcast (MPI::Intracomm &comm, T &data, int root)

  Broadcast a single T value.
- template< typename T >
  void send (MPI::Comm &comm, T ∗array, int count, int dest, int tag)

  Send a C-array of T values.
- template< typename T >
  void recv (MPI::Comm &comm, T ∗array, int count, int source, int tag)

  Receive a C-array of T objects.
- template< typename T >
  void bcast (MPI::Intracomm &comm, T ∗array, int count, int root)

  Broadcast a C-array of T objects.
- template< typename T >
  void send (MPI::Comm &comm, DArray< T > &array, int count, int dest, int tag)

  Send a DArray<T> container.
- template< typename T >
  void recv (MPI::Comm &comm, DArray< T > &array, int count, int source, int tag)

  Receive a DArray<T> container.
- template< typename T >
  void bcast (MPI::Intracomm &comm, DArray< T > &array, int count, int root)

  Broadcast a DArray<T> container.
- template< typename T >
  void send (MPI::Comm &comm, DMatrix< T > &matrix, int m, int n, int dest, int tag)

  Send a DMatrix<T> container.

- template<typename T >
  void recv (MPI::Comm &comm, DMatrix< T > &matrix, int m, int n, int source, int tag)

  Receive a DMatrix<T> container.
- template<typename T >
  void bcast (MPI::Intracomm &comm, DMatrix< T > &matrix, int m, int n, int root)

  Broadcast a DMatrix<T> container.
- std::istream & operator>> (std::istream &in, Label label)

  Extractor for Label.
- std::ostream & operator<< (std::ostream &out, Label label)

  Inserter for Label.
- template<typename T >
  bool isNull (FlexPtr< T > p)

  Return true iff the enclosed built-in pointer is null.
- template<typename T >
  bool isNull (T *ptr)

  Return true iff a built-in pointer is null.
- template<typename T >
  bool isNull (ScopedPtr< T > p)

  Return true iff the enclosed built-in pointer is null.
- bool operator== (const IntVector &v1, const IntVector &v2)

  Equality for IntVectors.
- bool operator== (const IntVector &v1, const int *v2)

  Equality of IntVector and C array.
- bool operator== (const int *v1, const IntVector &v2)

  Equality of C array and IntVector.
- bool operator!= (const IntVector &v1, const IntVector &v2)

  Inequality of two IntVectors.
- bool operator!= (const IntVector &v1, const int *v2)

  Inequality of IntVector and C array.
- bool operator!= (const int *v1, const IntVector &v2)

  Inequality of C array and IntVector.
- std::istream & operator>> (std::istream &in, IntVector &vector)

  istream extractor for a IntVector.
- std::ostream & operator<< (std::ostream &out, const IntVector &vector)

  ostream inserter for a IntVector.
- bool operator== (const Tensor &t1, const Tensor &t2)

  Equality for Tensors.
- bool operator== (const Tensor &t1, const double t2[ ][Dimension])

  Equality of Tensor and 2D C array.
- bool operator== (const double t1[ ][Dimension], const Tensor &t2)

  Equality of C array and Tensor.
- bool operator!= (const Tensor &t1, const Tensor &t2)

  Negation of t1 == t2 (tensors t1 and t2)
- bool operator!= (const Tensor &t1, const double a2[ ][Dimension])

  Negation of t1 == a2 (tensor t1, 2D array a2)
- bool operator!= (const double a1[ ][Dimension], const Tensor &t2)

  Negation of t1 == a2 (tensor t2, 2D array a1)
- std::istream & operator>> (std::istream &in, Tensor &tensor)

istream extractor for a Tensor.

- std::ostream & operator<< (std::ostream &out, const Tensor &tensor)

  ostream inserter for a Tensor.
- bool operator== (const Vector &v1, const Vector &v2)

  Equality for Vectors.
- bool operator== (const Vector &v1, const double *v2)

  Equality of Vector and C array.
- bool operator== (const double *v1, const Vector &v2)

  Equality of C array and Vector.
- bool operator!= (const Vector &v1, const Vector &v2)

  Inequality of two Vectors.
- bool operator!= (const Vector &v1, const double *v2)

  Inequality of Vector and C array.
- bool operator!= (const double *v1, const Vector &v2)

  Inequality of C array and Vector.
- std::istream & operator>> (std::istream &in, Vector &vector)

  istream extractor for a Vector.
- std::ostream & operator<< (std::ostream &out, const Vector &vector)

  ostream inserter for a Vector.

Variables

- const int Dimension = 3

  Dimensionality of space.
- const int DimensionSq = Dimension*Dimension

  Square of Dimensionality of space.

11.6.1 Detailed Description

Utility classes for scientific computation.

11.6.2 Typedef Documentation

11.6.2.1 Byte `typedef unsigned char Util::Byte`

Define a "Byte" type.
Definition at line 19 of file Byte.h.

11.6.3 Function Documentation

11.6.3.1 product() [1/6]

```
float Util::product (
        float a,
        float b )   [inline]
```

Product for float Data.
Definition at line 22 of file product.h.
Referenced by Util::AutoCorr< Data, Product >::autoCorrelation(), Util::AutoCorr< Data, Product >::corrTime(), Util::AutoCorrArray< Data, Product >::corrTime(), Util::AutoCorr< Data, Product >::output(), Util::AutoCorrStage< Data, Product >::sample(), Util::AutoCorr< Data, Product >::sample(), and Util::AutoCorrArray< Data, Product >::sample().

11.6.3.2 product() [2/6]

```
double Util::product (
    double a,
    double b ) [inline]
```

Product for double Data.

Definition at line 28 of file product.h.

11.6.3.3 product() [3/6]

```
double Util::product (
    const Vector & a,
    const Vector & b ) [inline]
```

Dot product for Vector Data.

Definition at line 34 of file product.h.

References Util::Vector::dot().

11.6.3.4 product() [4/6]

```
double Util::product (
    const Tensor & a,
    const Tensor & b ) [inline]
```

Double contraction for Tensor Data.

Definition at line 40 of file product.h.

References Dimension.

11.6.3.5 product() [5/6]

```
complex<float> Util::product (
    complex< float > a,
    complex< float > b ) [inline]
```

Inner product for complex<float> Data.

Definition at line 55 of file product.h.

11.6.3.6 product() [6/6]

```
complex<double> Util::product (
    complex< double > a,
    complex< double > b ) [inline]
```

Inner product for complex<double> Data.

Definition at line 61 of file product.h.

11.6.3.7 setToZero() [1/7]

```
void Util::setToZero (
    int & value ) [inline]
```

Set an int variable to zero.

Parameters

value	value to be zeroed.

Definition at line 25 of file setToZero.h.

Referenced by Util::AutoCorr< Data, Product >::AutoCorr(), Util::AutoCorrArray< Data, Product >::AutoCorrArray(), Util::AutoCorrStage< Data, Product >::autoCorrelation(), Util::AutoCorrStage< Data, Product >::AutoCorrStage(), Util::AutoCorr< Data, Product >::clear(), Util::AutoCorrStage< Data, Product >::clear(), Util::MeanSqDispArray< Data >::clear(), Util::AutoCorrArray< Data, Product >::clear(), Util::AutoCorr< Data, Product >::corrTime(), Util::AutoCorrArray< Data, Product >::corrTime(), Util::AutoCorrStage< Data, Product >::corrTime(), Pscf::dot(), Util::Polynomial< double >::operator*=(), Util::Polynomial< double >::operator=(), Util::AutoCorrStage< Data, Product >::output(), Util::AutoCorrStage< Data, Product >::sample(), and Pscf::Vec< D, int >::setToZero().

11.6.3.8 setToZero() [2/7]

```
void Util::setToZero (
    float & value ) [inline]
```

Set a float variable to zero.

Parameters

value	value to be zeroed.

Definition at line 33 of file setToZero.h.

11.6.3.9 setToZero() [3/7]

```
void Util::setToZero (
    double & value ) [inline]
```

Set a double variable to zero.

Parameters

value	value to be zeroed.

Definition at line 41 of file setToZero.h.

11.6.3.10 setToZero() [4/7]

```
void Util::setToZero (
    Vector & value ) [inline]
```

Set a Vector variable to zero.

Parameters

value	value to be zeroed.

Definition at line 49 of file setToZero.h.
References Util::Vector::zero().

11.6.3.11 setToZero() [5/7]

```
void Util::setToZero (
    Tensor & value ) [inline]
```

Set a Vector variable to zero.

Parameters

value	value to be zeroed.

Definition at line 57 of file setToZero.h.
References Util::Tensor::zero().

11.6.3.12 setToZero() [6/7]

```
void Util::setToZero (
    complex< float > & value ) [inline]
```

Set a complex<float> variable to zero.

Parameters

value	value to be zeroed.

Definition at line 65 of file setToZero.h.

11.6.3.13 setToZero() [7/7]

```
void Util::setToZero (
            complex< double > & value )  [inline]
```

Set a complex<double> variable to zero.

Parameters

value	value to be zeroed.

Definition at line 73 of file setToZero.h.

11.6.3.14 memorySize()

```
template<typename T >
int Util::memorySize (
            T & data )
```

Function template to compute memory size of one object.
Definition at line 130 of file MemoryCounter.h.
References Util::MemoryCounter::size().

11.6.3.15 operator>>() [1/10]

```
template<typename Data >
std::istream& Util::operator>> (
            std::istream & in,
            Pair< Data > & pair )
```

Input a Pair from an istream.

Parameters

in	istream from which to read
pair	Pair to be read

Definition at line 44 of file Pair.h.

11.6.3.16 operator<<() [1/11]

```
template<typename Data >
std::ostream& Util::operator<< (
            std::ostream & out,
            const Pair< Data > & pair )
```

Output a Pair to an ostream, without line breaks.

Parameters

out	ostream to which to write
pair	Pair to be written

Definition at line 57 of file Pair.h.

11.6.3.17 operator>>() [2/10]

```
std::istream & Util::operator>> (
    std::istream & in,
    Bool & object )
```

Input stream extractor for an Bool object.

Parameters

in	input stream
object	Bool object to be read from stream

Returns

modified input stream

Definition at line 47 of file Bool.cpp.

11.6.3.18 operator<<() [2/11]

```
std::ostream & Util::operator<< (
    std::ostream & out,
    const Bool & object )
```

Output stream inserter for an Bool object.

Parameters

out	output stream
object	Bool to be written to stream

Returns

modified output stream

Definition at line 56 of file Bool.cpp.

11.6.3.19 operator>>() [3/10]

```
std::istream & Util::operator>> (
    std::istream & in,
    Dbl & object )
```

Input stream extractor for an Dbl object.

Parameters

in	input stream
object	Dbl object to be read from stream

Returns

modified input stream

Definition at line 73 of file Dbl.cpp.

11.6.3.20 operator<<() [3/11]

```
std::ostream & Util::operator<< (
        std::ostream & out,
        const Dbl & object )
```

Output stream inserter for an Dbl object.

Parameters

out	output stream
object	Dbl to be written to stream

Returns

modified output stream

Definition at line 86 of file Dbl.cpp.

11.6.3.21 operator>>() [4/10]

```
std::istream & Util::operator>> (
        std::istream & in,
        Int & object )
```

Input stream extractor for an Int object.

Parameters

in	input stream
object	Int object to be read from stream

Returns

modified input stream

Definition at line 71 of file Int.cpp.

11.6.3.22 operator<<() [4/11]

```
std::ostream & Util::operator<< (
        std::ostream & out,
        const Int & object )
```

Output stream inserter for an Int object.

Parameters

out	output stream
object	Int to be written to stream

Returns

modified output stream

Definition at line 84 of file Int.cpp.

11.6.3.23 operator>>() [5/10]

```
std::istream & Util::operator>> (
        std::istream & in,
```

Lng & *object*)

Input stream extractor for an Lng object.

Parameters

in	input stream
object	Lng object to be read from stream

Returns

modified input stream

Definition at line 53 of file Lng.cpp.

11.6.3.24 operator<<() [5/11] std::ostream & Util::operator<< (

std::ostream & *out*,
const Lng & *object*)

Output stream inserter for an Lng object.

Parameters

out	output stream
object	Lng to be written to stream

Returns

modified output stream

Definition at line 66 of file Lng.cpp.

11.6.3.25 operator>>() [6/10] std::istream & Util::operator>> (

std::istream & *in*,
Str & *object*)

Input stream extractor for an Str object.

Parameters

in	input stream
object	Str object to be read from stream

Returns

modified input stream

Definition at line 49 of file Str.cpp.

11.6.3.26 operator<<() [6/11] std::ostream & Util::operator<< (

std::ostream & *out*,
const Str & *object*)

Output stream inserter for an Str object.

Parameters

out	output stream
object	Str to be written to stream

Returns

modified output stream

Definition at line 58 of file Str.cpp.

11.6.3.27 write() [1/6] `template<>`

```
void Util::write (
        std::ostream & out,
        double data )
```

Explicit specialization of write for double data.
Definition at line 20 of file write.cpp.

11.6.3.28 write() [2/6] `template<>`

```
void Util::write (
        std::ostream & out,
        std::complex< double > data )
```

Explicit specialization of write for double data.
Definition at line 24 of file write.cpp.

11.6.3.29 write() [3/6] `template<>`

```
void Util::write (
        std::ostream & out,
        int data )
```

Explicit specialization of write for int data.
Definition at line 28 of file write.cpp.

11.6.3.30 write() [4/6] `template<>`

```
void Util::write (
        std::ostream & out,
        long data )
```

Explicit specialization of write for long data.
Definition at line 32 of file write.cpp.

11.6.3.31 write() [5/6] `template<>`

```
void Util::write (
        std::ostream & out,
        bool data )
```

Explicit specialization of write for bool data.
Definition at line 36 of file write.cpp.

11.6.3.32 write() [6/6] template<>

```
void Util::write (
        std::ostream & out,
        std::string data )
```

Explicit specialization of write for std::string data.

11.6.3.33 operator==() [1/13] template<typename T >

```
bool Util::operator== (
        Polynomial< T > & a,
        Polynomial< T > & b )
```

Equality operator for polynomials.

Two polynomials are equal iff they have the same degree and the the same values for all coefficients.

Parameters

a	1st polynomial
b	2nd polynomial

Returns

true if a != b

Definition at line 676 of file Polynomial.h.

11.6.3.34 operator"!=() [1/13] template<typename T >

```
bool Util::operator!= (
        Polynomial< T > & a,
        Polynomial< T > & b )
```

Inequality operator for polynomials.

Parameters

a	1st polynomial
b	2nd polynomial

Returns

true if a != b

Definition at line 695 of file Polynomial.h.

11.6.3.35 operator-() [1/5] template<typename T >

```
Polynomial<T> Util::operator- (
        Polynomial< T > const & a ) [inline]
```

Unary negation of polynomial.

Parameters

a	input polynomial

Returns

negated polynomial -a

Definition at line 706 of file Polynomial.h.

11.6.3.36 operator<<() [7/11]

`std::ostream & Util::operator<< ( std::ostream & out, Rational const & rational )`

Output stream inserter for a Rational.
Output elements of a rational to stream, without line breaks.

Parameters

out	output stream
rational	Rational to be written to stream

Returns

modified output stream

Definition at line 16 of file Rational.cpp.
References UTIL_CHECK.

11.6.3.37 operator+() [1/3]

`Rational Util::operator+ ( Rational const & a, Rational const & b ) [inline]`

Compute sum of two rationals.

Parameters

a	1st argument
b	2st argument

Returns

sum a + b

Definition at line 490 of file Rational.h.

11.6.3.38 operator+() [2/3]

`Rational Util::operator+ ( Rational const & a, int b ) [inline]`

Compute sum of rational and integer.

Parameters

a	Rational argument
b	integer argument

Returns

sum a + b

Definition at line 505 of file Rational.h.

11.6.3.39 operator+() [3/3]

```
Rational Util::operator+ (
    int b,
    Rational const & a ) [inline]
```

Compute sum of integer and integer.

Parameters

b	integer argument
a	Rational argument

Returns

sum a + b

Definition at line 519 of file Rational.h.

11.6.3.40 operator-() [2/5]

```
Rational Util::operator- (
    Rational const & a,
    Rational const & b ) [inline]
```

Compute difference of rationals.

Parameters

a	1st argument
b	2st argument

Returns

difference a - b

Definition at line 530 of file Rational.h.

11.6.3.41 operator-() [3/5]

```
Rational Util::operator- (
    Rational const & a,
    int b ) [inline]
```

Compute difference of rational and integer.

Parameters

a	Rational argument
b	integer argument

Returns

difference a - b

Definition at line 545 of file Rational.h.

11.6.3.42 operator-() [4/5] Rational Util::operator- (int b, Rational const & a) [inline]

Compute difference of integer and rational.

Parameters

b	integer argument
a	Rational argument

Returns

difference b - a

Definition at line 559 of file Rational.h.

11.6.3.43 operator*() [1/3] Rational Util::operator* (Rational const & a, Rational const & b) [inline]

Compute product of rationals.

Parameters

a	1st Rational argument
b	2st Rational argument

Returns

product a∗b

Definition at line 573 of file Rational.h.

11.6.3.44 operator*() [2/3] Rational Util::operator* (Rational const & a, int b) [inline]

Compute product of rational and integer.

Parameters

a	Rational argument
b	integer argument

Returns

product a∗b

Definition at line 588 of file Rational.h.

11.6.3.45 operator∗() [3/3]

```
Rational Util::operator* (
    int b,
    Rational const & a ) [inline]
```

Compute product of integer and rational.

Parameters

b	integer argument
a	Rational argument

Returns

product a∗b

Definition at line 599 of file Rational.h.

11.6.3.46 operator/() [1/3]

```
Rational Util::operator/ (
    Rational const & a,
    Rational const & b ) [inline]
```

Compute quotient of two rationals.

Parameters

a	1st Rational argument (numerator)
b	2st Rational argument (denominator)

Returns

ratio a/b

Definition at line 610 of file Rational.h.
References UTIL_THROW.

11.6.3.47 operator/() [2/3]

```
Rational Util::operator/ (
    Rational const & a,
    int b ) [inline]
```

Compute quotient Rational divided by integer.

Parameters

a	Rational argument (numerator)
b	integer argument (denominator)

Returns

ratio a/b

Definition at line 628 of file Rational.h.
References UTIL_THROW.

11.6.3.48 operator/() [3/3]

```
Rational Util::operator/ (
    int b,
    Rational const & a ) [inline]
```

Compute quotient integer divided by Rational.

Parameters

b	integer argument (numerator)
a	Rational argument (denominator)

Returns

ratio b/a

Definition at line 644 of file Rational.h.
References UTIL_THROW.

11.6.3.49 operator-() [5/5]

```
Rational Util::operator- (
    Rational const & a ) [inline]
```

Unary negation of Rational.

Parameters

a	Rational number

Returns

negation -a

Definition at line 661 of file Rational.h.

11.6.3.50 operator==() [2/13]

```
bool Util::operator== (
    Rational const & a,
    Rational const & b ) [inline]
```

Equality operators.
Equality operator for two Rational numbers.

Parameters

a	1st Rational
b	2nd Rational

Returns

true if equal, false otherwise

Definition at line 674 of file Rational.h.

11.6.3.51 operator==() [3/13]

```
bool Util::operator== (
    Rational const & a,
    int b ) [inline]
```

Equality operator for a Rational and an integer.

Parameters

a	Rational number
b	integer number

Returns

true if equal, false otherwise

Definition at line 684 of file Rational.h.

11.6.3.52 operator==() [4/13]

```
bool Util::operator== (
    int b,
    Rational const & a ) [inline]
```

Equality operator for an integer and a Rational.

Parameters

b	integer number
a	Rational number

Returns

true if equal, false otherwise

Definition at line 694 of file Rational.h.

11.6.3.53 operator"!=() [2/13]

```
bool Util::operator!= (
    Rational const & a,
    Rational const & b ) [inline]
```

Inequality operators.
Inequality operator for two Rational numbers.

Parameters

a	1st Rational
b	2nd Rational

Returns

true if unequal, false if equal

Definition at line 706 of file Rational.h.

11.6.3.54 operator"!=() [3/13]

```
bool Util::operator!= (
    Rational const & a,
    int b ) [inline]
```

Inequality operator for a Rational and an integer.

Parameters

a	Rational number
b	integer number

Returns

true if unequal, false if equal

Definition at line 716 of file Rational.h.

11.6.3.55 operator"!=() [4/13]

```
bool Util::operator!= (
    int b,
    Rational const & a ) [inline]
```

Inequality operator for an integer and a Rational.

Parameters

b	integer number
a	Rational number

Returns

true if unequal, false if equal

Definition at line 726 of file Rational.h.

11.6.3.56 MpiThrow()

```
void Util::MpiThrow (
    Exception & e )
```

Function to throw exception in MPI code.

If MPI is not initialized, this function writes the message and calls MPI Abort. If MPI is not initialized, it simply throws the Exception.

Parameters

e	Exception to be thrown.

Definition at line 90 of file Exception.cpp.

References Util::Log::close(), Util::Log::file(), and Util::Exception::message().

11.6.3.57 initStatic() `void Util::initStatic ( )`

Guarantee initialization of all static class members in Util namespace.
Definition at line 26 of file initStatic.cpp.
References Util::Format::initStatic(), Util::Constants::initStatic(), Util::Log::initStatic(), Util::Memory::initStatic(), Util::ParamComponent::initStatic(), Util::Tensor::initStatic(), Util::Vector::initStatic(), and Util::IntVector::initStatic().

11.6.3.58 checkRequiredIstream() `void Util::checkRequiredIstream ( std::istream & in )`

Check status of a std::istream just before reading required variable.
Throw Exception with appropriate error message if not good.

Parameters

in	input stream from which to read.

Definition at line 124 of file ioUtil.cpp.
References UTIL_THROW.

11.6.3.59 baseOffset() `template<typename D , typename B > ptrdiff_t Util::baseOffset ( D & object )`

Template for calculating offsets of base class subobjects.
Types: D - derived class B - base class
Definition at line 40 of file Offset.h.

11.6.3.60 send< bool >() `template<> void Util::send< bool > ( MPI::Comm & comm, bool & data, int dest, int tag )`

Explicit specialization of send for bool data.
Definition at line 19 of file MpiSendRecv.cpp.

11.6.3.61 recv< bool >() `template<> void Util::recv< bool > ( MPI::Comm & comm, bool & data, int source, int tag )`

Explicit specialization of recv for bool data.
Definition at line 26 of file MpiSendRecv.cpp.

11.6.3.62 bcast< bool >() `template<> void Util::bcast< bool > ( MPI::Intracomm & comm, bool & data, int root )`

Explicit specialization of bcast for bool data.
Definition at line 34 of file MpiSendRecv.cpp.
Referenced by Util::Parameter::load(), Util::ParamComposite::loadOptional(), Util::Begin::readParam(), and Util::Parameter::readParam().

11.6.3.63 send< std::string >() template<>

```
void Util::send< std::string > (
            MPI::Comm & comm,
            std::string & data,
            int dest,
            int tag )
```

Explicit specialization of send for std::string data.
Definition at line 48 of file MpiSendRecv.cpp.

11.6.3.64 recv< std::string >() template<>

```
void Util::recv< std::string > (
            MPI::Comm & comm,
            std::string & data,
            int source,
            int tag )
```

Explicit specialization of recv for std::string data.
Definition at line 64 of file MpiSendRecv.cpp.

11.6.3.65 bcast< std::string >() template<>

```
void Util::bcast< std::string > (
            MPI::Intracomm & comm,
            std::string & data,
            int root )
```

Explicit specialization of bcast for std::string data.
Definition at line 80 of file MpiSendRecv.cpp.

11.6.3.66 send() [1/4] template<typename T >

```
void Util::send (
            MPI::Comm & comm,
            T & data,
            int dest,
            int tag )
```

Send a single T value.
Throws an Exception if no associated MPI data type is available, i.e., if MpiTraits<T>::hasType is false.

Parameters

comm	MPI communicator
data	value
dest	MPI rank of receiving processor in comm
tag	user-defined integer identifier for message

Definition at line 97 of file MpiSendRecv.h.

References UTIL_THROW.

11.6.3.67 recv() [1/4] template<typename T >

```
void Util::recv (
            MPI::Comm & comm,
            T & data,
            int source,
            int tag )
```

Receive a single T value.
Throws an Exception if no associated MPI data type is available, i.e., if MpiTraits<T>::hasType is false.

Parameters

comm	MPI communicator
data	value
source	MPI rank of sending processor in comm
tag	user-defined integer identifier for message

Definition at line 116 of file MpiSendRecv.h.
References UTIL_THROW.

11.6.3.68 bcast() [1/4] template<typename T >

```
void Util::bcast (
            MPI::Intracomm & comm,
            T & data,
            int root )
```

Broadcast a single T value.
Throws an Exception if no associated MPI data type is available, i.e., if MpiTraits<T>::hasType is false.

Parameters

comm	MPI communicator
data	value
root	MPI rank of root (sending) processor in comm

Definition at line 134 of file MpiSendRecv.h.
References UTIL_THROW.

11.6.3.69 send() [2/4] template<typename T >

```
void Util::send (
            MPI::Comm & comm,
            T * array,
            int count,
            int dest,
            int tag )
```

Send a C-array of T values.
Throws an exception if their exists neither an associated MPI data type nor an explicit specialization of the scalar send<T>.

Parameters

comm	MPI communicator
array	address of first element in array
count	number of elements in array
dest	MPI rank of destination (receiving) processor in comm
tag	user-defined integer identifier for this message

Definition at line 156 of file MpiSendRecv.h.

11.6.3.70 recv() [2/4] template<typename T >

```
void Util::recv (
            MPI::Comm & comm,
            T * array,
            int count,
            int source,
            int tag )
```

Receive a C-array of T objects.

Throws an exception if their exists neither an associated MPI data type nor an explicit specialization of the scalar recv<T>.

Parameters

comm	MPI communicator
array	address of first element in array
count	number of elements in array
source	MPI rank of source (sending) processor in comm
tag	user-defined integer identifier for this message

Definition at line 182 of file MpiSendRecv.h.

11.6.3.71 bcast() [2/4] template<typename T >

```
void Util::bcast (
            MPI::Intracomm & comm,
            T * array,
            int count,
            int root )
```

Broadcast a C-array of T objects.

Throws an exception if their exists neither an associated MPI data type nor an explicit specialization of the scalar bcast<T>.

Parameters

comm	MPI communicator
array	address of first element in array
count	number of elements in array
root	MPI rank of root (sending) processor in comm

Definition at line 207 of file MpiSendRecv.h.

11.6.3.72 send() [3/4] template<typename T >

```
void Util::send (
            MPI::Comm & comm,
            DArray< T > & array,
            int count,
            int dest,
            int tag )
```

Send a DArray<T> container.

Throws an exception if their exists neither an associated MPI data type nor an explicit specialization of the scalar send<T> method.

Parameters

comm	MPI communicator
array	DArray object
count	logical number of elements in array
dest	MPI rank of destination (receiving) processor in comm
tag	user-defined integer identifier for this message

Definition at line 235 of file MpiSendRecv.h.

References Util::Array< Data >::capacity(), Util::DArray< Data >::isAllocated(), and UTIL_THROW.

11.6.3.73 recv() [3/4] template<typename T >

```
void Util::recv (
            MPI::Comm & comm,
            DArray< T > & array,
            int count,
            int source,
            int tag )
```

Receive a DArray<T> container.

Throws an exception if their exists neither an associated MPI data type nor an explicit specialization of the scalar recv<T> method.

Parameters

comm	MPI communicator
array	DArray object
count	logical number of elements in array
source	MPI rank of source (sending) processor in comm
tag	user-defined integer identifier for this message

Definition at line 269 of file MpiSendRecv.h.

References Util::Array< Data >::capacity(), Util::DArray< Data >::isAllocated(), and UTIL_THROW.

11.6.3.74 bcast() [3/4] template<typename T >

```
void Util::bcast (
            MPI::Intracomm & comm,
            DArray< T > & array,
```

```
int count,
int root )
```

Broadcast a DArray<T> container.

Throws an exception if their exists neither an associated MPI data type nor an explicit specialization of the scalar bcast<T>.

Parameters

comm	MPI communicator
array	address of first element in array
count	number of elements in array
root	MPI rank of root (sending) processor in comm

Definition at line 302 of file MpiSendRecv.h.

References Util::Array< Data >::capacity(), Util::DArray< Data >::isAllocated(), and UTIL_THROW.

11.6.3.75 send() [4/4] template<typename T >

```
void Util::send (
    MPI::Comm & comm,
    DMatrix< T > & matrix,
    int m,
    int n,
    int dest,
    int tag )
```

Send a DMatrix<T> container.

Throws an exception if their exists neither an associated MPI data type nor an explicit specialization of the scalar send<T>.

Parameters

comm	MPI communicator
matrix	DMatrix object to send
m	logical number of rows in matrix
n	logical number of columns in matrix
dest	MPI rank of destination (receiving) processor in comm
tag	user-defined integer identifier for this message

Definition at line 339 of file MpiSendRecv.h.

References Util::Matrix< Data >::capacity1(), Util::Matrix< Data >::capacity2(), Util::DMatrix< Data >::isAllocated(), and UTIL_THROW.

11.6.3.76 recv() [4/4] template<typename T >

```
void Util::recv (
    MPI::Comm & comm,
    DMatrix< T > & matrix,
    int m,
    int n,
    int source,
    int tag )
```

Receive a DMatrix<T> container.

Throws an exception if their exists neither an associated MPI data type nor an explicit specialization of the scalar recv<T>.

Parameters

comm	MPI communicator
matrix	DMatrix object to receive
m	logical number of rows in matrix
n	logical number of columns in matrix
source	MPI rank of source (sending) processor in comm
tag	user-defined integer identifier for this message

Definition at line 383 of file MpiSendRecv.h.
References Util::Matrix< Data >::capacity1(), Util::Matrix< Data >::capacity2(), Util::DMatrix< Data >::isAllocated(), and UTIL_THROW.

11.6.3.77 bcast() [4/4]

```
template<typename T >
void Util::bcast (
        MPI::Intracomm & comm,
        DMatrix< T > & matrix,
        int m,
        int n,
        int root )
```

Broadcast a DMatrix<T> container.
Throws an exception if their exists neither an associated MPI data type nor an explicit specialization of the scalar bcast<T>.

Parameters

comm	MPI communicator
matrix	DMatrix object
m	logical number of rows in matrix
n	logical number of columns in matrix
root	MPI rank of root (sending) processor in comm

Definition at line 427 of file MpiSendRecv.h.
References Util::Matrix< Data >::capacity1(), Util::Matrix< Data >::capacity2(), Util::DMatrix< Data >::isAllocated(), and UTIL_THROW.

11.6.3.78 operator>>() [7/10]

```
std::istream & Util::operator>> (
        std::istream & in,
        Label label )
```

Extractor for Label.

Parameters

in	input stream
label	Label to be read from file

Definition at line 104 of file Label.cpp.
References Util::Log::file(), Util::Label::isRequired(), UTIL_CHECK, and UTIL_THROW.

11.6.3.79 operator<<() [8/11] std::ostream & Util::operator<< (

```
std::ostream & out,
Label label )
```

Inserter for Label.

Parameters

out	output stream
label	Label to be written to file

Definition at line 158 of file Label.cpp.
References Util::Label::LabelWidth.

11.6.3.80 isNull() [1/3] template<typename T >

```
bool Util::isNull (
FlexPtr< T > p ) [inline]
```

Return true iff the enclosed built-in pointer is null.
Definition at line 143 of file FlexPtr.h.
References Util::FlexPtr< T >::get().

11.6.3.81 isNull() [2/3] template<typename T >

```
bool Util::isNull (
T * ptr ) [inline]
```

Return true iff a built-in pointer is null.
Definition at line 18 of file isNull.h.

11.6.3.82 isNull() [3/3] template<typename T >

```
bool Util::isNull (
ScopedPtr< T > p ) [inline]
```

Return true iff the enclosed built-in pointer is null.
Definition at line 90 of file ScopedPtr.h.
References Util::ScopedPtr< T >::get().

11.6.3.83 operator==() [5/13] bool Util::operator== (

```
const IntVector & v1,
const IntVector & v2 )
```

Equality for IntVectors.
Definition at line 24 of file IntVector.cpp.
References Dimension.

11.6.3.84 operator==() [6/13] bool Util::operator== (

```
const IntVector & v1,
const int * v2 )
```

Equality of IntVector and C array.
Definition at line 35 of file IntVector.cpp.
References Dimension.

11.6.3.85 operator==() [7/13]

```
bool Util::operator== (
    const int * v1,
    const IntVector & v2 )
```

Equality of C array and IntVector.
Definition at line 45 of file IntVector.cpp.

11.6.3.86 operator"!=() [5/13]

```
bool Util::operator!= (
    const IntVector & v1,
    const IntVector & v2 )
```

Inequality of two IntVectors.
Definition at line 50 of file IntVector.cpp.

11.6.3.87 operator"!=() [6/13]

```
bool Util::operator!= (
    const IntVector & v1,
    const int * v2 )
```

Inequality of IntVector and C array.
Definition at line 54 of file IntVector.cpp.

11.6.3.88 operator"!=() [7/13]

```
bool Util::operator!= (
    const int * v1,
    const IntVector & v2 )
```

Inequality of C array and IntVector.
Definition at line 58 of file IntVector.cpp.

11.6.3.89 operator>>() [8/10]

```
std::istream & Util::operator>> (
    std::istream & in,
    IntVector & vector )
```

istream extractor for a IntVector.
Input elements of a vector from stream, without line breaks.

Parameters

in	input stream
vector	IntVector to be read from stream

Returns

modified input stream

Definition at line 64 of file IntVector.cpp.
References Dimension.

11.6.3.90 operator<<() [9/11]

```
std::ostream & Util::operator<< (
```

```
std::ostream & out,
const IntVector & vector )
```

ostream inserter for a IntVector.
Output elements of a vector to stream, without line breaks.

Parameters

out	output stream
vector	IntVector to be written to stream

Returns

modified output stream

Definition at line 75 of file IntVector.cpp.
References Dimension.

11.6.3.91 operator==() [8/13]

```
bool Util::operator== (
const Tensor & t1,
const Tensor & t2 )
```

Equality for Tensors.
Definition at line 43 of file Tensor.cpp.
References DimensionSq.

11.6.3.92 operator==() [9/13]

```
bool Util::operator== (
const Tensor & t1,
const double a2[ ][Dimension] )
```

Equality of Tensor and 2D C array.
Definition at line 56 of file Tensor.cpp.
References Dimension.

11.6.3.93 operator==() [10/13]

```
bool Util::operator== (
const double a1[ ][Dimension],
const Tensor & t2 )
```

Equality of C array and Tensor.
Definition at line 71 of file Tensor.cpp.

11.6.3.94 operator"!=() [8/13]

```
bool Util::operator!= (
const Tensor & t1,
const Tensor & t2 )
```

Negation of t1 == t2 (tensors t1 and t2)
Inequality of two Tensors.
Definition at line 79 of file Tensor.cpp.

11.6.3.95 operator"!=() [9/13]

```
bool Util::operator!= (
const Tensor & t1,
const double a2[ ][Dimension] )
```

Negation of t1 == a2 (tensor t1, 2D array a2)

Inequality of Tensor and C array.
Definition at line 83 of file Tensor.cpp.

11.6.3.96 operator"!=() [10/13]

```
bool Util::operator!= (
    const double a1[ ][Dimension],
    const Tensor & t2 )
```

Negation of t1 == a2 (tensor t2, 2D array a1)
Inequality of C array and Tensor.
Definition at line 87 of file Tensor.cpp.

11.6.3.97 operator>>() [9/10]

```
std::istream & Util::operator>> (
    std::istream & in,
    Tensor & tensor )
```

istream extractor for a Tensor.
Input elements of a tensor from stream, without line breaks.

Parameters

in	input stream
tensor	Tensor to be read from stream

Returns

modified input stream

Definition at line 93 of file Tensor.cpp.
References DimensionSq.

11.6.3.98 operator<<() [10/11]

```
std::ostream & Util::operator<< (
    std::ostream & out,
    const Tensor & tensor )
```

ostream inserter for a Tensor.
Output elements of a tensor to stream, without line breaks.

Parameters

out	output stream
tensor	Tensor to be written to stream

Returns

modified output stream

Definition at line 104 of file Tensor.cpp.
References DimensionSq.

11.6.3.99 operator==() [11/13]

```
bool Util::operator== (
    const Vector & v1,
    const Vector & v2 )
```

Equality for Vectors.
Definition at line 26 of file Vector.cpp.
References Dimension.

11.6.3.100 operator==() [12/13]

```
bool Util::operator== (
    const Vector & v1,
    const double * v2 )
```

Equality of Vector and C array.
Definition at line 36 of file Vector.cpp.
References Dimension.

11.6.3.101 operator==() [13/13]

```
bool Util::operator== (
    const double * v1,
    const Vector & v2 )
```

Equality of C array and Vector.
Definition at line 48 of file Vector.cpp.

11.6.3.102 operator"!=() [11/13]

```
bool Util::operator!= (
    const Vector & v1,
    const Vector & v2 )
```

Inequality of two Vectors.
Definition at line 53 of file Vector.cpp.

11.6.3.103 operator"!=() [12/13]

```
bool Util::operator!= (
    const Vector & v1,
    const double * v2 )
```

Inequality of Vector and C array.
Definition at line 56 of file Vector.cpp.

11.6.3.104 operator"!=() [13/13]

```
bool Util::operator!= (
    const double * v1,
    const Vector & v2 )
```

Inequality of C array and Vector.
Definition at line 59 of file Vector.cpp.

11.6.3.105 operator>>() [10/10]

```
std::istream & Util::operator>> (
    std::istream & in,
    Vector & vector )
```

istream extractor for a Vector.
Input elements of a vector from stream, without line breaks.

Parameters

in	input stream
vector	Vector to be read from stream

Returns

modified input stream

Definition at line 65 of file Vector.cpp.
References Dimension.

11.6.3.106 operator<<() [11/11] `std::ostream & Util::operator<< ( std::ostream & out, const Vector & vector )`

ostream inserter for a Vector.
Output elements of a vector to stream, without line breaks.

Parameters

out	output stream
vector	Vector to be written to stream

Returns

modified output stream

Definition at line 76 of file Vector.cpp.
References Dimension.

12 Class Documentation

12.1 CommandLine Class Reference

Abstraction of a C array of command line arguments.
`#include <CommandLine.h>`

Public Member Functions

- CommandLine ()
  Constructor.
- void append (const char ∗arg)
  Add a command line argument string.
- void clear ()
  Clear all arguments.
- int argc ()
  Return number of command line arguments.
- char ∗∗ argv ()
  Return array of C-string command line argument strings.

12.1.1 Detailed Description

Abstraction of a C array of command line arguments.
Definition at line 21 of file CommandLine.h.

12.1.2 Constructor & Destructor Documentation

12.1.2.1 CommandLine() `CommandLine::CommandLine ( ) [inline]`

Constructor.
Definition at line 29 of file CommandLine.h.
References clear().

12.1.3 Member Function Documentation

12.1.3.1 append() `void CommandLine::append ( const char * arg ) [inline]`

Add a command line argument string.
Definition at line 35 of file CommandLine.h.
Referenced by clear().

12.1.3.2 clear() `void CommandLine::clear ( ) [inline]`

Clear all arguments.
Definition at line 41 of file CommandLine.h.
References append().
Referenced by CommandLine().

12.1.3.3 argc() `int CommandLine::argc ( ) [inline]`

Return number of command line arguments.
Definition at line 54 of file CommandLine.h.

12.1.3.4 argv() `char** CommandLine::argv ( ) [inline]`

Return array of C-string command line argument strings.

Returns

pointer to C array of null terminated C strings.

Definition at line 62 of file CommandLine.h.
The documentation for this class was generated from the following file:

- CommandLine.h

12.2 CompositeTestRunner Class Reference

A TestRunner comprised of one or more child TestRunners.
#include <CompositeTestRunner.h>
Inheritance diagram for CompositeTestRunner:

classCompositeTestRunner-eps-converted-to.pdf

Public Member Functions

- virtual ~CompositeTestRunner ()

  Destructor.
- void addChild (TestRunner &child)

  Add an existing TestRunner as a child.
- void addChild (TestRunner ∗childPtr)

  Add a TestRunner as a child, and accept ownership.
- void addChild (TestRunner ∗childPtr, const std::string &prefix)

  Add a TestRunner as a child, accept ownership, and initialize filePrefix.
- virtual void addFilePrefix (const std::string &prefix)

  Prepend argument prefix to existing filePrefix.
- virtual int run ()

  Run all children in sequence, using depth-first recursion.

Additional Inherited Members

12.2.1 Detailed Description

A TestRunner comprised of one or more child TestRunners.
Definition at line 19 of file CompositeTestRunner.h.

12.2.2 Constructor & Destructor Documentation

12.2.2.1 ~CompositeTestRunner() `CompositeTestRunner::~CompositeTestRunner ( ) [virtual]`

Destructor.
Definition at line 97 of file CompositeTestRunner.h.

12.2.3 Member Function Documentation

12.2.3.1 addChild() [1/3] `void CompositeTestRunner::addChild ( TestRunner & child )`

Add an existing TestRunner as a child.
Children added by this method are not destroyed by the parent CompositeTestRunner destructor.

Parameters

child	enclosed TestRunner object

Definition at line 108 of file CompositeTestRunner.h.
References TestRunner::setParent().
Referenced by addChild().

12.2.3.2 addChild() [2/3] `void CompositeTestRunner::addChild ( TestRunner * childPtr )`

Add a TestRunner as a child, and accept ownership.
Children added by this method are owned by the parent CompositeTestRunner, and so are destroyed by its destructor.

Parameters

childPtr	pointer to child TestRunner

Definition at line 117 of file CompositeTestRunner.h.
References TestRunner::setParent().

12.2.3.3 addChild() [3/3] `void CompositeTestRunner::addChild ( TestRunner * childPtr, const std::string & prefix )`

Add a TestRunner as a child, accept ownership, and initialize filePrefix.
Children added by this method are owned by the parent CompositeTestRunner, and so are destroyed by its destructor. The file prefix argument should normally be a path for a particular child defined relative to any common prefix used by all tests in this composite. The common prefix can then be prepended by calling addFilePrefix at run time.

Parameters

childPtr	pointer to child TestRunner
prefix	prefix to append to file names in all descendants

Definition at line 127 of file CompositeTestRunner.h.
References addChild(), and TestRunner::addFilePrefix().

12.2.3.4 addFilePrefix() `void CompositeTestRunner::addFilePrefix ( const std::string & prefix ) [virtual]`

Prepend argument prefix to existing filePrefix.
This function also prepends prefix to all children. If this function is called at run-time for the highest level composite in a hierarchy, the prefix is thus propagated to all TestRunners in the hierarchy, and thus also used in the methods of UnitTest that are used to open files.

Parameters

prefix	string to prepend to existing filePrefix.

Reimplemented from TestRunner.
Definition at line 137 of file CompositeTestRunner.h.
References TestRunner::addFilePrefix().

12.2.3.5 run() `int CompositeTestRunner::run ( ) [virtual]`

Run all children in sequence, using depth-first recursion.
Implements TestRunner.
Definition at line 148 of file CompositeTestRunner.h.
References TestRunner::nFailure(), and TestRunner::report().
The documentation for this class was generated from the following file:

- CompositeTestRunner.h

12.3 FCT< D > Class Template Reference

12.3.1 Detailed Description

template< int D >
class FCT< D >

Definition at line 71 of file FCT.h.
The documentation for this class was generated from the following files:

- FCT.h
- FCT.tpp

12.4 ParamFileTest Class Reference

A UnitTest with a built-in input file.
`#include <ParamFileTest.h>`
Inheritance diagram for ParamFileTest:

classParamFileTest-eps-converted-to.pdf

Public Member Functions

- ParamFileTest ()
  Constructor.
- ~ParamFileTest ()
  Destructor.
- virtual void tearDown ()
  Close the input file.
- void openFile (const char ∗fileName)
  Open the input file.
- void closeFile ()
  Close the input file.
- std::ifstream & file ()
  Returns input file by reference.

Additional Inherited Members

12.4.1 Detailed Description

A UnitTest with a built-in input file.
Definition at line 15 of file ParamFileTest.h.

12.4.2 Constructor & Destructor Documentation

12.4.2.1 ParamFileTest() `ParamFileTest::ParamFileTest ( ) [inline]`

Constructor.
Definition at line 23 of file ParamFileTest.h.

12.4.2.2 ∼ParamFileTest() `ParamFileTest::∼ParamFileTest ( ) [inline]`

Destructor.

Definition at line 30 of file ParamFileTest.h.

References closeFile().

12.4.3 Member Function Documentation

12.4.3.1 tearDown() `virtual void ParamFileTest::tearDown ( ) [inline], [virtual]`

Close the input file.

Reimplemented from UnitTest.

Definition at line 36 of file ParamFileTest.h.

References closeFile().

12.4.3.2 openFile() `void ParamFileTest::openFile ( const char * fileName ) [inline]`

Open the input file.

Definition at line 42 of file ParamFileTest.h.

References UnitTest::isIoProcessor(), and UnitTest::openInputFile().

12.4.3.3 closeFile() `void ParamFileTest::closeFile ( ) [inline]`

Close the input file.

Definition at line 52 of file ParamFileTest.h.

Referenced by tearDown(), and ∼ParamFileTest().

12.4.3.4 file() `std::ifstream& ParamFileTest::file ( ) [inline]`

Returns input file by reference.

Definition at line 58 of file ParamFileTest.h.

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

- ParamFileTest.h

12.5 Pscf::Basis< D > Class Template Reference

Symmetry-adapted basis for pseudo-spectral scft.

`#include <Basis.h>`

Classes

- class Star

  List of wavevectors that are related by space-group symmetries.
- class Wave

  Wavevector used to construct a basis function.

Public Member Functions

- Basis ()

  Default constructor.
- ∼Basis ()

Destructor.

- void makeBasis (const Mesh< D > &mesh, const UnitCell< D > &unitCell, std::string groupName)

  Construct basis for a specific grid and space group.
- void makeBasis (const Mesh< D > &mesh, const UnitCell< D > &unitCell, const SpaceGroup< D > &group)

  Construct basis for a specific grid and space group.
- void update ()

  Update values after change in unit cell parameters.
- void outputWaves (std::ostream &out, bool outputAll=false) const

  Print a list of all waves to an output stream.
- void outputStars (std::ostream &out, bool outputAll=false) const

  Print a list of all stars to an output stream.
- bool isValid () const

  Returns true if valid, false otherwise.
- int nWave () const

  Total number of wavevectors.
- int nBasisWave () const

  Total number of wavevectors in uncancelled stars.
- int nStar () const

  Total number of stars.
- int nBasis () const

  Total number of nonzero symmetry-adapted basis functions.
- const Star & star (int i) const

  Get a Star, access by integer index.
- const Wave & wave (int i) const

  Get a specific Wave, access by integer index.
- int waveId (IntVec< D > vector) const

  Get integer index of a Wave.

12.5.1 Detailed Description

template<int D>
class Pscf::Basis< D >

Symmetry-adapted basis for pseudo-spectral scft.
Definition at line 27 of file Basis.h.

12.5.2 Constructor & Destructor Documentation

12.5.2.1 Basis() `template<int D>`

`Pscf::Basis< D >::Basis`
Default constructor.
Definition at line 27 of file Basis.tpp.

12.5.2.2 ∼Basis() `template<int D>`

`Pscf::Basis< D >::~Basis`
Destructor.
Definition at line 39 of file Basis.tpp.

12.5.3 Member Function Documentation

12.5.3.1 makeBasis() [1/2]

```
template<int D>
void Pscf::Basis< D >::makeBasis (
            const Mesh< D > & mesh,
            const UnitCell< D > & unitCell,
            std::string groupName )
```

Construct basis for a specific grid and space group.

Proposal: Initially implementation functions correctly only for identity group, withgroupName == 'I'.

Definition at line 46 of file Basis.tpp.

References Util::Log::file(), Pscf::SymmetryGroup< SpaceSymmetry< D > >::makeCompleteGroup(), Pscf::make↩ GroupFileName(), UTIL_CHECK, and UTIL_THROW.

12.5.3.2 makeBasis() [2/2]

```
template<int D>
void Pscf::Basis< D >::makeBasis (
            const Mesh< D > & mesh,
            const UnitCell< D > & unitCell,
            const SpaceGroup< D > & group )
```

Construct basis for a specific grid and space group.

Definition at line 87 of file Basis.tpp.

References Pscf::Mesh< D >::size(), and UTIL_THROW.

12.5.3.3 update()

```
template<int D>
void Pscf::Basis< D >::update
```

Update values after change in unit cell parameters.

Definition at line 647 of file Basis.tpp.

12.5.3.4 outputWaves()

```
template<int D>
void Pscf::Basis< D >::outputWaves (
            std::ostream & out,
            bool outputAll = false ) const
```

Print a list of all waves to an output stream.

Parameters

out	output stream to which to write
outputAll	output cancelled waves only if true

Definition at line 671 of file Basis.tpp.

12.5.3.5 outputStars()

```
template<int D>
void Pscf::Basis< D >::outputStars (
            std::ostream & out,
            bool outputAll = false ) const
```

Print a list of all stars to an output stream.

Parameters

out	output stream to which to write
outputAll	output cancelled waves only if true

Definition at line 706 of file Basis.tpp.

12.5.3.6 isValid() template<int D>

bool Pscf::Basis< D >::isValid

Returns true if valid, false otherwise.

Definition at line 743 of file Basis.tpp.

References Pscf::MeshIterator< D >::atEnd(), Pscf::MeshIterator< D >::begin(), Pscf::Vec< D, T >::negate(), and Pscf::MeshIterator< D >::position().

12.5.3.7 nWave() template<int D>

int Pscf::Basis< D >::nWave [inline]

Total number of wavevectors.

Definition at line 278 of file Basis.h.

12.5.3.8 nBasisWave() template<int D>

int Pscf::Basis< D >::nBasisWave [inline]

Total number of wavevectors in uncancelled stars.

Definition at line 282 of file Basis.h.

12.5.3.9 nStar() template<int D>

int Pscf::Basis< D >::nStar [inline]

Total number of stars.

Definition at line 286 of file Basis.h.

12.5.3.10 nBasis() template<int D>

int Pscf::Basis< D >::nBasis

Total number of nonzero symmetry-adapted basis functions.

Definition at line 666 of file Basis.tpp.

12.5.3.11 star() template<int D>

const Basis< D >::Star & Pscf::Basis< D >::star (
 int *i*) const [inline]

Get a Star, access by integer index.

Definition at line 296 of file Basis.h.

12.5.3.12 wave() template<int D>

const Basis< D >::Wave & Pscf::Basis< D >::wave (
 int *i*) const [inline]

Get a specific Wave, access by integer index.

Definition at line 291 of file Basis.h.

12.5.3.13 waveId() `template<int D>`

`int Pscf::Basis< D >::waveId ( IntVec< D > vector ) const`

Get integer index of a Wave.
Definition at line 300 of file Basis.h.
The documentation for this class was generated from the following files:

- Basis.h
- Basis.tpp

12.6 Pscf::Basis< D >::Star Class Reference

List of wavevectors that are related by space-group symmetries.
`#include <Basis.h>`

Public Attributes

- double eigen

  Eigenvalue of negative Laplacian for this star.
- int size

  Number of wavevectors in the star.
- int beginId

  Wave index of first wavevector in star.
- int endId

  Wave index of last wavevector in star.
- int invertFlag

  Index for inversion symmetry of star.
- IntVec< D > waveBz

  Integer indices of characteristic wave of this star.
- bool cancel

  Is this star cancelled, i.e., associated with a zero function?

12.6.1 Detailed Description

template< int D>
class Pscf::Basis< D >::Star

List of wavevectors that are related by space-group symmetries.
The indices of the wavevectors in a star form a continuous block. Within this block, waves are listed in descending lexigraphical order of their integer (ijk) indices, with more signficant indices listed first.
Definition at line 68 of file Basis.h.

12.6.2 Member Data Documentation

12.6.2.1 eigen `template<int D>`

`double Pscf::Basis< D >::Star::eigen`

Eigenvalue of negative Laplacian for this star.
Equal to square norm of any wavevector in this star.
Definition at line 78 of file Basis.h.

12.6.2.2 size template<int D>

`int Pscf::Basis< D >::Star::size`

Number of wavevectors in the star.
Definition at line 83 of file Basis.h.
Referenced by Pscf::Pspg::FieldIo< D >::convertKGridToBasis().

12.6.2.3 beginId template<int D>

`int Pscf::Basis< D >::Star::beginId`

Wave index of first wavevector in star.
Definition at line 88 of file Basis.h.
Referenced by Pscf::Pspg::FieldIo< D >::convertBasisToKGrid(), and Pscf::Pspg::FieldIo< D >::convertKGridTo←Basis().

12.6.2.4 endId template<int D>

`int Pscf::Basis< D >::Star::endId`

Wave index of last wavevector in star.
Definition at line 93 of file Basis.h.
Referenced by Pscf::Pspg::FieldIo< D >::convertKGridToBasis().

12.6.2.5 invertFlag template<int D>

`int Pscf::Basis< D >::Star::invertFlag`

Index for inversion symmetry of star.
A star is said to be closed under inversion iff, for each vector G in the star, -G is also in the star. If a star S is not closed under inversion, then there is another star S' that is related to S by inversion, i.e., such that for each G in S, -G is in S'. Stars that are related by inversion are listed consecutively.
If a star is closed under inversion, then invertFlag = 0.
If a star is not closed under inversion, then invertFlag = +1 or -1, with inverFlag = +1 for the first star in the pair of stars related by inversion and invertFlag = -1 for the second.
In a centro-symmetric group, all stars are closed under inversion. In a non-centro-symmetric group, some stars may still be closed under inversion.
Definition at line 114 of file Basis.h.
Referenced by Pscf::Pspg::FieldIo< D >::convertBasisToKGrid(), and Pscf::Pspg::FieldIo< D >::convertKGridTo←Basis().

12.6.2.6 waveBz template<int D>

`IntVec<D> Pscf::Basis< D >::Star::waveBz`

Integer indices of characteristic wave of this star.
Wave given here is in or on boundary of first Brillouin zone. As a result, computing the norm of this wave must yield eigen. For invertFlag = 0 or 1, this is the first wave in the star. For invertFlag = -1, this is the last wave in the star.
Definition at line 124 of file Basis.h.

12.6.2.7 cancel template<int D>

`bool Pscf::Basis< D >::Star::cancel`

Is this star cancelled, i.e., associated with a zero function?
The cancel flag is true iff there is no nonzero basis function associated with this star.
Definition at line 132 of file Basis.h.
Referenced by Pscf::Pspg::FieldIo< D >::convertBasisToKGrid(), and Pscf::Pspg::FieldIo< D >::convertKGridTo←Basis().

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

- Basis.h

12.7 Pscf::Basis< D >::Wave Class Reference

Wavevector used to construct a basis function.
`#include <Basis.h>`

12.7.1 Detailed Description

template<int D>
class Pscf::Basis< D >::Wave

Wavevector used to construct a basis function.
Definition at line 35 of file Basis.h.
The documentation for this class was generated from the following file:

- Basis.h

12.8 Pscf::BlockDescriptor Class Reference

A linear homopolymer block within a block copolymer.
`#include <BlockDescriptor.h>`
Inheritance diagram for Pscf::BlockDescriptor:

classPscf_1_1BlockDescriptor-eps-converted-to.pdf

Public Member Functions

- BlockDescriptor ()

  Constructor.
- template<class Archive >
  void serialize (Archive &ar, unsigned int versionId)

  Serialize to/from archive.

Setters

- void setId (int id)

  Set the id for this block.
- void setVertexIds (int vertexAId, int vertexBId)

  Set indices of associated vertices.
- void setMonomerId (int monomerId)

  Set the monomer id.
- virtual void setLength (double length)

  Set the length of this block.

Accessors (getters)

- int id () const
  Get the id of this block.
- int monomerId () const
  Get the monomer type id.
- const Pair< int > & vertexIds () const
  Get the pair of associated vertex ids.
- int vertexId (int i) const
  Get id of an associated vertex.
- double length () const
  Get the length (number of monomers) in this block.
- std::istream & operator>> (std::istream &in, BlockDescriptor &block)
  istream extractor for a BlockDescriptor.
- std::ostream & operator<< (std::ostream &out, const BlockDescriptor &block)
  ostream inserter for a BlockDescriptor.

12.8.1 Detailed Description

A linear homopolymer block within a block copolymer.
(continuous Gaussian chain)
Definition at line 26 of file BlockDescriptor.h.

12.8.2 Constructor & Destructor Documentation

12.8.2.1 BlockDescriptor() `Pscf::BlockDescriptor::BlockDescriptor ( )`

Constructor.
Definition at line 16 of file BlockDescriptor.cpp.

12.8.3 Member Function Documentation

12.8.3.1 serialize() `template<class Archive >`

```
void Pscf::BlockDescriptor::serialize (
            Archive & ar,
            unsigned int versionId )
```

Serialize to/from archive.

Parameters

ar	input or output Archive
version←Id	archive format version index

Definition at line 187 of file BlockDescriptor.h.

12.8.3.2 setId() `void Pscf::BlockDescriptor::setId (`

```
            int id )
```

Set the id for this block.

Parameters

id	integer index for this block

Definition at line 26 of file BlockDescriptor.cpp.
References id().

12.8.3.3 setVertexIds() `void Pscf::BlockDescriptor::setVertexIds ( int vertexAId, int vertexBId )`

Set indices of associated vertices.

Parameters

vertex←↩ AId	integer id of vertex A
vertex←↩ BId	integer id of vertex B

Definition at line 32 of file BlockDescriptor.cpp.

12.8.3.4 setMonomerId() `void Pscf::BlockDescriptor::setMonomerId ( int monomerId )`

Set the monomer id.

Parameters

monomer←↩ Id	integer id of monomer type (>=0)

Definition at line 41 of file BlockDescriptor.cpp.
References monomerId().

12.8.3.5 setLength() `void Pscf::BlockDescriptor::setLength ( double length ) [virtual]`

Set the length of this block.
The "length" is steric volume / reference volume.

Parameters

length	block length (number of monomers).

Definition at line 47 of file BlockDescriptor.cpp.
References length().

12.8.3.6 id() `int Pscf::BlockDescriptor::id ( ) const [inline]`

Get the id of this block.
Definition at line 156 of file BlockDescriptor.h.
Referenced by Pscf::Vertex::addBlock(), and setId().

12.8.3.7 monomerId() `int Pscf::BlockDescriptor::monomerId ( ) const [inline]`

Get the monomer type id.
Definition at line 162 of file BlockDescriptor.h.
Referenced by setMonomerId().

12.8.3.8 vertexIds() `const Pair< int > & Pscf::BlockDescriptor::vertexIds ( ) const [inline]`

Get the pair of associated vertex ids.
Definition at line 168 of file BlockDescriptor.h.

12.8.3.9 vertexId() `int Pscf::BlockDescriptor::vertexId ( int i ) const [inline]`

Get id of an associated vertex.

Parameters

i	index of vertex (0 or 1)

Definition at line 174 of file BlockDescriptor.h.
Referenced by Pscf::Vertex::addBlock().

12.8.3.10 length() `double Pscf::BlockDescriptor::length ( ) const [inline]`

Get the length (number of monomers) in this block.
Definition at line 180 of file BlockDescriptor.h.
Referenced by setLength().

12.8.4 Friends And Related Function Documentation

12.8.4.1 operator>> `std::istream& operator>> ( std::istream & in, BlockDescriptor & block ) [friend]`

istream extractor for a BlockDescriptor.

Parameters

in	input stream
block	BlockDescriptor to be read from stream

Returns

modified input stream

Definition at line 53 of file BlockDescriptor.cpp.

12.8.4.2 operator<<

`std::ostream& operator<< ( std::ostream & out, const BlockDescriptor & block ) [friend]`

ostream inserter for a BlockDescriptor.

Parameters

out	output stream
block	BlockDescriptor to be written to stream

Returns

modified output stream

Definition at line 66 of file BlockDescriptor.cpp.

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

- BlockDescriptor.h
- BlockDescriptor.cpp

12.9 Pscf::BlockTmpl< TP > Class Template Reference

Class template for a block in a block copolymer.

`#include <BlockTmpl.h>`

Inheritance diagram for Pscf::BlockTmpl< TP >:

classPscf_1_1BlockTmpl-eps-converted-to.pdf

Public Member Functions

- BlockTmpl ()

    Constructor.
- virtual ~BlockTmpl ()

    Destructor.
- virtual void setKuhn (double kuhn)

    Set monomer statistical segment length.
- TP & propagator (int directionId)

    Get a Propagator for a specified direction.
- const TP & propagator (int directionId) const

    Get a const Propagator for a specified direction.
- TP::CField & cField ()

    Get the associated monomer concentration field.
- double kuhn () const

    Get monomer statistical segment length.

12.9.1 Detailed Description

template<class TP>
class Pscf::BlockTmpl< TP >

Class template for a block in a block copolymer.
Class TP is a concrete propagator class. A BlockTmpl<TP> object has:

- two TP propagator objects, one per direction
- a single monomer concentration field
- a single kuhn length

Each implementation of self-consistent field theory (SCFT) is defined in a different sub-namespace of Pscf. Each such implementation defines a concrete propagator class and a concrete block class, which ar named Propagator and Block by convention.The Block class in each implementation is derived from BlockTmpl<Propagator>, using the following syntax:

```
class Block : public BlockTmpl<Propagator>
{
  ....
}
```

The algorithms for taking one step of integration of the modified diffusion equation and for computing the monomer concentration field arising from monomers in one block must be implemented in the Block class. These algorithms must be implemented in member functions of the concrete Block class with the following interfaces:

```
// ----------------------------------------------------------------
// Take one step of integration of the modified diffusion equation.
// ----------------------------------------------------------------
void step(Propagator::QField const & in, Propagator::QField& out);
// ----------------------------------------------------------------
// Compute monomer concentration field for this block.
//
// \param prefactor  numerical prefactor of phi/(Q*length)
// ----------------------------------------------------------------
void computeConcentration(double prefactor);
```

These core algorithms are implemented in the Block class, rather than the Propagator class, because the data required to implement these algorithms generally depends on the monomer type and contour length step size ds of a particular block, and can thus be shared by the two propagators associated with a particular block. The data required to implement these algorithms cannot, however, be shared among propagators in different blocks of the same monomer type because the requirement that the length of each block be divided into an integer number of contour length steps implies that different blocks of arbitrary user-specified length must generally be assumed to have slightly different values for the step size ds.
The step() function is called in the implementation of the PropagatorTmpl::solve() member function, within a loop over steps. The computeConcentration() function is called in the implementation of the PolymerTmpl::solve() member function, within a loop over all blocks of the molecule that is called after solution of the modified diffusion equation for all propagators.
Definition at line 89 of file BlockTmpl.h.

12.9.2 Constructor & Destructor Documentation

12.9.2.1 BlockTmpl() `template<class TP >`

`Pscf::BlockTmpl< TP >::BlockTmpl`

Constructor.
Definition at line 205 of file BlockTmpl.h.

12.9.2.2 ∼BlockTmpl() `template<class TP >`

`Pscf::BlockTmpl< TP >::~BlockTmpl ( ) [virtual], [default]`

Destructor.

12.9.3 Member Function Documentation

12.9.3.1 setKuhn() template<class TP >

```
void Pscf::BlockTmpl< TP >::setKuhn (
            double kuhn )  [virtual]
```

Set monomer statistical segment length.

Parameters

kuhn	monomer statistical segment length

Definition at line 226 of file BlockTmpl.h.

12.9.3.2 propagator() [1/2] template<class TP >

```
TP & Pscf::BlockTmpl< TP >::propagator (
            int directionId )  [inline]
```

Get a Propagator for a specified direction.

For a block with v0 = vertexId(0) and v1 = vertexId(1), propagator(0) propagates from vertex v0 to v1, while propagator(1) propagates from vertex v1 to v0.

Parameters

direction↔ Id	integer index for direction (0 or 1)

Definition at line 165 of file BlockTmpl.h.

12.9.3.3 propagator() [2/2] template<class TP >

```
const TP & Pscf::BlockTmpl< TP >::propagator (
            int directionId ) const  [inline]
```

Get a const Propagator for a specified direction.

See above for number conventions.

Parameters

direction↔ Id	integer index for direction (0 or 1)

Definition at line 174 of file BlockTmpl.h.

12.9.3.4 cField() template<class TP >

```
TP::CField & Pscf::BlockTmpl< TP >::cField  [inline]
```

Get the associated monomer concentration field.

Definition at line 185 of file BlockTmpl.h.

12.9.3.5 kuhn() template<class TP >

```
double Pscf::BlockTmpl< TP >::kuhn  [inline]
```

Get monomer statistical segment length.
Definition at line 194 of file BlockTmpl.h.
The documentation for this class was generated from the following file:

- BlockTmpl.h

12.10 Pscf::BondDescriptor Class Reference

A linear bond (including block-bond and joint-bond) within a block copolymer.

```
#include <BondDescriptor.h>
```

Inheritance diagram for Pscf::BondDescriptor:

classPscf_1_1BondDescriptor-eps-converted-to.pdf

Public Member Functions

- BondDescriptor ()

  Constructor.
- template<class Archive >
  void serialize (Archive &ar, unsigned int versionId)

  Serialize to/from archive.

Setters

- void setId (int id)

  Set the id for this block.
- void setVertexIds (int vertexAId, int vertexBId)

  Set indices of associated vertices.
- virtual void setLength (int N)

  Set the length of this bond.

Accessors (getters)

- int id () const

  Get the id of this bond.
- int monomerId (int i) const

  Get the monomer type id at vertex i.
- const Pair< int > & vertexIds () const

  Get the pair of associated vertex ids.
- int vertexId (int i) const

  Get id of an associated vertex.
- int length () const

  Get the number of segments in this bond.
- bool bondtype () const

  Get the type of this bond.
- std::istream & operator>> (std::istream &in, BondDescriptor &bond)

  istream extractor for a BlockDescriptor.
- std::ostream & **operator**<< (std::ostream &out, const BondDescriptor &bond)

12.10.1 Detailed Description

A linear bond (including block-bond and joint-bond) within a block copolymer.
(discrete chain)
Definition at line 19 of file BondDescriptor.h.

12.10.2 Constructor & Destructor Documentation

12.10.2.1 BondDescriptor() `Pscf::BondDescriptor::BondDescriptor ( )`

Constructor.
Definition at line 11 of file BondDescriptor.cpp.

12.10.3 Member Function Documentation

12.10.3.1 serialize() `template<class Archive >`

```
void Pscf::BondDescriptor::serialize (
            Archive & ar,
            unsigned int versionId )
```

Serialize to/from archive.

Parameters

ar	input or output Archive
version←↩ Id	archive format version index

Definition at line 196 of file BondDescriptor.h.

12.10.3.2 setId() `void Pscf::BondDescriptor::setId (`

```
            int id )
```

Set the id for this block.

Parameters

id	integer index for this bond

Definition at line 20 of file BondDescriptor.cpp.
References id().

12.10.3.3 setVertexIds() `void Pscf::BondDescriptor::setVertexIds (`

```
            int vertexAId,
            int vertexBId )
```

Set indices of associated vertices.

Parameters

vertex←↩ AId	integer id of vertex A

Parameters

vertex↩ BId	integer id of vertex B

Definition at line 26 of file BondDescriptor.cpp.

12.10.3.4 setLength() `void Pscf::BondDescriptor::setLength ( int N ) [virtual]`

Set the length of this bond.
The "length" is the number of segments on this bond.

Parameters

length	block length (number of monomers).

Definition at line 35 of file BondDescriptor.cpp.

12.10.3.5 id() `int Pscf::BondDescriptor::id ( ) const [inline]`

Get the id of this bond.
Definition at line 152 of file BondDescriptor.h.
Referenced by Pscf::Vertex::addBond(), and setId().

12.10.3.6 monomerId() `int Pscf::BondDescriptor::monomerId ( int i ) const [inline]`

Get the monomer type id at vertex i.
Definition at line 159 of file BondDescriptor.h.

12.10.3.7 vertexIds() `const Pair< int > & Pscf::BondDescriptor::vertexIds ( ) const [inline]`

Get the pair of associated vertex ids.
Definition at line 167 of file BondDescriptor.h.

12.10.3.8 vertexId() `int Pscf::BondDescriptor::vertexId ( int i ) const [inline]`

Get id of an associated vertex.

Parameters

i	index of vertex (0 or 1)

Definition at line 174 of file BondDescriptor.h.
Referenced by Pscf::Vertex::addBond().

12.10.3.9 length() `int Pscf::BondDescriptor::length ( ) const [inline]`

Get the number of segments in this bond.
For a block bond, the return value should be a positive integer number.

Definition at line 181 of file BondDescriptor.h.

12.10.3.10 bondtype() `bool Pscf::BondDescriptor::bondtype ( ) const [inline]`

Get the type of this bond.
0 for block-bond; 1 for joint-bond
Definition at line 189 of file BondDescriptor.h.

12.10.4 Friends And Related Function Documentation

12.10.4.1 operator>> `std::istream& operator>> ( std::istream & in, BondDescriptor & bond ) [friend]`

istream extractor for a BlockDescriptor.

Parameters

in	input stream
bond	BondDescriptor to be read from stream

Returns

modified input stream

Definition at line 41 of file BondDescriptor.cpp.
The documentation for this class was generated from the following files:

- BondDescriptor.h
- BondDescriptor.cpp

12.11 Pscf::ChiInteraction Class Reference

Flory-Huggins excess free energy model.
`#include <ChiInteraction.h>`
Inheritance diagram for Pscf::ChiInteraction:

classPscf_1_1ChiInteraction-eps-converted-to.pdf

Public Member Functions

- ChiInteraction ()

Constructor.

- virtual ~ChiInteraction ()

  Destructor.
- virtual void readParameters (std::istream &in)

  Read chi parameters.
- virtual double fHelmholtz (Array< double > const &c) const

  Compute excess Helmholtz free energy per monomer.
- virtual void computeW (Array< double > const &c, Array< double > &w) const

  Compute chemical potential from concentration.
- virtual void computeC (Array< double > const &w, Array< double > &c, double &xi) const

  Compute concentration from chemical potential fields.
- virtual void computeXi (Array< double > const &w, double &xi) const

  Compute Langrange multiplier xi from chemical potential fields.
- virtual void computeDwDc (Array< double > const &c, Matrix< double > &dWdC) const

  Compute second derivatives of free energy.
- double chi (int i, int j)

  Return one element of the chi matrix.
- double chiInverse (int i, int j)

  Return one element of the inverse chi matrix.
- double idemp (int i, int j)

  Return one element of the idempotent matrix.

Additional Inherited Members

12.11.1 Detailed Description

Flory-Huggins excess free energy model.
Definition at line 23 of file ChiInteraction.h.

12.11.2 Constructor & Destructor Documentation

12.11.2.1 ChiInteraction() `Pscf::ChiInteraction::ChiInteraction ( )`

Constructor.
Definition at line 18 of file ChiInteraction.cpp.
References Util::ParamComposite::setClassName().

12.11.2.2 ~ChiInteraction() `Pscf::ChiInteraction::~ChiInteraction ( ) [virtual]`

Destructor.
Definition at line 25 of file ChiInteraction.cpp.

12.11.3 Member Function Documentation

12.11.3.1 readParameters() `void Pscf::ChiInteraction::readParameters ( std::istream & in ) [virtual]`

Read chi parameters.

Must be called after setNMonomer.

Reimplemented from Util::ParamComposite.

Definition at line 31 of file ChiInteraction.cpp.

References Pscf::LuSolver::allocate(), Util::DMatrix< Data >::allocate(), Pscf::LuSolver::computeLU(), Pscf::LuSolver::inverse(), Pscf::Interaction::nMonomer(), Util::ParamComposite::readDSymmMatrix(), UTIL_CHECK, and UTIL_THROW.

12.11.3.2 fHelmholtz() `double Pscf::ChiInteraction::fHelmholtz ( Array< double > const & c ) const [virtual]`

Compute excess Helmholtz free energy per monomer.

Parameters

c	array of concentrations, for each type (input)

Implements Pscf::Interaction.

Definition at line 86 of file ChiInteraction.cpp.

References Pscf::Interaction::nMonomer().

12.11.3.3 computeW() `void Pscf::ChiInteraction::computeW ( Array< double > const & c, Array< double > & w ) const [virtual]`

Compute chemical potential from concentration.

Parameters

c	array of concentrations, for each type (input)
w	array of chemical potentials for types (ouptut)

Implements Pscf::Interaction.

Definition at line 102 of file ChiInteraction.cpp.

References Pscf::Interaction::nMonomer().

12.11.3.4 computeC() `void Pscf::ChiInteraction::computeC ( Array< double > const & w, Array< double > & c, double & xi ) const [virtual]`

Compute concentration from chemical potential fields.

Parameters

w	array of chemical potentials for types (inut)
c	array of vol. fractions, for each type (output)
xi	Langrange multiplier pressure (output)

Implements Pscf::Interaction.

Definition at line 118 of file ChiInteraction.cpp.
References Pscf::Interaction::nMonomer().

12.11.3.5 computeXi()

```
void Pscf::ChiInteraction::computeXi (
        Array< double > const & w,
        double & xi ) const [virtual]
```

Compute Langrange multiplier xi from chemical potential fields.

Parameters

w	array of chemical potentials for types (inut)
xi	Langrange multiplier pressure (output)

Implements Pscf::Interaction.
Definition at line 144 of file ChiInteraction.cpp.
References Pscf::Interaction::nMonomer().

12.11.3.6 computeDwDc()

```
void Pscf::ChiInteraction::computeDwDc (
        Array< double > const & c,
        Matrix< double > & dWdC ) const [virtual]
```

Compute second derivatives of free energy.
Upon return, the elements of the square matrix dWdC, are given by derivatives dWdC(i,j) = dW(i)/dC(j), which are also second derivatives of the interaction free energy. For this Flory-Huggins chi parameter model, this is simply given by the chi matrix dWdC(i,j) = chi(i, j).

Parameters

c	array of concentrations, for each type (input)
dWdC	matrix of derivatives (output)

Implements Pscf::Interaction.
Definition at line 163 of file ChiInteraction.cpp.
References Pscf::Interaction::nMonomer().

12.11.3.7 chi()

```
double Pscf::ChiInteraction::chi (
        int i,
        int j ) [inline]
```

Return one element of the chi matrix.

Parameters

i	row index
j	column index

Definition at line 142 of file ChiInteraction.h.

12.11.3.8 chiInverse()

```
double Pscf::ChiInteraction::chiInverse (
        int i,
```

int *j*) [inline]

Return one element of the inverse chi matrix.

Parameters

i	row index
j	column index

Definition at line 145 of file ChiInteraction.h.

12.11.3.9 idemp() double Pscf::ChiInteraction::idemp (

int *i*,

int *j*) [inline]

Return one element of the idempotent matrix.

Parameters

i	row index
j	column index

Definition at line 148 of file ChiInteraction.h.

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

- ChiInteraction.h
- ChiInteraction.cpp

12.12 Pscf::DPolymerTmpl< Bond > Class Template Reference

Inheritance diagram for Pscf::DPolymerTmpl< Bond >:

classPscf_1_1DPolymerTmpl-eps-converted-to.pdf

Public Member Functions

- virtual void readParameters (std::istream &in)

    Read the body of parameter block, without begin and end lines.
- int nBond () const

    Accessors.

Additional Inherited Members

12.12.1 Detailed Description

template<class Bond>
class Pscf::DPolymerTmpl< Bond >

Definition at line 23 of file DPolymerTmpl.h.

12.12.2 Member Function Documentation

12.12.2.1 readParameters() `template<class Bond >`

`void Pscf::DPolymerTmpl< Bond >::readParameters ( std::istream & in ) [virtual]`

Read the body of parameter block, without begin and end lines.

Most subclasses of ParamComposite should re-implement this function, which has an empty default implementation. Every subclass of Paramcomposite must either: (1) Re-implement this function and rely on the default implementation of readParam(), which calls this function. (2) Re-implement readParam() itself. Option (1) is far more common. Option (2) is required only for classes that require a non-standard treatment of the beginning and ending lines (e.g., the Manager class template).

Parameters

in	input stream for reading

Reimplemented from Util::ParamComposite.
Definition at line 192 of file DPolymerTmpl.h.

12.12.2.2 nBond() `template<class Bond >`

`int Pscf::DPolymerTmpl< Bond >::nBond [inline]`

Accessors.
Definition at line 91 of file DPolymerTmpl.h.
The documentation for this class was generated from the following file:

- DPolymerTmpl.h

12.13 Pscf::Field< T > Class Template Reference

Base class template for a field defined on a spatial grid.
`#include <Field.h>`
Inheritance diagram for Pscf::Field< T >:

classPscf_1_1Field-eps-converted-to.pdf

Public Member Functions

- Field ()

  Constructor.

- Field (Field< T > const &other)

  Copy constructor.
- Field< T > & operator= (Field< T > const &other)

  Assignment operator.
- Field< T > & operator= (T &scalar)

  Assignment - assign all elements to a common scalar.
- Field< T > & operator+= (Field< T > &other)

  Increment operator - add one field by another.
- Field< T > & operator-= (Field< T > &other)

  Decrement operator - subtract one field from another.
- Field< T > & operator*= (T scalar)

  Multiplication operator - multiply one field by a scalar.
- Field< T > & operator*= (Field< T > &other)

  Pointwise multipication of one field by another.
- void setToZero ()

  Set all elements to zero.
- T average () const

  Compute and return average of all elements.

Additional Inherited Members

12.13.1 Detailed Description

template<typename T = double>
class Pscf::Field< T >

Base class template for a field defined on a spatial grid.
Derived from DArray<T>, and provides useful arithmetic operations.
Definition at line 23 of file Field.h.

12.13.2 Constructor & Destructor Documentation

12.13.2.1 Field() [1/2] `template<typename T = double>`

`Pscf::Field< T >::Field ( )`

Constructor.

12.13.2.2 Field() [2/2] `template<class T >`

```
Pscf::Field< T >::Field (
            Field< T > const & other )
```

Copy constructor.
Definition at line 98 of file Field.h.
References Util::Memory::allocate(), Util::Array< double >::capacity_, Util::Array< double >::data_, Util::DArray< double >::isAllocated(), and UTIL_THROW.

12.13.3 Member Function Documentation

12.13.3.1 operator=() [1/2] template<class T >

```
Field< T > & Pscf::Field< T >::operator= (
            Field< T > const & other )
```

Assignment operator.

Definition at line 122 of file Field.h.

References Util::Array< double >::capacity(), Util::Array< double >::capacity_, Util::DArray< double >::isAllocated(), and UTIL_THROW.

12.13.3.2 operator=() [2/2] template<typename T >

```
Field< T > & Pscf::Field< T >::operator= (
            T & scalar )
```

Assignment - assign all elements to a common scalar.

Definition at line 150 of file Field.h.

References UTIL_THROW.

12.13.3.3 operator+=() template<typename T >

```
Field< T > & Pscf::Field< T >::operator+= (
            Field< T > & other )
```

Increment operator - add one field by another.

Definition at line 165 of file Field.h.

References Util::Array< double >::capacity_, Util::Array< double >::data_, Util::DArray< double >::isAllocated(), and UTIL_THROW.

12.13.3.4 operator-=() template<typename T >

```
Field< T > & Pscf::Field< T >::operator-= (
            Field< T > & other )
```

Decrement operator - subtract one field from another.

Definition at line 186 of file Field.h.

References Util::Array< double >::capacity_, Util::Array< double >::data_, Util::DArray< double >::isAllocated(), and UTIL_THROW.

12.13.3.5 operator*=() [1/2] template<typename T >

```
Field< T > & Pscf::Field< T >::operator*= (
            T scalar )
```

Multiplication operator - multiply one field by a scalar.

Definition at line 210 of file Field.h.

References UTIL_THROW.

12.13.3.6 operator*=() [2/2] template<typename T >

```
Field< T > & Pscf::Field< T >::operator*= (
            Field< T > & other )
```

Pointwise multipication of one field by another.

Definition at line 227 of file Field.h.

References Util::Array< double >::capacity_, Util::Array< double >::data_, Util::DArray< double >::isAllocated(), and UTIL_THROW.

12.13.3.7 setToZero() `template<typename T >`

`void Pscf::Field< T >::setToZero`

Set all elements to zero.

Definition at line 250 of file Field.h.

12.13.3.8 average() `template<typename T >`

`T Pscf::Field< T >::average`

Compute and return average of all elements.

Definition at line 261 of file Field.h.

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

- Field.h

12.14 Pscf::Homogeneous::Clump Class Reference

Collection of all monomers of a single type in a molecule.

`#include <Clump.h>`

Public Member Functions

- Clump ()

  Constructor.
- template<class Archive >
  void serialize (Archive &ar, unsigned int versionId)

  Serialize to/from archive.

Setters

- void setMonomerId (int monomerId)

  Set the monomer id.
- void setSize (double size)

  Set the size of this block.

Accessors (getters)

- int monomerId () const

  Get the monomer type id.
- double size () const

  Get the size (number of monomers) in this block.
- std::istream & operator>> (std::istream &in, Clump &block)

  istream extractor for a Clump.
- std::ostream & operator<< (std::ostream &out, const Clump &block)

  ostream inserter for a Clump.

12.14.1 Detailed Description

Collection of all monomers of a single type in a molecule.

A clump has a monomer id and a size. The size of a clump is the volume occupied by all monomers of the specified type in a particular molecular species, divided by a monomer reference volume.

For a block copolymer, a clump is generally different than a block because a clump may include the monomers in two or more blocks of the same monomer type. Hompolymer and point solvent molecular species each have only one clump.

Definition at line 35 of file Clump.h.

12.14.2 Constructor & Destructor Documentation

12.14.2.1 Clump() `Pscf::Homogeneous::Clump::Clump ( )`

Constructor.
Definition at line 16 of file Clump.cpp.

12.14.3 Member Function Documentation

12.14.3.1 serialize() `template<class Archive >`

```
void Pscf::Homogeneous::Clump::serialize (
            Archive & ar,
            unsigned int versionId )
```

Serialize to/from archive.

Parameters

ar	input or output Archive
versionId	archive format version index

Definition at line 140 of file Clump.h.

12.14.3.2 setMonomerId() `void Pscf::Homogeneous::Clump::setMonomerId (`

```
            int monomerId )
```

Set the monomer id.

Parameters

monomerId	integer id of monomer type (>=0)

Definition at line 24 of file Clump.cpp.
References monomerId().

12.14.3.3 setSize() `void Pscf::Homogeneous::Clump::setSize (`

```
            double size )
```

Set the size of this block.
The ``size'' is steric volume / reference volume.

Parameters

size	block size (number of monomers).

Definition at line 30 of file Clump.cpp.
References size().

12.14.3.4 monomerId() `int Pscf::Homogeneous::Clump::monomerId ( ) const [inline]`

Get the monomer type id.

Definition at line 127 of file Clump.h.

Referenced by Pscf::Homogeneous::Mixture::computeMu(), Pscf::Homogeneous::Mixture::computePhi(), setMonomerId(), and Pscf::Homogeneous::Mixture::validate().

12.14.3.5 size() `double Pscf::Homogeneous::Clump::size ( ) const [inline]`

Get the size (number of monomers) in this block.

Definition at line 133 of file Clump.h.

Referenced by Pscf::Homogeneous::Mixture::computeMu(), Pscf::Homogeneous::Mixture::computePhi(), and setSize().

12.14.4 Friends And Related Function Documentation

12.14.4.1 operator>>

```
std::istream& operator>> (
        std::istream & in,
        Clump & block )  [friend]
```

istream extractor for a Clump.

Parameters

in	input stream
block	Clump to be read from stream

Returns

modified input stream

Definition at line 36 of file Clump.cpp.

12.14.4.2 operator<<

```
std::ostream& operator<< (
        std::ostream & out,
        const Clump & block )  [friend]
```

ostream inserter for a Clump.

Parameters

out	output stream
block	Clump to be written to stream

Returns

modified output stream

Definition at line 46 of file Clump.cpp.

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

- Clump.h
- Clump.cpp

12.15 Pscf::Homogeneous::Mixture Class Reference

A spatially homogeneous mixture.

`#include <Mixture.h>`

Inheritance diagram for Pscf::Homogeneous::Mixture:

classPscf_1_1Homogeneous_1_1Mixture-eps-converted-to.pdf

Public Member Functions

- Mixture ()

  Constructor.
- ∼Mixture ()

  Destructor.

Initialization.

- virtual void readParameters (std::istream &in)

  Read parameters from file and initialize.
- void setNMolecule (int nMonomer)

  Set the number of molecular species and allocate memory.
- void setNMonomer (int nMonomer)

  Set the number of monomer types.

Thermodynamics Computations

- void setComposition (DArray< double > const &phi)

  Set system composition.
- void computeMu (Interaction const &interaction, double xi=0.0)

  Compute chemical potential from preset composition.
- void computePhi (Interaction const &interaction, DArray< double > const &mu, DArray< double > const &phi, double &xi)

  Compute composition from chemical potentials.
- void computeFreeEnergy (Interaction const &interaction)

  Compute Helmholtz free energy and pressure.

Accessors

- Molecule & molecule (int id)

  Get a molecule object.
- double mu (int id) const

  Return chemical potential for one species.
- double phi (int id) const

  Return molecular volume fraction for one species.
- double c (int id) const

Return monomer volume fraction for one monomer type.

- double fHelmholtz () const

  Return Helmholtz free energy per monomer / kT.
- double pressure () const

  Return pressure in units of kT / monomer volume.
- int nMolecule () const

  Get number of molecule species.
- int nMonomer () const

  Get number of monomer types.
- void validate () const

  Validate all data structures.

Additional Inherited Members

12.15.1 Detailed Description

A spatially homogeneous mixture.
Definition at line 33 of file pscf/homogeneous/Mixture.h.

12.15.2 Constructor & Destructor Documentation

12.15.2.1 Mixture() `Pscf::Homogeneous::Mixture::Mixture ( )`

Constructor.
Definition at line 21 of file Mixture.cpp.
References Util::ParamComposite::setClassName().

12.15.2.2 ∼Mixture() `Pscf::Homogeneous::Mixture::∼Mixture ( )`

Destructor.
Definition at line 45 of file Mixture.cpp.

12.15.3 Member Function Documentation

12.15.3.1 readParameters() `void Pscf::Homogeneous::Mixture::readParameters ( std::istream & in ) [virtual]`

Read parameters from file and initialize.

Parameters

in	input parameter file

Reimplemented from Util::ParamComposite.
Definition at line 51 of file Mixture.cpp.
References Util::DArray< Data >::allocate(), Util::ParamComposite::readParamComposite(), UTIL_ASSERT, and validate().

12.15.3.2 setNMolecule() `void Pscf::Homogeneous::Mixture::setNMolecule (`

`int nMonomer )`

Set the number of molecular species and allocate memory.
Definition at line 71 of file Mixture.cpp.
References Util::DArray< Data >::allocate(), nMolecule(), and UTIL_ASSERT.

12.15.3.3 setNMonomer() `void Pscf::Homogeneous::Mixture::setNMonomer ( int nMonomer )`

Set the number of monomer types.
Definition at line 80 of file Mixture.cpp.
References Util::DArray< Data >::allocate(), nMonomer(), and UTIL_ASSERT.

12.15.3.4 setComposition() `void Pscf::Homogeneous::Mixture::setComposition ( DArray< double > const & phi )`

Set system composition.

Parameters

phi	array of molecular volume fractions.

Definition at line 91 of file Mixture.cpp.
References phi(), UTIL_ASSERT, UTIL_CHECK, and validate().
Referenced by computePhi().

12.15.3.5 computeMu() `void Pscf::Homogeneous::Mixture::computeMu ( Interaction const & interaction, double xi = 0.0 )`

Compute chemical potential from preset composition.
Precondition: setComposition must be called prior. Postcondition: Upon return, mu array is set.

Parameters

interaction	excess free energy model (input)
xi	Lagrange multiplier field (input)

Definition at line 149 of file Mixture.cpp.
References c(), Pscf::Homogeneous::Molecule::clump(), Pscf::Interaction::computeW(), Pscf::Homogeneous::Clump::monomerId(), mu(), Pscf::Homogeneous::Molecule::nClump(), Pscf::Interaction::nMonomer(), Pscf::Homogeneous::Clump::size(), Pscf::Homogeneous::Molecule::size(), and UTIL_CHECK.
Referenced by computePhi().

12.15.3.6 computePhi() `void Pscf::Homogeneous::Mixture::computePhi ( Interaction const & interaction, DArray< double > const & mu, DArray< double > const & phi, double & xi )`

Compute composition from chemical potentials.

Parameters

interaction	excess free energy model (input)
mu	target molecular chemical potentials (input)
phi	guess of molecular volume fractions (input)
xi	Lagrange multiplier field (input/output)

Definition at line 180 of file Mixture.cpp.
References Pscf::LuSolver::allocate(), Util::DMatrix< Data >::allocate(), Util::DArray< Data >::allocate(), Util::Array< Data >::capacity(), Pscf::Homogeneous::Molecule::clump(), Pscf::Interaction::computeDwDc(), Pscf::LuSolver↩ ::computeLU(), computeMu(), molecule(), Pscf::Homogeneous::Clump::monomerId(), mu(), Pscf::Homogeneous::↩ Molecule::nClump(), Pscf::Interaction::nMonomer(), phi(), setComposition(), Pscf::Homogeneous::Clump::size(), Pscf↩ ::Homogeneous::Molecule::size(), Pscf::LuSolver::solve(), UTIL_ASSERT, and UTIL_THROW.

12.15.3.7 computeFreeEnergy() `void Pscf::Homogeneous::Mixture::computeFreeEnergy ( Interaction const & interaction )`

Compute Helmholtz free energy and pressure.
Preconditions and postconditions:

Precondition

setComposition must be called prior.

computeMu must be called prior.

Postcondition

fHelmholtz and pressure are set.

Parameters

interaction	excess free energy model (input)

Definition at line 374 of file Mixture.cpp.
References Pscf::Interaction::fHelmholtz().

12.15.3.8 molecule() `Molecule & Pscf::Homogeneous::Mixture::molecule ( int id ) [inline]`

Get a molecule object.

Parameters

id	integer molecule species index (0 <= id < nMolecule)

Definition at line 285 of file pscf/homogeneous/Mixture.h.
References UTIL_ASSERT.
Referenced by computePhi().

12.15.3.9 mu() `double Pscf::Homogeneous::Mixture::mu ( int id ) const [inline]`

Return chemical potential for one species.

Parameters

id	integer molecule species index (0 <= id < nMolecule)

Definition at line 292 of file pscf/homogeneous/Mixture.h.
References UTIL_ASSERT.
Referenced by computeMu(), and computePhi().

12.15.3.10 phi() `double Pscf::Homogeneous::Mixture::phi ( int id ) const [inline]`

Return molecular volume fraction for one species.

Parameters

id	integer molecule species index (0 <= id < nMolecule)

Definition at line 299 of file pscf/homogeneous/Mixture.h.
References UTIL_ASSERT.
Referenced by computePhi(), and setComposition().

12.15.3.11 c() `double Pscf::Homogeneous::Mixture::c ( int id ) const [inline]`

Return monomer volume fraction for one monomer type.

Parameters

id	monomer type index (0 <= id < nMonomer)

Definition at line 306 of file pscf/homogeneous/Mixture.h.
References UTIL_ASSERT.
Referenced by computeMu().

12.15.3.12 fHelmholtz() `double Pscf::Homogeneous::Mixture::fHelmholtz ( ) const [inline]`

Return Helmholtz free energy per monomer / kT.
Definition at line 313 of file pscf/homogeneous/Mixture.h.

12.15.3.13 pressure() `double Pscf::Homogeneous::Mixture::pressure ( ) const [inline]`

Return pressure in units of kT / monomer volume.
Definition at line 316 of file pscf/homogeneous/Mixture.h.

12.15.3.14 nMolecule() `int Pscf::Homogeneous::Mixture::nMolecule ( ) const [inline]`

Get number of molecule species.
Definition at line 319 of file pscf/homogeneous/Mixture.h.
Referenced by setNMolecule().

12.15.3.15 nMonomer() `int Pscf::Homogeneous::Mixture::nMonomer ( ) const [inline]`

Get number of monomer types.

Definition at line 322 of file pscf/homogeneous/Mixture.h.

Referenced by setNMonomer().

12.15.3.16 validate() `void Pscf::Homogeneous::Mixture::validate ( ) const`

Validate all data structures.

Throw an exception if an error is found.

Definition at line 400 of file Mixture.cpp.

References Pscf::Homogeneous::Molecule::clump(), Pscf::Homogeneous::Clump::monomerId(), Pscf::Homogeneous::Molecule::nClump(), and UTIL_ASSERT.

Referenced by readParameters(), and setComposition().

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

- pscf/homogeneous/Mixture.h
- Mixture.cpp

12.16 Pscf::Homogeneous::Molecule Class Reference

Descriptor of a molecular species in a homogeneous mixture.

`#include <Molecule.h>`

Inheritance diagram for Pscf::Homogeneous::Molecule:

classPscf_1_1Homogeneous_1_1Molecule-eps-converted-to.pdf

Public Member Functions

- Molecule ()

  Constructor.
- ∼Molecule ()

  Destructor.
- virtual void readParameters (std::istream &in)

  Read and initialize.
- void setNClump (int nClump)

  Set the number of clumps, and allocate memory.
- void computeSize ()

  Compute total molecule size by adding clump sizes.

Accessors

- Clump & clump (int id)

  Get a specified Clump.

- const Clump & clump (int id) const

    Get a specified Clump.
- int nClump () const

    Number of monomer clumps (monomer types).
- double size () const

    Total molecule size = volume / reference volume.

Additional Inherited Members

12.16.1 Detailed Description

Descriptor of a molecular species in a homogeneous mixture.
A Homogeneous::Molecule has:

- An array of Homogeneous::Clump objects
- An overall size (volume/monomer volume)

Each Clump has a monomer type id and a size. The size is the total volume of monomers of that type in a molecule of this species.
Definition at line 38 of file Molecule.h.

12.16.2 Constructor & Destructor Documentation

12.16.2.1 Molecule() `Pscf::Homogeneous::Molecule::Molecule ( )`

Constructor.
Definition at line 15 of file Molecule.cpp.
References Util::ParamComposite::setClassName().

12.16.2.2 ∼Molecule() `Pscf::Homogeneous::Molecule::∼Molecule ( )`

Destructor.
Definition at line 25 of file Molecule.cpp.

12.16.3 Member Function Documentation

12.16.3.1 readParameters() `void Pscf::Homogeneous::Molecule::readParameters ( std::istream & in ) [virtual]`

Read and initialize.
Call either this or setNClump to initialize, not both.

Parameters

in	input parameter stream

Reimplemented from Util::ParamComposite.
Definition at line 31 of file Molecule.cpp.
References UTIL_ASSERT.

12.16.3.2 setNClump() `void Pscf::Homogeneous::Molecule::setNClump ( int nClump )`

Set the number of clumps, and allocate memory.
Call either this or readParameters to initialize, not both. If this is used to allocate memory, all clump properties must be set using Clump::setMonomerId() and Clump::setSize().
Definition at line 47 of file Molecule.cpp.
References UTIL_ASSERT.

12.16.3.3 computeSize() `void Pscf::Homogeneous::Molecule::computeSize ( )`

Compute total molecule size by adding clump sizes.
Definition at line 58 of file Molecule.cpp.
References UTIL_ASSERT.

12.16.3.4 clump() [1/2] `Homogeneous::Clump & Pscf::Homogeneous::Molecule::clump ( int id ) [inline]`

Get a specified Clump.

Parameters

id	clump index, 0 <= id < nClump

Definition at line 141 of file Molecule.h.
Referenced by Pscf::Homogeneous::Mixture::computeMu(), Pscf::Homogeneous::Mixture::computePhi(), and Pscf::Homogeneous::Mixture::validate().

12.16.3.5 clump() [2/2] `const Homogeneous::Clump & Pscf::Homogeneous::Molecule::clump ( int id ) const [inline]`

Get a specified Clump.

Parameters

id	clump index, 0 <= id < nClump

Definition at line 147 of file Molecule.h.

12.16.3.6 nClump() `int Pscf::Homogeneous::Molecule::nClump ( ) const [inline]`

Number of monomer clumps (monomer types).
Definition at line 126 of file Molecule.h.
Referenced by Pscf::Homogeneous::Mixture::computeMu(), Pscf::Homogeneous::Mixture::computePhi(), and Pscf::Homogeneous::Mixture::validate().

12.16.3.7 size() `double Pscf::Homogeneous::Molecule::size ( ) const [inline]`

Total molecule size = volume / reference volume.
Definition at line 132 of file Molecule.h.
References UTIL_CHECK.
Referenced by Pscf::Homogeneous::Mixture::computeMu(), and Pscf::Homogeneous::Mixture::computePhi().
The documentation for this class was generated from the following files:

- Molecule.h
- Molecule.cpp

12.17 Pscf::Interaction Class Reference

Base class for excess free energy models.

```
#include <Interaction.h>
```

Inheritance diagram for Pscf::Interaction:

classPscf_1_1Interaction-eps-converted-to.pdf

Public Member Functions

- Interaction ()

  Constructor.
- virtual ∼Interaction ()

  Destructor.
- void setNMonomer (int nMonomer)

  Set the number of monomer types.
- virtual double fHelmholtz (Array< double > const &c) const =0

  Compute excess Helmholtz free energy per monomer.
- virtual void computeW (Array< double > const &c, Array< double > &w) const =0

  Compute interaction contributions to chemical potentials.
- virtual void computeC (Array< double > const &w, Array< double > &c, double &xi) const =0

  Compute concentration and xi from chemical potentials.
- virtual void computeXi (Array< double > const &w, double &xi) const =0

  Compute Langrange multiplier xi from chemical potentials.
- virtual void computeDwDc (Array< double > const &c, Matrix< double > &dWdC) const =0

  Compute matrix of derivatives of w fields w/ respect to c fields.
- int nMonomer () const

  Get number of monomer types.

Additional Inherited Members

12.17.1 Detailed Description

Base class for excess free energy models.

Definition at line 25 of file Interaction.h.

12.17.2 Constructor & Destructor Documentation

12.17.2.1 Interaction() `Pscf::Interaction::Interaction ( )`

Constructor.

Definition at line 14 of file Interaction.cpp.

References Util::ParamComposite::setClassName().

12.17.2.2 ∼Interaction() `Pscf::Interaction::∼Interaction ( ) [virtual]`

Destructor.

Definition at line 18 of file Interaction.cpp.

12.17.3 Member Function Documentation

12.17.3.1 setNMonomer() `void Pscf::Interaction::setNMonomer ( int nMonomer )`

Set the number of monomer types.

Parameters

nMonomer	number of monomer types.

Definition at line 21 of file Interaction.cpp.

References nMonomer().

12.17.3.2 fHelmholtz() `virtual double Pscf::Interaction::fHelmholtz ( Array< double > const & c ) const [pure virtual]`

Compute excess Helmholtz free energy per monomer.

Parameters

c	array of concentrations, for each type (input)

Implemented in Pscf::ChiInteraction.

Referenced by Pscf::Homogeneous::Mixture::computeFreeEnergy().

12.17.3.3 computeW() `virtual void Pscf::Interaction::computeW ( Array< double > const & c, Array< double > & w ) const [pure virtual]`

Compute interaction contributions to chemical potentials.

The resulting chemical potential fields are those obtained with a vanishing Lagrange multiplier / pressure field, xi = 0.

Parameters

c	array of concentrations, for each type (input)
w	array of chemical potentials, for each type (output)

Implemented in Pscf::ChiInteraction.
Referenced by Pscf::Homogeneous::Mixture::computeMu().

12.17.3.4 computeC()

```
virtual void Pscf::Interaction::computeC (
    Array< double > const & w,
    Array< double > & c,
    double & xi ) const [pure virtual]
```

Compute concentration and xi from chemical potentials.

Parameters

w	array of chemical potentials, for each type (input)
c	array of concentrations, for each type (output)
xi	Lagrange multiplier pressure (output)

Implemented in Pscf::ChiInteraction.

12.17.3.5 computeXi()

```
virtual void Pscf::Interaction::computeXi (
    Array< double > const & w,
    double & xi ) const [pure virtual]
```

Compute Langrange multiplier xi from chemical potentials.

Parameters

w	array of chemical potentials, for each type (input)
xi	Lagrange multiplier pressure (output)

Implemented in Pscf::ChiInteraction.

12.17.3.6 computeDwDc()

```
virtual void Pscf::Interaction::computeDwDc (
    Array< double > const & c,
    Matrix< double > & dWdC ) const [pure virtual]
```

Compute matrix of derivatives of w fields w/ respect to c fields.
Upon return, the elements of the matrix dWdC are given by derivatives elements dWdC(i,j) = dW(i)/dC(j), which are also second derivatives of fHelmholtz with respect to concentrations.

Parameters

c	array of concentrations, for each type (input)
dWdC	square symmetric matrix of derivatives (output)

Implemented in Pscf::ChiInteraction.
Referenced by Pscf::Homogeneous::Mixture::computePhi().

12.17.3.7 nMonomer()

```
int Pscf::Interaction::nMonomer ( ) const [inline]
```

Get number of monomer types.
Definition at line 118 of file Interaction.h.

Referenced by Pscf::ChiInteraction::computeC(), Pscf::ChiInteraction::computeDwDc(), Pscf::Homogeneous::Mixture::computeMu(), Pscf::Homogeneous::Mixture::computePhi(), Pscf::ChiInteraction::computeW(), Pscf::ChiInteraction::computeXi(), Pscf::ChiInteraction::fHelmholtz(), Pscf::ChiInteraction::readParameters(), and setNMonomer().
The documentation for this class was generated from the following files:

- Interaction.h
- Interaction.cpp

12.18 Pscf::IntVec< D, T > Class Template Reference

An IntVec<D, T> is a D-component vector of elements of integer type T.
`#include <IntVec.h>`
Inheritance diagram for Pscf::IntVec< D, T >:

classPscf_1_1IntVec-eps-converted-to.pdf

Constructors

- static const int Width = 10
  Width of field per Cartesian coordinate in stream IO.
- IntVec ()
  Default constructor.
- IntVec (const IntVec< D, T > &v)
  Copy constructor.
- IntVec (T const ∗v)
  Construct from C array.
- IntVec (T s)
  Constructor, initialize all elements to a scalar value.

Additional Inherited Members

12.18.1 Detailed Description

template<int D, typename T = int>
class Pscf::IntVec< D, T >

An IntVec<D, T> is a D-component vector of elements of integer type T.
Default of type T is T = int.
Definition at line 26 of file IntVec.h.

12.18.2 Constructor & Destructor Documentation

12.18.2.1 IntVec() [1/4] template<int D, typename T = int>

Pscf::IntVec< D, T >::IntVec ( ) [inline]
Default constructor.
Definition at line 37 of file IntVec.h.

12.18.2.2 IntVec() [2/4] template<int D, typename T = int>

```
Pscf::IntVec< D, T >::IntVec (
            const IntVec< D, T > & v ) [inline]
```

Copy constructor.

Parameters

v	IntVec<D, T> to be copied

Definition at line 46 of file IntVec.h.

12.18.2.3 IntVec() [3/4] template<int D, typename T = int>

```
Pscf::IntVec< D, T >::IntVec (
            T const * v ) [inline]
```

Construct from C array.

Parameters

v	C array to be copied

Definition at line 55 of file IntVec.h.

12.18.2.4 IntVec() [4/4] template<int D, typename T = int>

```
Pscf::IntVec< D, T >::IntVec (
            T s ) [inline], [explicit]
```

Constructor, initialize all elements to a scalar value.

Parameters

s	scalar initial value for all elements.

Definition at line 64 of file IntVec.h.

12.18.3 Member Data Documentation

12.18.3.1 Width template<int D, typename T = int>

```
const int Pscf::IntVec< D, T >::Width = 10 [static]
```

Width of field per Cartesian coordinate in stream IO.

Definition at line 69 of file IntVec.h.

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

- IntVec.h

12.19 Pscf::LuSolver Class Reference

Solve Ax=b by LU decomposition of A.

```
#include <LuSolver.h>
```

Public Member Functions

- LuSolver ()

    Constructor.
- ∼LuSolver ()

    Destructor.
- void allocate (int n)

    Allocate memory.
- void computeLU (const Matrix< double > &A)

    Compute the LU decomposition for later use.
- void solve (Array< double > &b, Array< double > &x)

    Solve Ax = b for known b to compute x.
- void inverse (Matrix< double > &inv)

    Compute inverse of matrix A.

12.19.1 Detailed Description

Solve Ax=b by LU decomposition of A.
This class is a simple wrapper for the functions provided by the Gnu Scientific Library (GSL).
Definition at line 30 of file LuSolver.h.

12.19.2 Constructor & Destructor Documentation

12.19.2.1 LuSolver() `Pscf::LuSolver::LuSolver ( )`

Constructor.
Definition at line 14 of file LuSolver.cpp.

12.19.2.2 ∼LuSolver() `Pscf::LuSolver::~LuSolver ( )`

Destructor.
Definition at line 34 of file LuSolver.cpp.

12.19.3 Member Function Documentation

12.19.3.1 allocate() `void Pscf::LuSolver::allocate ( int n )`

Allocate memory.

Parameters

n	dimension of n x n square array.

Definition at line 46 of file LuSolver.cpp.
References UTIL_CHECK.
Referenced by Pscf::Homogeneous::Mixture::computePhi(), Pscf::Pspg::Continuous::AmIterator< D >::minimize←Coeff(), and Pscf::ChiInteraction::readParameters().

12.19.3.2 computeLU()

```
void Pscf::LuSolver::computeLU (
        const Matrix< double > & A )
```

Compute the LU decomposition for later use.

Parameters

A	the square matrix A in problem Ax=b.

Definition at line 63 of file LuSolver.cpp.
References Util::Matrix< Data >::capacity1(), Util::Matrix< Data >::capacity2(), and UTIL_CHECK.
Referenced by Pscf::Homogeneous::Mixture::computePhi(), Pscf::Pspg::Continuous::AmIterator< D >::minimize↩ Coeff(), and Pscf::ChiInteraction::readParameters().

12.19.3.3 solve()

```
void Pscf::LuSolver::solve (
        Array< double > & b,
        Array< double > & x )
```

Solve Ax = b for known b to compute x.

Parameters

b	the RHS vector
x	the solution vector

Definition at line 83 of file LuSolver.cpp.
References Util::Array< Data >::capacity(), Util::Array< Data >::cArray(), and UTIL_CHECK.
Referenced by Pscf::Homogeneous::Mixture::computePhi(), and Pscf::Pspg::Continuous::AmIterator< D >::minimize↩ Coeff().

12.19.3.4 inverse()

```
void Pscf::LuSolver::inverse (
        Matrix< double > & inv )
```

Compute inverse of matrix A.

Parameters

inv	inverse of matrix A (output)

Definition at line 104 of file LuSolver.cpp.
References Util::Matrix< Data >::cArray(), and UTIL_CHECK.
Referenced by Pscf::ChiInteraction::readParameters().
The documentation for this class was generated from the following files:

- LuSolver.h
- LuSolver.cpp

12.20 Pscf::Mesh< D > Class Template Reference

Description of a regular grid of points in a periodic domain.

```
#include <Mesh.h>
```

Public Member Functions

- Mesh ()

Default constructor.

- Mesh (const IntVec< D > &dimensions)

  Constructor.
- void setDimensions (const IntVec< D > &dimensions)

  Set the grid dimensions in all directions.
- IntVec< D > dimensions () const

  Get an IntVec<D> of the grid dimensions.
- int dimension (int i) const

  Get grid dimension along Cartesian direction i.
- int size () const

  Get total number of grid points.
- IntVec< D > position (int rank) const

  Get the position IntVec<D> of a grid point with a specified rank.
- int rank (const IntVec< D > &position) const

  Get the rank of a grid point with specified position.
- bool isInMesh (int coordinate, int i) const

  Is this coordinate in range?
- bool isInMesh (IntVec< D > &position) const

  Is this IntVec<D> grid position within the grid?
- int shift (int &coordinate, int i) const

  Shift a periodic coordinate into range.
- IntVec< D > shift (IntVec< D > &position) const

  Shift a periodic position into primary grid.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize to/from an archive.

Friends

- std::ostream & operator<< (std::ostream &, Mesh< D > &)

  Output stream inserter for writing a Mesh<D>::LatticeSystem.
- std::istream & operator>> (std::istream &, Mesh< D > &)

  Input stream extractor for reading a Mesh<D> object.

12.20.1 Detailed Description

template<int D>
class Pscf::Mesh< D >

Description of a regular grid of points in a periodic domain.
The coordinates of a point on a grid form an IntVec<D>, referred to here as a grid position. Each element of a grid position must lie in the range 0 <= position[i] < dimension(i), where i indexes a Cartesian axis, and dimension(i) is the dimension of the grid along axis i.
Each grid position is also assigned a non-negative integer rank. Mesh position ranks are ordered sequentially like elements in a multi-dimensional C array, with the last coordinate being the most rapidly varying.
Definition at line 21 of file Mesh.h.

12.20.2 Constructor & Destructor Documentation

12.20.2.1 Mesh() [1/2] `template<int D>`

```
Pscf::Mesh< D >::Mesh
```

Default constructor.
Definition at line 21 of file Mesh.tpp.
References Pscf::Mesh< D >::dimensions(), and Pscf::Mesh< D >::setDimensions().

12.20.2.2 Mesh() [2/2] `template<int D>`

```
Pscf::Mesh< D >::Mesh (
            const IntVec< D > & dimensions )
```

Constructor.

Parameters

dimensions	IntVec< D> of grid dimensions

Definition at line 31 of file Mesh.tpp.
References Pscf::Mesh< D >::dimensions(), and Pscf::Mesh< D >::setDimensions().

12.20.3 Member Function Documentation

12.20.3.1 setDimensions() `template<int D>`

```
void Pscf::Mesh< D >::setDimensions (
            const IntVec< D > & dimensions )
```

Set the grid dimensions in all directions.

Parameters

dimensions	IntVec< D> of grid dimensions.

Definition at line 40 of file Mesh.tpp.
References UTIL_THROW.
Referenced by Pscf::Mesh< D >::Mesh(), and Pscf::operator>>().

12.20.3.2 dimensions() `template<int D>`

```
IntVec< D > Pscf::Mesh< D >::dimensions  [inline]
```

Get an IntVec< D> of the grid dimensions.
Definition at line 202 of file Mesh.h.
Referenced by Pscf::Pspg::Continuous::Propagator< D >::allocate(), Pscf::Pspg::Continuous::Polymer< D >↩
::computeJoint(), Pscf::Mesh< D >::Mesh(), and Pscf::Pspg::Continuous::Mixture< D >::setMesh().

12.20.3.3 dimension() `template<int D>`

```
int Pscf::Mesh< D >::dimension (
            int i ) const  [inline]
```

Get grid dimension along Cartesian direction i.

Parameters

i	index of Cartesian direction 0 <=i < Util::Dimension

Definition at line 206 of file Mesh.h.
References Util::Dimension.

12.20.3.4 size()

```
template<int D>
int Pscf::Mesh< D >::size [inline]
```

Get total number of grid points.
Definition at line 214 of file Mesh.h.
Referenced by Pscf::Pspg::Continuous::Propagator< D >::allocate(), Pscf::Pspg::Continuous::Polymer< D >::computeJoint(), Pscf::Pspg::FieldIo< D >::convertBasisToKGrid(), Pscf::Basis< D >::makeBasis(), and Pscf::Pspg::Continuous::Block< D >::setDiscretization().

12.20.3.5 position()

```
template<int D>
IntVec< D > Pscf::Mesh< D >::position (
        int rank ) const
```

Get the position IntVec<D> of a grid point with a specified rank.

Parameters

rank	integer rank of a grid point.

Returns

IntVec<D> containing coordinates of specified point.

Definition at line 74 of file Mesh.tpp.

12.20.3.6 rank()

```
template<int D>
int Pscf::Mesh< D >::rank (
        const IntVec< D > & position ) const
```

Get the rank of a grid point with specified position.

Parameters

position	integer position of a grid point

Returns

integer rank of specified grid point

Definition at line 58 of file Mesh.tpp.
Referenced by Pscf::Pspg::FieldIo< D >::convertBasisToKGrid(), and Pscf::Pspg::FieldIo< D >::convertKGridToBasis().

12.20.3.7 isInMesh() [1/2]

```
template<int D>
bool Pscf::Mesh< D >::isInMesh (
        int coordinate,
        int i ) const
```

Is this coordinate in range?

Parameters

coordinate	coordinate value for direction i
i	index for Cartesian direction

Returns

true iff 0 <= coordinate < dimension(i).

Definition at line 89 of file Mesh.tpp.

12.20.3.8 isInMesh() [2/2]

```
template<int D>
bool Pscf::Mesh< D >::isInMesh (
        IntVec< D > & position ) const
```

Is this IntVec<D> grid position within the grid?
Returns true iff 0 <= coordinate[i] < dimension(i) for all i.

Parameters

position	grid point position

Returns

true iff 0 <= coordinate[i] < dimension(i) for all i.

Definition at line 100 of file Mesh.tpp.

12.20.3.9 shift() [1/2]

```
template<int D>
int Pscf::Mesh< D >::shift (
        int & coordinate,
        int i ) const
```

Shift a periodic coordinate into range.
Upon return, the coordinate will be shifted to lie within the range 0 <= coordinate < dimension(i) by subtracting an integer multiple of dimension(i), giving coordinate - shift∗dimension(i). The return value is the required integer ‘shift’.

Parameters

coordinate	coordinate in Cartesian direction i.
i	index of Cartesian direction, i >= 0.

Returns

multiple of dimension(i) subtracted from input value.

Definition at line 113 of file Mesh.tpp.

12.20.3.10 shift() [2/2]

```
template<int D>
IntVec< D > Pscf::Mesh< D >::shift (
        IntVec< D > & position ) const
```

Shift a periodic position into primary grid.

Upon return, each element of the parameter position is shifted to lie within the range 0 <= position[i] < dimension(i) by adding or subtracting an integer multiple of dimension(i). The IntVec<D> of shift values is returned.

Parameters

position	IntVec<D> position within a grid.

Returns

IntVec<D> of integer shifts.

Definition at line 126 of file Mesh.tpp.

12.20.3.11 serialize() template<int D>

```
template<class Archive >
void Pscf::Mesh< D >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize to/from an archive.

Parameters

ar	archive
version	archive version id

Definition at line 222 of file Mesh.h.

12.20.4 Friends And Related Function Documentation

12.20.4.1 operator<< template<int D>

```
std::ostream& operator<< (
            std::ostream & out,
            Mesh< D > & mesh ) [friend]
```

Output stream inserter for writing a Mesh<D>::LatticeSystem.

Parameters

out	output stream
mesh	Mesh<D> to be written

Returns

modified output stream

Definition at line 149 of file Mesh.tpp.

12.20.4.2 operator>> template<int D>

```
std::istream& operator>> (
            std::istream & in,
            Mesh< D > & mesh ) [friend]
```

Input stream extractor for reading a Mesh<D> object.

Parameters

in	input stream
mesh	Mesh<D> object to be read

Returns

modified input stream

Definition at line 136 of file Mesh.tpp.
The documentation for this class was generated from the following files:

- Mesh.h
- Mesh.tpp

12.21 Pscf::MeshIterator< D > Class Template Reference

Base class for mesh iterator class template.

```
#include <MeshIterator.h>
```

Public Member Functions

- MeshIterator ()

  Default constructor.
- MeshIterator (const IntVec< D > &dimensions)

  Constructor.
- void setDimensions (const IntVec< D > &dimensions)

  Set the grid dimensions in all directions.
- void begin ()

  Set iterator to the first point in the mesh.
- void operator++ ()

  Increment iterator to next mesh point.
- bool atEnd () const

  Is this the end (i.e., one past the last point)?
- IntVec< D > position () const

  Get current position in the grid, as integer vector.
- int position (int i) const

  Get component i of the current position vector.
- int rank () const

  Get the rank of current element.

12.21.1 Detailed Description

template<int D>
class Pscf::MeshIterator< D >

Base class for mesh iterator class template.
A mesh iterator iterates over the points of a mesh, keeping track of both the IntVec<D> position and integer rank of the current point as it goes.
Definition at line 28 of file MeshIterator.h.

12.21.2 Constructor & Destructor Documentation

12.21.2.1 MeshIterator() [1/2]

```
template<int D>
Pscf::MeshIterator< D >::MeshIterator
```

Default constructor.

Definition at line 205 of file MeshIterator.h.

12.21.2.2 MeshIterator() [2/2]

```
template<int D>
Pscf::MeshIterator< D >::MeshIterator (
        const IntVec< D > & dimensions )
```

Constructor.

Parameters

dimensions	IntVec<D> of grid dimensions

Definition at line 216 of file MeshIterator.h.

References Pscf::MeshIterator< D >::setDimensions().

12.21.3 Member Function Documentation

12.21.3.1 setDimensions()

```
template<int D>
void Pscf::MeshIterator< D >::setDimensions (
        const IntVec< D > & dimensions )
```

Set the grid dimensions in all directions.

Parameters

dimensions	IntVec<D> of grid dimensions.

Definition at line 229 of file MeshIterator.h.

References UTIL_THROW.

Referenced by Pscf::MeshIterator< D >::MeshIterator(), Pscf::Pspg::Continuous::Block< D >::setupUnitCell(), and Pscf::Pspg::Continuous::Mixture< D >::setupUnitCell().

12.21.3.2 begin()

```
template<int D>
void Pscf::MeshIterator< D >::begin
```

Set iterator to the first point in the mesh.

Definition at line 248 of file MeshIterator.h.

Referenced by Pscf::Basis< D >::isValid(), Pscf::Pspg::Continuous::Block< D >::setupUnitCell(), and Pscf::Pspg::↩
Continuous::Mixture< D >::setupUnitCell().

12.21.3.3 operator++()

```
template<int D>
void Pscf::MeshIterator< D >::operator++  [inline]
```

Increment iterator to next mesh point.

Definition at line 136 of file MeshIterator.h.

12.21.3.4 atEnd() `template<int D>`

`bool Pscf::MeshIterator< D >::atEnd [inline]`

Is this the end (i.e., one past the last point)?

Definition at line 150 of file MeshIterator.h.

Referenced by Pscf::Basis< D >::isValid(), Pscf::Pspg::Continuous::Block< D >::setupUnitCell(), and Pscf::Pspg::Continuous::Mixture< D >::setupUnitCell().

12.21.3.5 position() [1/2] `template<int D>`

`IntVec< D > Pscf::MeshIterator< D >::position [inline]`

Get current position in the grid, as integer vector.

Definition at line 120 of file MeshIterator.h.

Referenced by Pscf::Basis< D >::isValid(), Pscf::Pspg::Continuous::Block< D >::setupUnitCell(), and Pscf::Pspg::Continuous::Mixture< D >::setupUnitCell().

12.21.3.6 position() [2/2] `template<int D>`

```
int Pscf::MeshIterator< D >::position (
            int i ) const [inline]
```

Get component i of the current position vector.

Parameters

i	index of Cartesian direction 0 <=i < D.

Definition at line 124 of file MeshIterator.h.

12.21.3.7 rank() `template<int D>`

`int Pscf::MeshIterator< D >::rank [inline]`

Get the rank of current element.

Definition at line 132 of file MeshIterator.h.

Referenced by Pscf::Pspg::Continuous::Block< D >::setupUnitCell(), and Pscf::Pspg::Continuous::Mixture< D >::setupUnitCell().

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

- MeshIterator.h

12.22 Pscf::MixtureTmpl< TP, TS > Class Template Reference

A mixture of polymer and solvent species.

`#include <MixtureTmpl.h>`

Inheritance diagram for Pscf::MixtureTmpl< TP, TS >:

classPscf_1_1MixtureTmpl-eps-converted-to.pdf

Public Types

- typedef TP Polymer

  Polymer species solver type.
- typedef TS Solvent

  Solvent species solver type.

Public Member Functions

- MixtureTmpl ()

  Constructor.
- ∼MixtureTmpl ()

  Destructor.
- virtual void readParameters (std::istream &in)

  Read parameters from file and initialize.

Accessors (by non-const reference)

- Monomer & monomer (int id)

  Get a Monomer type descriptor.
- Polymer & polymer (int id)

  Get a polymer object.
- Solvent & solvent (int id)

  Set a solvent solver object.

Accessors (by value)

- int nMonomer () const

  Get number of monomer types.
- int nPolymer () const

  Get number of polymer species.
- int nSolvent () const

  Get number of solvent (point particle) species.

Additional Inherited Members

12.22.1 Detailed Description

template<class TP, class TS>
class Pscf::MixtureTmpl< TP, TS >

A mixture of polymer and solvent species.
Definition at line 26 of file MixtureTmpl.h.

12.22.2 Member Typedef Documentation

12.22.2.1 Polymer `template<class TP , class TS >`

`typedef TP Pscf::MixtureTmpl< TP, TS >::Polymer`

Polymer species solver type.

Definition at line 35 of file MixtureTmpl.h.

12.22.2.2 Solvent `template<class TP , class TS >`

`typedef TS Pscf::MixtureTmpl< TP, TS >::Solvent`

Solvent species solver type.

Definition at line 40 of file MixtureTmpl.h.

12.22.3 Constructor & Destructor Documentation

12.22.3.1 MixtureTmpl() `template<class TP , class TS >`

`Pscf::MixtureTmpl< TP, TS >::MixtureTmpl`

Constructor.

Definition at line 176 of file MixtureTmpl.h.

12.22.3.2 ∼MixtureTmpl() `template<class TP , class TS >`

`Pscf::MixtureTmpl< TP, TS >::~MixtureTmpl`

Destructor.

Definition at line 190 of file MixtureTmpl.h.

12.22.4 Member Function Documentation

12.22.4.1 readParameters() `template<class TP , class TS >`

```
void Pscf::MixtureTmpl< TP, TS >::readParameters (
            std::istream & in ) [virtual]
```

Read parameters from file and initialize.

Parameters

in	input parameter file

Reimplemented from Util::ParamComposite.

Reimplemented in Pscf::Pspg::Continuous::Mixture< D >.

Definition at line 197 of file MixtureTmpl.h.

12.22.4.2 monomer() `template<class TP , class TS >`

```
Monomer & Pscf::MixtureTmpl< TP, TS >::monomer (
            int id ) [inline]
```

Get a Monomer type descriptor.

Parameters

id	integer monomer type index (0 <= id < nMonomer)

Definition at line 159 of file MixtureTmpl.h.

12.22.4.3 polymer() `template<class TP , class TS >`

```
TP & Pscf::MixtureTmpl< TP, TS >::polymer (
             int id )  [inline]
```

Get a polymer object.

Parameters

id	integer polymer species index (0 <= id < nPolymer)

Definition at line 163 of file MixtureTmpl.h.

12.22.4.4 solvent() `template<class TP , class TS >`

```
TS & Pscf::MixtureTmpl< TP, TS >::solvent (
             int id )  [inline]
```

Set a solvent solver object.

Parameters

id	integer solvent species index (0 <= id < nSolvent)

Definition at line 167 of file MixtureTmpl.h.

12.22.4.5 nMonomer() `template<class TP , class TS >`

```
int Pscf::MixtureTmpl< TP, TS >::nMonomer  [inline]
```

Get number of monomer types.
Definition at line 147 of file MixtureTmpl.h.

12.22.4.6 nPolymer() `template<class TP , class TS >`

```
int Pscf::MixtureTmpl< TP, TS >::nPolymer  [inline]
```

Get number of polymer species.
Definition at line 151 of file MixtureTmpl.h.

12.22.4.7 nSolvent() `template<class TP , class TS >`

```
int Pscf::MixtureTmpl< TP, TS >::nSolvent  [inline]
```

Get number of solvent (point particle) species.
Definition at line 155 of file MixtureTmpl.h.
The documentation for this class was generated from the following file:

- MixtureTmpl.h

12.23 Pscf::Monomer Class Reference

Descriptor for a monomer or particle type.
`#include <Monomer.h>`

Public Member Functions

- Monomer ()

  Constructor.
- int id () const

  Unique integer index for monomer type.
- double step () const

  Statistical segment length (random walk step size).
- std::string name () const

  Monomer name string.
- template<class Archive >
  void serialize (Archive ar, const unsigned int version)

  Serialize to or from an archive.

Friends

- std::istream & operator>> (std::istream &in, Monomer &monomer)

  istream extractor for a Monomer.
- std::ostream & operator<< (std::ostream &out, const Monomer &monomer)

  ostream inserter for a Monomer.

12.23.1 Detailed Description

Descriptor for a monomer or particle type.
Definition at line 22 of file Monomer.h.

12.23.2 Constructor & Destructor Documentation

12.23.2.1 Monomer() `Pscf::Monomer::Monomer ( )`

Constructor.
Definition at line 13 of file Monomer.cpp.

12.23.3 Member Function Documentation

12.23.3.1 id() `int Pscf::Monomer::id ( ) const [inline]`

Unique integer index for monomer type.
Definition at line 94 of file Monomer.h.

12.23.3.2 step() `double Pscf::Monomer::step ( ) const [inline]`

Statistical segment length (random walk step size).
Definition at line 100 of file Monomer.h.

12.23.3.3 name() `std::string Pscf::Monomer::name ( ) const [inline]`

Monomer name string.

Definition at line 106 of file Monomer.h.

12.23.3.4 serialize() `template<class Archive >`

```
void Pscf::Monomer::serialize (
            Archive ar,
            const unsigned int version )
```

Serialize to or from an archive.

Parameters

ar	Archive object
version	archive format version index

Definition at line 113 of file Monomer.h.

12.23.4 Friends And Related Function Documentation

12.23.4.1 operator>> `std::istream& operator>> (`

```
            std::istream & in,
            Monomer & monomer ) [friend]
```

istream extractor for a Monomer.

Parameters

in	input stream
monomer	Monomer to be read from stream

Returns

modified input stream

Definition at line 22 of file Monomer.cpp.

12.23.4.2 operator<< `std::ostream& operator<< (`

```
            std::ostream & out,
            const Monomer & monomer ) [friend]
```

ostream inserter for a Monomer.

Parameters

out	output stream
monomer	Monomer to be written to stream

Returns

modified output stream

Definition at line 33 of file Monomer.cpp.
The documentation for this class was generated from the following files:

- Monomer.h
- Monomer.cpp

12.24 Pscf::PolymerTmpl< Block > Class Template Reference

Descriptor and MDE solver for an acyclic block polymer.
`#include <PolymerTmpl.h>`
Inheritance diagram for Pscf::PolymerTmpl< Block >:

classPscf_1_1PolymerTmpl-eps-converted-to.pdf

Public Member Functions

- PolymerTmpl ()

  Constructor.
- ∼PolymerTmpl ()

  Destructor.
- virtual void readParameters (std::istream &in)

  Read and initialize.
- virtual void solve ()

  Solve modified diffusion equation.

Accessors (objects, by reference)

- Block & block (int id)

  Get a specified Block.
- const Block & block (int id) const

  Get a specified Block by const reference.
- const Vertex & vertex (int id) const

  Get a specified Vertex by const reference.
- Propagator & propagator (int blockId, int directionId)

  Get propagator for a specific block and direction.
- const Propagator & propagator (int blockId, int directionId) const

  Get a const propagator for a specific block and direction.
- Propagator & propagator (int id)

  Get propagator indexed in order of computation.
- const Pair< int > & propagatorId (int i) const

  Propagator identifier, indexed by order of computation.

Accessors (by value)

- int nBlock () const

  Number of blocks.
- int nVertex () const

  Number of vertices (junctions and chain ends).
- int nPropagator () const

  Number of propagators (twice nBlock).
- double length () const

  Total length of all blocks = volume / reference volume.
- double **Q** () const
- virtual void **makePlan** ()

Additional Inherited Members

12.24.1 Detailed Description

template<class Block>
class Pscf::PolymerTmpl< Block >

Descriptor and MDE solver for an acyclic block polymer.
A PolymerTmpl<Block> object has arrays of Block and Vertex objects. Each Block has two propagator MDE solver objects. The compute() member function solves the modified diffusion equation (MDE) for the entire molecule and computes monomer concentration fields for all blocks.
Definition at line 42 of file PolymerTmpl.h.

12.24.2 Constructor & Destructor Documentation

12.24.2.1 PolymerTmpl() `template<class Block >`

`Pscf::PolymerTmpl< Block >::PolymerTmpl`

Constructor.
Definition at line 319 of file PolymerTmpl.h.

12.24.2.2 ∼PolymerTmpl() `template<class Block >`

`Pscf::PolymerTmpl< Block >::~PolymerTmpl ( ) [default]`

Destructor.

12.24.3 Member Function Documentation

12.24.3.1 readParameters() `template<class Block >`

```
void Pscf::PolymerTmpl< Block >::readParameters (
            std::istream & in ) [virtual]
```

Read and initialize.

Parameters

in	input parameter stream

Reimplemented from Util::ParamComposite.

Definition at line 338 of file PolymerTmpl.h.

12.24.3.2 solve() `template<class Block >`

`void Pscf::PolymerTmpl< Block >::solve [virtual]`

Solve modified diffusion equation.

Upon return, q functions and block concentration fields are computed for all propagators and blocks.

Definition at line 483 of file PolymerTmpl.h.

12.24.3.3 block() [1/2] `template<class Block >`

```
Block & Pscf::PolymerTmpl< Block >::block (
            int id ) [inline]
```

Get a specified Block.

Parameters

id	block index, 0 <= id < nBlock

Definition at line 254 of file PolymerTmpl.h.

12.24.3.4 block() [2/2] `template<class Block >`

```
const Block & Pscf::PolymerTmpl< Block >::block (
            int id ) const [inline]
```

Get a specified Block by const reference.

Parameters

id	block index, 0 <= id < nBlock

Definition at line 263 of file PolymerTmpl.h.

12.24.3.5 vertex() `template<class Block >`

```
const Vertex & Pscf::PolymerTmpl< Block >::vertex (
            int id ) const [inline]
```

Get a specified Vertex by const reference.

Both chain ends and junctions are vertices.

Parameters

id	vertex index, 0 <= id < nVertex

Definition at line 245 of file PolymerTmpl.h.

12.24.3.6 propagator() [1/3] `template<class Block >`

```
Block::Propagator & Pscf::PolymerTmpl< Block >::propagator (
            int blockId,
            int directionId ) [inline]
```

Get propagator for a specific block and direction.

Parameters

blockId	integer index of associated block
direction↩ *Id*	integer index for direction (0 or 1)

Definition at line 285 of file PolymerTmpl.h.

12.24.3.7 propagator() [2/3]

```
template<class Block >
const Block::Propagator & Pscf::PolymerTmpl< Block >::propagator (
            int blockId,
            int directionId ) const [inline]
```

Get a const propagator for a specific block and direction.

Parameters

blockId	integer index of associated block
direction↩ *Id*	integer index for direction (0 or 1)

Definition at line 296 of file PolymerTmpl.h.

12.24.3.8 propagator() [3/3]

```
template<class Block >
Block::Propagator & Pscf::PolymerTmpl< Block >::propagator (
            int id ) [inline]
```

Get propagator indexed in order of computation.

The propagator index must satisfy 0 <= id < 2∗nBlock.

Parameters

id	integer index, in order of computation plan

Definition at line 307 of file PolymerTmpl.h.

12.24.3.9 propagatorId()

```
template<class Block >
const Pair< int > & Pscf::PolymerTmpl< Block >::propagatorId (
            int i ) const [inline]
```

Propagator identifier, indexed by order of computation.

The return value is a pair of integers. The first of which is a block index between 0 and nBlock - 1 and the second is a direction id, which must be 0 or 1.

Definition at line 272 of file PolymerTmpl.h.

12.24.3.10 nBlock()

```
template<class Block >
int Pscf::PolymerTmpl< Block >::nBlock [inline]
```

Number of blocks.

Definition at line 213 of file PolymerTmpl.h.

12.24.3.11 nVertex() `template<class Block >`

`int Pscf::PolymerTmpl< Block >::nVertex [inline]`

Number of vertices (junctions and chain ends).

Definition at line 204 of file PolymerTmpl.h.

12.24.3.12 nPropagator() `template<class Block >`

`int Pscf::PolymerTmpl< Block >::nPropagator [inline]`

Number of propagators (twice nBlock).

Definition at line 222 of file PolymerTmpl.h.

12.24.3.13 length() `template<class Block >`

`double Pscf::PolymerTmpl< Block >::length [inline]`

Total length of all blocks = volume / reference volume.

Definition at line 231 of file PolymerTmpl.h.

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

- PolymerTmpl.h

12.25 Pscf::PropagatorTmpl< TP > Class Template Reference

Template for propagator classes.

`#include <PropagatorTmpl.h>`

Public Member Functions

- PropagatorTmpl ()

  Constructor.

Mutators

- void setDirectionId (int directionId)

  Associate this propagator with a direction index.
- void setPartner (const TP &partner)

  Set the partner of this propagator.
- void addSource (const TP &source)

  Add a propagator to the list of sources for this one.
- void setIsSolved (bool isSolved)

  Set the isSolved flag to true or false.

Accessors

- const TP & source (int id) const

  Get a source propagator.
- const TP & partner () const

  Get partner propagator.
- int directionId () const

  Get direction index for this propagator.
- int nSource () const

  Number of source / prerequisite propagators.
- bool hasPartner () const

  Does this have a partner propagator?

- bool isSolved () const

  Has the modified diffusion equation been solved?
- bool isReady () const

  Are all source propagators are solved?

12.25.1 Detailed Description

template<class TP>
class Pscf::PropagatorTmpl< TP >

Template for propagator classes.

The template argument TP should be a concrete propagator class that is derived from the template PropagatorTmpl<TP>. By convention, each implementation of SCFT is defined in a different sub-namespace of namespace Pscf. For each such implementation, there is a concrete propagator class, named Propagator by convention, that is a subclass of the template instance PropagatorTmpl<Propagator>, using the syntax shown below:

```
class Propagator : public PropagatorTmpl<Propagator>
{
 ...
};
```

This usage is an example of the so-called "curiously recurring template pattern" (CRTP). It is used here to allow the template PropagatorTmpl<Propagator> to have a member variables that store pointers to other instances of derived class Propagator (or TP).

The TP propagator class is used in templates BlockTmpl, PolymerTmpl and SystemTmpl. The usage in those templates require that it define the following public typedefs and member functions:

```
class TP : public PropagatorTmpl<TP>
{
public:
    // Chemical potential field type.
    typedef DArray<double> WField;
    // Monomer concentration field type.
    typedef DArray<double> CField;
    // Solve the modified diffusion equation for this direction.
    void solve();
    // Compute and return the molecular partition function Q.
    double computeQ();
};
```

The typedefs WField and CField define the types of the objects used to represent a chemical potential field for a particular monomer type and a monomer concentration field. In the above example, both of these type names are defined to be synonyms for DArray<double>, i.e., for dynamically allocated arrays of double precision floating point numbers. Other implementations may use more specialized types.

Definition at line 76 of file PropagatorTmpl.h.

12.25.2 Constructor & Destructor Documentation

12.25.2.1 PropagatorTmpl() `template<class TP >`

`Pscf::PropagatorTmpl< TP >::PropagatorTmpl`

Constructor.

Definition at line 226 of file PropagatorTmpl.h.

12.25.3 Member Function Documentation

12.25.3.1 setDirectionId() `template<class TP >`

```
void Pscf::PropagatorTmpl< TP >::setDirectionId (
              int directionId )
```

Associate this propagator with a direction index.

Parameters

direction↩ Id	direction = 0 or 1.

Definition at line 237 of file PropagatorTmpl.h.

12.25.3.2 setPartner() `template<class TP >`

```
void Pscf::PropagatorTmpl< TP >::setPartner (
            const TP & partner )
```

Set the partner of this propagator.
The partner of a propagator is the propagator for the same block that propagates in the opposite direction.

Parameters

partner	reference to partner propagator

Definition at line 244 of file PropagatorTmpl.h.

12.25.3.3 addSource() `template<class TP >`

```
void Pscf::PropagatorTmpl< TP >::addSource (
            const TP & source )
```

Add a propagator to the list of sources for this one.
A source is a propagator that terminates at the root vertex of this one and is needed to compute the initial condition for this one, and that thus must be computed before this.

Parameters

source	reference to source propagator

Definition at line 251 of file PropagatorTmpl.h.

12.25.3.4 setIsSolved() `template<class TP >`

```
void Pscf::PropagatorTmpl< TP >::setIsSolved (
            bool isSolved )
```

Set the isSolved flag to true or false.
Definition at line 269 of file PropagatorTmpl.h.

12.25.3.5 source() `template<class TP >`

```
const TP & Pscf::PropagatorTmpl< TP >::source (
            int id ) const [inline]
```

Get a source propagator.

Parameters

id	index of source propagator, < nSource

Definition at line 202 of file PropagatorTmpl.h.

12.25.3.6 partner() template<class TP >

const TP & Pscf::PropagatorTmpl< TP >::partner

Get partner propagator.

Definition at line 258 of file PropagatorTmpl.h.

12.25.3.7 directionId() template<class TP >

int Pscf::PropagatorTmpl< TP >::directionId [inline]

Get direction index for this propagator.

Definition at line 187 of file PropagatorTmpl.h.

12.25.3.8 nSource() template<class TP >

int Pscf::PropagatorTmpl< TP >::nSource [inline]

Number of source / prerequisite propagators.

Definition at line 194 of file PropagatorTmpl.h.

12.25.3.9 hasPartner() template<class TP >

bool Pscf::PropagatorTmpl< TP >::hasPartner [inline]

Does this have a partner propagator?

Definition at line 210 of file PropagatorTmpl.h.

12.25.3.10 isSolved() template<class TP >

bool Pscf::PropagatorTmpl< TP >::isSolved [inline]

Has the modified diffusion equation been solved?

Definition at line 217 of file PropagatorTmpl.h.

12.25.3.11 isReady() template<class TP >

bool Pscf::PropagatorTmpl< TP >::isReady

Are all source propagators are solved?

Definition at line 276 of file PropagatorTmpl.h.

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

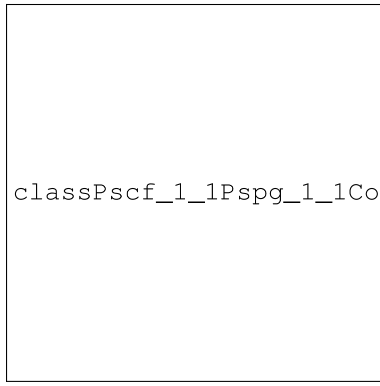
- PropagatorTmpl.h

12.26 Pscf::Pspg::Continuous::AmIterator< D > Class Template Reference

Anderson mixing iterator for the pseudo spectral method.

#include <AmIterator.h>

Inheritance diagram for Pscf::Pspg::Continuous::AmIterator< D >:



Public Member Functions

- AmIterator ()

    Default constructor.
- AmIterator (System< D > *system)

    Constructor.
- ∼AmIterator ()

    Destructor.
- void readParameters (std::istream &in)

    Read all parameters and initialize.
- void allocate ()

    Allocate all arrays.
- int solve ()

    Iterate to a solution.
- double epsilon ()

    Getter for epsilon.
- int maxHist ()

    Getter for the maximum number of field histories to convolute into a new field.
- int maxItr ()

    Getter for the maximum number of iteration before convergence.
- void computeDeviation ()

    Compute the deviation of wFields from a mean field solution.
- bool isConverged (int itr)

    Compute the error from deviations of wFields and compare with epsilon_.
- int minimizeCoeff (int itr)

    Determine the coefficients that would minimize invertMatrix_ Umn.
- void buildOmega (int itr)

    Rebuild wFields for the next iteration from minimized coefficients.

Additional Inherited Members

12.26.1 Detailed Description

template<int D>
class Pscf::Pspg::Continuous::AmIterator< D >

Anderson mixing iterator for the pseudo spectral method.
Definition at line 36 of file pgc/iterator/AmIterator.h.

12.26.2 Constructor & Destructor Documentation

12.26.2.1 AmIterator() [1/2] template<int D>

`Pscf::Pspg::Continuous::AmIterator< D >::AmIterator`

Default constructor.

Definition at line 28 of file pgc/iterator/AmIterator.tpp.

References Util::ParamComposite::setClassName().

12.26.2.2 AmIterator() [2/2] template<int D>

```
Pscf::Pspg::Continuous::AmIterator< D >::AmIterator (
            System< D > * system ) [explicit]
```

Constructor.

Parameters

system	pointer to a system object

Definition at line 40 of file pgc/iterator/AmIterator.tpp.

References Util::ParamComposite::setClassName().

12.26.2.3 ∼AmIterator() template<int D>

`Pscf::Pspg::Continuous::AmIterator< D >::~AmIterator`

Destructor.

Definition at line 52 of file pgc/iterator/AmIterator.tpp.

12.26.3 Member Function Documentation

12.26.3.1 readParameters() template<int D>

```
void Pscf::Pspg::Continuous::AmIterator< D >::readParameters (
            std::istream & in ) [virtual]
```

Read all parameters and initialize.

Parameters

in	input filestream

Reimplemented from Util::ParamComposite.

Definition at line 59 of file pgc/iterator/AmIterator.tpp.

12.26.3.2 allocate() template<int D>

`void Pscf::Pspg::Continuous::AmIterator< D >::allocate`

Allocate all arrays.

Definition at line 69 of file pgc/iterator/AmIterator.tpp.

References Pscf::Pspg::ThreadGrid::setThreadsLogical().

12.26.3.3 solve() template<int D>

`int Pscf::Pspg::Continuous::AmIterator< D >::solve [virtual]`

Iterate to a solution.

Implements Pscf::Pspg::Continuous::Iterator< D >.

Definition at line 118 of file pgc/iterator/AmIterator.tpp.

References Util::Log::file(), Util::Timer::now(), Util::Timer::start(), Util::Timer::stop(), and Util::Timer::time().

12.26.3.4 epsilon() template<int D>

`double Pscf::Pspg::Continuous::AmIterator< D >::epsilon [inline]`

Getter for epsilon.

Definition at line 196 of file pgc/iterator/AmIterator.h.

12.26.3.5 maxHist() template<int D>

`int Pscf::Pspg::Continuous::AmIterator< D >::maxHist [inline]`

Getter for the maximum number of field histories to convolute into a new field.

Definition at line 202 of file pgc/iterator/AmIterator.h.

12.26.3.6 maxItr() template<int D>

`int Pscf::Pspg::Continuous::AmIterator< D >::maxItr [inline]`

Getter for the maximum number of iteration before convergence.

Definition at line 208 of file pgc/iterator/AmIterator.h.

12.26.3.7 computeDeviation() template<int D>

`void Pscf::Pspg::Continuous::AmIterator< D >::computeDeviation`

Compute the deviation of wFields from a mean field solution.

Definition at line 343 of file pgc/iterator/AmIterator.tpp.

References Pscf::Pspg::ThreadGrid::setThreadsLogical().

12.26.3.8 isConverged() template<int D>

```
bool Pscf::Pspg::Continuous::AmIterator< D >::isConverged (
            int itr )
```

Compute the error from deviations of wFields and compare with epsilon_.

Returns

true for error < epsilon and false for error >= epsilon

Definition at line 523 of file pgc/iterator/AmIterator.tpp.

References Util::Log::file().

12.26.3.9 minimizeCoeff() template<int D>

```
int Pscf::Pspg::Continuous::AmIterator< D >::minimizeCoeff (
            int itr )
```

Determine the coefficients that would minimize invertMatrix_ Umn.

Definition at line 586 of file pgc/iterator/AmIterator.tpp.

References Pscf::LuSolver::allocate(), Pscf::Pspg::RDField< D >::allocate(), Pscf::Pspg::DField< cudaReal >::c↩DField(), Pscf::LuSolver::computeLU(), Pscf::Pspg::DField< cudaReal >::deallocate(), Pscf::Pspg::ThreadGrid::set↩ThreadsLogical(), and Pscf::LuSolver::solve().

12.26.3.10 buildOmega() `template<int D>`

```
void Pscf::Pspg::Continuous::AmIterator< D >::buildOmega (
            int itr )
```

Rebuild wFields for the next iteration from minimized coefficients.

Definition at line 791 of file pgc/iterator/AmIterator.tpp.

References Pscf::Pspg::ThreadGrid::setThreadsLogical().

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

- pgc/iterator/AmIterator.h
- pgc/iterator/AmIterator.tpp

12.27 Pscf::Pspg::Continuous::Block< D > Class Template Reference

Block within a branched polymer.

```
#include <Block.h>
```

Inheritance diagram for Pscf::Pspg::Continuous::Block< D >:

classPscf_1_1Pspg_1_1Continuous_1_1Block-eps-converted-to.

Public Types

- typedef Pscf::Pspg::Continuous::Propagator< D >::Field Field

  Generic field (base class)
- typedef Pscf::Pspg::Continuous::Propagator< D >::WField WField

  Monomer chemical potential field.
- typedef Pscf::Pspg::Continuous::Propagator< D >::QField QField

  Constrained partition function q(r,s) for fixed s.

Public Member Functions

- Block ()

  Constructor.
- ∼Block ()

  Destructor.
- void setDiscretization (double ds, const Mesh< D > &mesh)

  Initialize discretization and allocate required memory.
- void setupUnitCell (const UnitCell< D > &unitCell, const WaveList< D > &wavelist)

  Setup parameters that depend on the unit cell.
- void setupSolver (WField const &w)

  Set solver for this block.
- void setupFFT ()

  Initialize FFT and batch FFT classes.
- void step (const cudaReal ∗q, cudaReal ∗qNew)

  Compute step of integration loop, from i to i+1.
- void computeStress (double prefactor)

Compute derivatives of free energy with respect to cell parameters.

- void computeInt (cudaReal ∗q, cudaReal ∗qs, int ic)

  Compute the integral using RI-K while the backword propagtors are being computed from "slices".
- double stress (int n)

  Get derivative of free energy with respect to a unit cell parameter.
- const Mesh< D > & mesh () const

  Return associated spatial Mesh by reference.
- double ds () const

  Contour length step size.
- int ns () const

  Number of contour length steps.
- FFT< D > & fft ()

  Return the fast Fourier transform object by reference.
- int id () const

  Get the id of this block.
- double length () const

  Get the length (number of monomers) in this block.
- int monomerId () const

  Get the monomer type id.
- void setId (int id)

  Set the id for this block.
- virtual void setLength (double length)

  Set the length of this block.
- void setMonomerId (int monomerId)

  Set the monomer id.
- void setVertexIds (int vertexAId, int vertexBId)

  Set indices of associated vertices.
- int vertexId (int i) const

  Get id of an associated vertex.
- const Pair< int > & vertexIds () const

  Get the pair of associated vertex ids.

12.27.1 Detailed Description

template<int D>
class Pscf::Pspg::Continuous::Block< D >

Block within a branched polymer.
(Continuous Gaussian chain)
Derived from BlockTmpl< Propagator<D> >. A BlockTmpl< Propagator<D> > has two Propagator<D> members and is derived from BlockDescriptor.
Definition at line 48 of file Block.h.

12.27.2 Member Typedef Documentation

12.27.2.1 Field `template<int D>`

`typedef Pscf::Pspg::Continuous::Propagator<D>::Field Pscf::Pspg::Continuous::Block< D >::Field`

Generic field (base class)

Definition at line 55 of file Block.h.

12.27.2.2 WField `template<int D>`

`typedef Pscf::Pspg::Continuous::Propagator<D>::WField Pscf::Pspg::Continuous::Block< D >::WField`

Monomer chemical potential field.

Definition at line 60 of file Block.h.

12.27.2.3 QField `template<int D>`

`typedef Pscf::Pspg::Continuous::Propagator<D>::QField Pscf::Pspg::Continuous::Block< D >::QField`

Constrained partition function q(r,s) for fixed s.

Definition at line 65 of file Block.h.

12.27.3 Constructor & Destructor Documentation

12.27.3.1 Block() `template<int D>`

`Pscf::Pspg::Continuous::Block< D >::Block`

Constructor.

Definition at line 328 of file Block.tpp.

References Pscf::BlockTmpl< Propagator< D > >::propagator().

12.27.3.2 ~Block() `template<int D>`

`Pscf::Pspg::Continuous::Block< D >::~Block`

Destructor.

Definition at line 341 of file Block.tpp.

12.27.4 Member Function Documentation

12.27.4.1 setDiscretization() `template<int D>`

```
void Pscf::Pspg::Continuous::Block< D >::setDiscretization (
            double ds,
            const Mesh< D > & mesh )
```

Initialize discretization and allocate required memory.

Parameters

ds	desired (optimal) value for contour length step
mesh	spatial discretization mesh

Definition at line 407 of file Block.tpp.

References Pscf::Pspg::ThreadGrid::setThreadsLogical(), Pscf::Mesh< D >::size(), and UTIL_CHECK.

12.27.4.2 setupUnitCell() `template<int D>`

```
void Pscf::Pspg::Continuous::Block< D >::setupUnitCell (
            const UnitCell< D > & unitCell,
            const WaveList< D > & wavelist )
```

Setup parameters that depend on the unit cell.

Parameters

unitCell	unit cell, defining cell dimensions
waveList	container for properties of recip wavevectors

Definition at line 665 of file Block.tpp.

References Pscf::MeshIterator< D >::atEnd(), Pscf::MeshIterator< D >::begin(), Pscf::UnitCellBase< D >::dksq(), Pscf::UnitCellBase< D >::ksq(), Pscf::UnitCellBase< D >::nParameter(), Pscf::MeshIterator< D >::position(), Pscf↩ ::MeshIterator< D >::rank(), and Pscf::MeshIterator< D >::setDimensions().

12.27.4.3 setupSolver() `template<int D>`

```
void Pscf::Pspg::Continuous::Block< D >::setupSolver (
            WField const & w )
```

Set solver for this block.

Parameters

w	chemical potential field for this monomer type

Definition at line 834 of file Block.tpp.

References Pscf::Pspg::DField< Data >::cDField(), Pscf::Pspg::ThreadGrid::setThreadsLogical(), and UTIL_CHECK.

12.27.4.4 setupFFT() `template<int D>`

```
void Pscf::Pspg::Continuous::Block< D >::setupFFT
```

Initialize FFT and batch FFT classes.

Definition at line 875 of file Block.tpp.

12.27.4.5 step() `template<int D>`

```
void Pscf::Pspg::Continuous::Block< D >::step (
            const cudaReal * q,
            cudaReal * qNew )
```

Compute step of integration loop, from i to i+1.

Apply pseudo-spectral algorithm

Definition at line 919 of file Block.tpp.

References Pscf::Pspg::ThreadGrid::setThreadsLogical(), and UTIL_CHECK.

12.27.4.6 computeStress() `template<int D>`

```
void Pscf::Pspg::Continuous::Block< D >::computeStress (
            double prefactor )
```

Compute derivatives of free energy with respect to cell parameters.

Parameters

waveList	container for properties of recip. latt. wavevectors.

Parameters

prefactor	prefactor = exp(mu_)/length(), where mu_ and length() is the chemical potential and chain length of the corresponding polymer, respectively.

Definition at line 1157 of file Block.tpp.

12.27.4.7 computeInt() template<int D>

```
void Pscf::Pspg::Continuous::Block< D >::computeInt (
            cudaReal * q,
            cudaReal * qs,
            int ic )
```

Compute the integral using RI-K while the backword propagtors are being computed from "slices".

Parameters

q	Foward propagator.
qs	Backward propagtor.
ic	Coefficient given by RI-K method.

Definition at line 1177 of file Block.tpp.
References Pscf::Pspg::ThreadGrid::setThreadsLogical(), and UTIL_CHECK.

12.27.4.8 stress() template<int D>

```
double Pscf::Pspg::Continuous::Block< D >::stress (
            int n ) [inline]
```

Get derivative of free energy with respect to a unit cell parameter.

Parameters

n	unit cell parameter index

Definition at line 336 of file Block.h.

12.27.4.9 mesh() template<int D>

```
const Mesh< D > & Pscf::Pspg::Continuous::Block< D >::mesh [inline]
```

Return associated spatial Mesh by reference.
Get Mesh by reference.
Definition at line 343 of file Block.h.
References UTIL_ASSERT.

12.27.4.10 ds() template<int D>

```
double Pscf::Pspg::Continuous::Block< D >::ds [inline]
```

Contour length step size.
Get number of contour steps.
Definition at line 329 of file Block.h.

12.27.4.11 ns() `template<int D>`

`int Pscf::Pspg::Continuous::Block< D >::ns [inline]`

Number of contour length steps.
Get number of contour steps.
Definition at line 322 of file Block.h.

12.27.4.12 fft() `template<int D>`

`FFT< D > & Pscf::Pspg::Continuous::Block< D >::fft [inline]`

Return the fast Fourier transform object by reference.
Definition at line 350 of file Block.h.

12.27.4.13 id() `template<int D>`

`int Pscf::BlockDescriptor::id [inline]`

Get the id of this block.
Definition at line 156 of file BlockDescriptor.h.

12.27.4.14 length() `template<int D>`

`double Pscf::BlockDescriptor::length [inline]`

Get the length (number of monomers) in this block.
Definition at line 180 of file BlockDescriptor.h.

12.27.4.15 monomerId() `template<int D>`

`int Pscf::BlockDescriptor::monomerId [inline]`

Get the monomer type id.
Definition at line 162 of file BlockDescriptor.h.

12.27.4.16 setId() `template<int D>`

`void Pscf::BlockDescriptor::setId`

Set the id for this block.

Parameters

id	integer index for this block

Definition at line 26 of file BlockDescriptor.cpp.

12.27.4.17 setLength() `template<int D>`

`void Pscf::BlockDescriptor::setLength`

Set the length of this block.
The "length" is steric volume / reference volume.

Parameters

length	block length (number of monomers).

Definition at line 47 of file BlockDescriptor.cpp.

12.27.4.18 setMonomerId() `template<int D>`

`void Pscf::BlockDescriptor::setMonomerId`

Set the monomer id.

Parameters

monomer↩ Id	integer id of monomer type (>=0)

Definition at line 41 of file BlockDescriptor.cpp.

12.27.4.19 setVertexIds() `template<int D>`

`void Pscf::BlockDescriptor::setVertexIds`

Set indices of associated vertices.

Parameters

vertex↩ AId	integer id of vertex A
vertex↩ BId	integer id of vertex B

Definition at line 32 of file BlockDescriptor.cpp.

12.27.4.20 vertexId() `template<int D>`

`int Pscf::BlockDescriptor::vertexId [inline]`

Get id of an associated vertex.

Parameters

i	index of vertex (0 or 1)

Definition at line 174 of file BlockDescriptor.h.

12.27.4.21 vertexIds() `template<int D>`

`const Pair< int > & Pscf::BlockDescriptor::vertexIds [inline]`

Get the pair of associated vertex ids.

Definition at line 168 of file BlockDescriptor.h.

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

- Block.h
- Block.tpp

12.28 Pscf::Pspg::Continuous::Iterator< D > Class Template Reference

Base class for iterative solvers for SCF equations.

`#include <Iterator.h>`

Inheritance diagram for Pscf::Pspg::Continuous::Iterator< D >:

classPscf_1_1Pspg_1_1Continuous_1_1Iterator-eps-converted-to.pd

Public Member Functions

- Iterator ()

    Default constructor.
- Iterator (System< D > ∗system)

    Constructor.
- ∼Iterator ()

    Destructor.
- virtual int solve ()=0

    Iterate to solution.

Additional Inherited Members

12.28.1 Detailed Description

template<int D>
class Pscf::Pspg::Continuous::Iterator< D >

Base class for iterative solvers for SCF equations.
Definition at line 32 of file pgc/iterator/Iterator.h.

12.28.2 Constructor & Destructor Documentation

12.28.2.1 Iterator() [1/2] `template<int D>`

`Pscf::Pspg::Continuous::Iterator< D >::Iterator`

Default constructor.
Definition at line 20 of file pgc/iterator/Iterator.tpp.

12.28.2.2 Iterator() [2/2] `template<int D>`

```
Pscf::Pspg::Continuous::Iterator< D >::Iterator (
            System< D > * system )
```

Constructor.

Parameters

system	parent System object

Definition at line 26 of file pgc/iterator/Iterator.tpp.
References Util::ParamComposite::setClassName().

12.28.2.3 ∼Iterator() `template<int D>`

`Pscf::Pspg::Continuous::Iterator< D >::∼Iterator`
Destructor.
Definition at line 33 of file pgc/iterator/Iterator.tpp.

12.28.3 Member Function Documentation

12.28.3.1 solve() `template<int D>`

`virtual int Pscf::Pspg::Continuous::Iterator< D >::solve ( ) [pure virtual]`
Iterate to solution.

Returns

error code: 0 for success, 1 for failure.

Implemented in Pscf::Pspg::Continuous::AmIterator< D >.
The documentation for this class was generated from the following files:

- pgc/iterator/Iterator.h
- pgc/iterator/Iterator.tpp

12.29 Pscf::Pspg::Continuous::Joint< D > Class Template Reference

12.29.1 Detailed Description

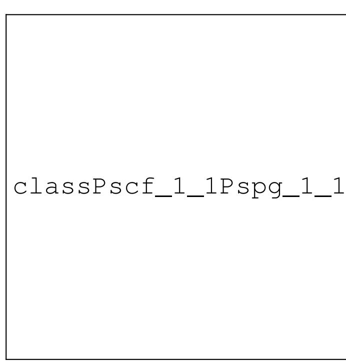
template< int D >
class Pscf::Pspg::Continuous::Joint< D >

Definition at line 19 of file Joint.h.
The documentation for this class was generated from the following files:

- Joint.h
- Joint.tpp

12.30 Pscf::Pspg::Continuous::Mixture< D > Class Template Reference

Solver for a mixture of polymers and solvents.
`#include <Mixture.h>`
Inheritance diagram for Pscf::Pspg::Continuous::Mixture< D >:



Public Types

- typedef Propagator< D >::WField WField
  Monomer chemical potential field type.
- typedef Propagator< D >::CField CField
  Monomer concentration or volume fraction field type.

Public Member Functions

- Mixture ()
  Constructor.
- ~Mixture ()
  Destructor.
- void readParameters (std::istream &in)
  Read all parameters and initialize.
- void setMesh (Mesh< D > const &mesh, UnitCell< D > &unitCell)
  Create an association with the mesh and allocate memory.
- void setupUnitCell (const UnitCell< D > &unitCell, const WaveList< D > &wavelist)
  Set unit cell parameters used in solver.
- void compute (DArray< WField > const &wFields, DArray< CField > &cFields)
  Compute concentrations.
- void computeStress (WaveList< D > &wavelist)
  Get monomer reference volume.
- double stress (int n)
  Get derivative of free energy w/ respect to cell parameter.
- double vMonomer () const
  Get monomer reference volume.

Protected Member Functions

- template<typename Type >
  ScalarParam< Type > & read (std::istream &in, const char *label, Type &value)
  Add and read a new required ScalarParam < Type > object.
- template<typename Type >
  ScalarParam< Type > & readOptional (std::istream &in, const char *label, Type &value)
  Add and read a new optional ScalarParam < Type > object.

Additional Inherited Members

12.30.1 Detailed Description

template<int D>
class Pscf::Pspg::Continuous::Mixture< D >

Solver for a mixture of polymers and solvents.
A Mixture contains a list of Polymer and Solvent objects. Each such object can solve the single-molecule statistical mechanics problem for an ideal gas of the associated species in a set of specified chemical potential fields, and thereby compute concentrations and single-molecule partition functions. A Mixture is thus both a chemistry descriptor and an ideal-gas solver.
A Mixture is associated with a Mesh<D> object, which models a spatial discretization mesh.
Definition at line 49 of file pgc/solvers/Mixture.h.

12.30.2 Member Typedef Documentation

12.30.2.1 WField `template<int D>`

`typedef Propagator<D>::WField Pscf::Pspg::Continuous::Mixture< D >::WField`

Monomer chemical potential field type.

Definition at line 58 of file pgc/solvers/Mixture.h.

12.30.2.2 CField `template<int D>`

`typedef Propagator<D>::CField Pscf::Pspg::Continuous::Mixture< D >::CField`

Monomer concentration or volume fraction field type.

Definition at line 63 of file pgc/solvers/Mixture.h.

12.30.3 Constructor & Destructor Documentation

12.30.3.1 Mixture() `template<int D>`

`Pscf::Pspg::Continuous::Mixture< D >::Mixture`

Constructor.

Definition at line 36 of file Mixture.tpp.

References Util::ParamComposite::setClassName().

12.30.3.2 ∼Mixture() `template<int D>`

`Pscf::Pspg::Continuous::Mixture< D >::~Mixture ( ) [default]`

Destructor.

12.30.4 Member Function Documentation

12.30.4.1 readParameters() `template<int D>`

```
void Pscf::Pspg::Continuous::Mixture< D >::readParameters (
            std::istream & in ) [virtual]
```

Read all parameters and initialize.

This function reads in a complete description of the chemical composition and structure of all species, as well as the target contour length step size ds.

Parameters

in	input parameter stream

Reimplemented from Pscf::MixtureTmpl< Polymer< D >, Solvent< D > >.

Definition at line 48 of file Mixture.tpp.

References UTIL_CHECK.

12.30.4.2 setMesh() `template<int D>`

```
void Pscf::Pspg::Continuous::Mixture< D >::setMesh (
            Mesh< D > const & mesh,
```

```
            UnitCell< D > & unitCell )
```

Create an association with the mesh and allocate memory.

The Mesh<D> object must have already been initialized, e.g., by reading its parameters from a file, so that the mesh dimensions are known on entry.

Parameters

mesh	associated Mesh<D> object (stores address).

Definition at line 81 of file Mixture.tpp.

References Pscf::Mesh< D >::dimensions(), Pscf::UnitCellBase< D >::nParameter(), Pscf::Pspg::ThreadGrid::set←ThreadsLogical(), and UTIL_CHECK.

12.30.4.3 setupUnitCell() `template<int D>`

```
void Pscf::Pspg::Continuous::Mixture< D >::setupUnitCell (
            const UnitCell< D > & unitCell,
            const WaveList< D > & wavelist )
```

Set unit cell parameters used in solver.

Parameters

unitCell	UnitCell<D> object that contains Bravais lattice.

Definition at line 238 of file Mixture.tpp.

References Pscf::MeshIterator< D >::atEnd(), Pscf::MeshIterator< D >::begin(), Pscf::UnitCellBase< D >::ksq(), Pscf::UnitCellBase< D >::nParameter(), Pscf::MeshIterator< D >::position(), Pscf::MeshIterator< D >::rank(), and Pscf::MeshIterator< D >::setDimensions().

12.30.4.4 compute() `template<int D>`

```
void Pscf::Pspg::Continuous::Mixture< D >::compute (
            DArray< WField > const & wFields,
            DArray< CField > & cFields )
```

Compute concentrations.

This function calls the compute function of every molecular species, and then adds the resulting block concentration fields for blocks of each type to compute a total monomer concentration (or volume fraction) for each monomer type. Upon return, values are set for volume fraction and chemical potential (mu) members of each species, and for the concentration fields for each Block and Solvent. The total concentration for each monomer type is returned in the cFields output parameter.

The arrays wFields and cFields must each have size nMonomer(), and contain fields that are indexed by monomer type index.

Parameters

wFields	array of chemical potential fields (input)
cFields	array of monomer concentration fields (output)

Definition at line 335 of file Mixture.tpp.

References Pscf::Pspg::DField< Data >::cDField(), Pscf::Pspg::ThreadGrid::setThreadsLogical(), and UTIL_CHECK.

12.30.4.5 computeStress() `template<int D>`

```
void Pscf::Pspg::Continuous::Mixture< D >::computeStress (
              WaveList< D > & wavelist )
```

Get monomer reference volume.

Definition at line 408 of file Mixture.tpp.

References Pscf::Pspg::ThreadGrid::setThreadsLogical().

12.30.4.6 stress() `template<int D>`

```
double Pscf::Pspg::Continuous::Mixture< D >::stress (
              int n )  [inline]
```

Get derivative of free energy w/ respect to cell parameter.

Get precomputed value of derivative of free energy per monomer with respect to unit cell parameter number n.

\int n unit cell parameter id

Definition at line 152 of file pgc/solvers/Mixture.h.

12.30.4.7 vMonomer() `template<int D>`

```
double Pscf::Pspg::Continuous::Mixture< D >::vMonomer  [inline]
```

Get monomer reference volume.

Definition at line 242 of file pgc/solvers/Mixture.h.

12.30.4.8 read() `template<int D>`

```
template<typename Type >
ScalarParam< Type > & Util::ParamComposite::read (
              typename Type  )  [protected]
```

Add and read a new required ScalarParam < Type > object.

This is equivalent to ScalarParam<Type>(in, label, value, true).

Parameters

in	input stream for reading
label	Label string
value	reference to new ScalarParam< Type >

Definition at line 1156 of file ParamComposite.h.

12.30.4.9 readOptional() `template<int D>`

```
template<typename Type >
ScalarParam< Type > & Util::ParamComposite::readOptional (
              typename Type  )  [inline], [protected]
```

Add and read a new optional ScalarParam < Type > object.

This is equivalent to ScalarParam<Type>(in, label, value, false).

Parameters

in	input stream for reading
label	Label string
value	reference to new ScalarParam< Type >

Definition at line 1164 of file ParamComposite.h.
The documentation for this class was generated from the following files:

- pgc/solvers/Mixture.h
- Mixture.tpp

12.31 Pscf::Pspg::Continuous::Polymer< D > Class Template Reference

Descriptor and solver for a branched polymer species.

```
#include <Polymer.h>
```

Inheritance diagram for Pscf::Pspg::Continuous::Polymer< D >:

classPscf_1_1Pspg_1_1Continuous_1_1Polymer-eps-converted-to.p

Public Member Functions

- void compute (DArray< WField > const &wFields)

  Compute solution to MDE and concentrations.
- void computeJoint (const Mesh< D > &mesh)

  Compute Joint density fields.
- void computeStress (WaveList< D > &wavelist)

  Compute stress from a polymer chain, needs a pointer to basis.
- double stress (int n)

  Get derivative of free energy w/ respect to a unit cell parameter.

Protected Member Functions

- void setClassName (const char ∗className)

  Set class name string.

Additional Inherited Members

12.31.1 Detailed Description

template<int D>
class Pscf::Pspg::Continuous::Polymer< D >

Descriptor and solver for a branched polymer species.
The block concentrations stored in the constituent Block<D> objects contain the block concentrations (i.e., volume fractions) computed in the most recent call of the compute function.
The phi() and mu() accessor functions, which are inherited from PolymerTmp< Block<D> >, return the value of phi (spatial average volume fraction of a species) or mu (chemical potential) computed in the last call of the compute function. If the ensemble for this species is closed, phi is read from the parameter file and mu is computed. If the ensemble is open, mu is read from the parameter file and phi is computed.
Definition at line 44 of file Polymer.h.

12.31.2 Member Function Documentation

12.31.2.1 compute() `template<int D>`

```
void Pscf::Pspg::Continuous::Polymer< D >::compute (
             DArray< WField > const & wFields )
```

Compute solution to MDE and concentrations.

Definition at line 63 of file Polymer.tpp.

12.31.2.2 computeJoint() `template<int D>`

```
void Pscf::Pspg::Continuous::Polymer< D >::computeJoint (
             const Mesh< D > & mesh )
```

Compute Joint density fields.

Definition at line 76 of file Polymer.tpp.

References Pscf::Mesh< D >::dimensions(), Pscf::Pspg::ThreadGrid::setThreadsLogical(), and Pscf::Mesh< D >↩
::size().

12.31.2.3 computeStress() `template<int D>`

```
void Pscf::Pspg::Continuous::Polymer< D >::computeStress (
             WaveList< D > & wavelist )
```

Compute stress from a polymer chain, needs a pointer to basis.

Definition at line 134 of file Polymer.tpp.

12.31.2.4 stress() `template<int D>`

```
double Pscf::Pspg::Continuous::Polymer< D >::stress (
             int n )  [inline]
```

Get derivative of free energy w/ respect to a unit cell parameter.

Get the contribution from this polymer species to the derivative of free energy per monomer with respect to unit cell parameter n.

Parameters

n	unit cell parameter index

Definition at line 122 of file Polymer.h.

12.31.2.5 setClassName() `template<int D>`

```
void Util::ParamComposite::setClassName  [protected]
```

Set class name string.

Should be set in subclass constructor.

Definition at line 377 of file ParamComposite.cpp.

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

- Polymer.h
- Polymer.tpp

12.32 Pscf::Pspg::Continuous::Propagator< D > Class Template Reference

MDE solver for one-direction of one block.

```
#include <Propagator.h>
```

Inheritance diagram for Pscf::Pspg::Continuous::Propagator< D >:

classPscf_1_1Pspg_1_1Continuous_1_1Propagator-eps-conver

Public Types

- typedef RDField< D > Field

  Generic field (function of position).
- typedef RDField< D > WField

  Chemical potential field type.
- typedef RDField< D > CField

  Monomer concentration field type.
- typedef RDField< D > QField

  Propagator q-field type.

Public Member Functions

- Propagator ()

  Constructor.
- ∼Propagator ()

  Destructor.
- void setBlock (Block< D > &block)

  Associate this propagator with a block.
- void allocate (int ns, const Mesh< D > &mesh)

  Associate this propagator with a block.
- void solveForward ()

  Compute the forward propagators and store the "slices" for the corresponding block.
- void solveBackward (cudaReal ∗q, int n)

  Compute the backward propagators with the stored "slices" as well as the integrals and normalized single chain partition function that are needed for concentration fields and for the corresponding block.
- double intQ (cudaReal ∗q, cudaReal ∗qs)

  Compute and return partition function for the molecule.
- const cudaReal ∗ q (int i) const

  Return q-field at specified step.
- cudaReal ∗ head () const

  Return q-field at beginning of block (initial condition).
- const cudaReal ∗ qtail () const

  Return q-field at end of block.
- Block< D > & block ()

  Get the associated Block object by reference.
- bool isAllocated () const

  Has memory been allocated for this propagator?

Protected Member Functions

- void computeHead ()

  Compute initial QField at head from tail QFields of sources.

12.32.1 Detailed Description

template<int D>
class Pscf::Pspg::Continuous::Propagator< D >

MDE solver for one-direction of one block.
Definition at line 38 of file Propagator.h.

12.32.2 Member Typedef Documentation

12.32.2.1 Field `template<int D>`

`typedef RDField<D> Pscf::Pspg::Continuous::Propagator< D >::Field`
Generic field (function of position).
Definition at line 48 of file Propagator.h.

12.32.2.2 WField `template<int D>`

`typedef RDField<D> Pscf::Pspg::Continuous::Propagator< D >::WField`
Chemical potential field type.
Definition at line 54 of file Propagator.h.

12.32.2.3 CField `template<int D>`

`typedef RDField<D> Pscf::Pspg::Continuous::Propagator< D >::CField`
Monomer concentration field type.
Definition at line 59 of file Propagator.h.

12.32.2.4 QField `template<int D>`

`typedef RDField<D> Pscf::Pspg::Continuous::Propagator< D >::QField`
Propagator q-field type.
Definition at line 64 of file Propagator.h.

12.32.3 Constructor & Destructor Documentation

12.32.3.1 Propagator() `template<int D>`

`Pscf::Pspg::Continuous::Propagator< D >::Propagator`
Constructor.
Definition at line 35 of file Propagator.tpp.

12.32.3.2 ∼Propagator() `template<int D>`

`Pscf::Pspg::Continuous::Propagator< D >::∼Propagator`
Destructor.
Definition at line 47 of file Propagator.tpp.

12.32.4 Member Function Documentation

12.32.4.1 setBlock() `template<int D>`

```
void Pscf::Pspg::Continuous::Propagator< D >::setBlock (
            Block< D > & block )  [inline]
```

Associate this propagator with a block.

Parameters

block	associated Block object.

Definition at line 269 of file Propagator.h.

12.32.4.2 allocate() `template<int D>`

```
void Pscf::Pspg::Continuous::Propagator< D >::allocate (
            int ns,
            const Mesh< D > & mesh )
```

Associate this propagator with a block.

Parameters

ns	number of contour length steps
mesh	spatial discretization mesh

Definition at line 56 of file Propagator.tpp.
References Pscf::Mesh< D >::dimensions(), Pscf::Pspg::ThreadGrid::nBlocks(), Pscf::Pspg::ThreadGrid::set↩ThreadsLogical(), and Pscf::Mesh< D >::size().

12.32.4.3 solveForward() `template<int D>`

```
void Pscf::Pspg::Continuous::Propagator< D >::solveForward
```

Compute the forward propagators and store the "slices" for the corresponding block.
Definition at line 120 of file Propagator.tpp.
References UTIL_CHECK.

12.32.4.4 solveBackward() `template<int D>`

```
void Pscf::Pspg::Continuous::Propagator< D >::solveBackward (
            cudaReal * q,
            int n )
```

Compute the backward propagators with the stored "slices" as well as the integrals and normalized single chain partition function that are needed for concentration fields and for the corresponding block.

Definition at line 222 of file Propagator.tpp.
References Pscf::Pspg::ThreadGrid::setThreadsLogical(), and UTIL_CHECK.

12.32.4.5 intQ() `template<int D>`

```
double Pscf::Pspg::Continuous::Propagator< D >::intQ (
```

```
cudaReal * q,
cudaReal * qs )
```

Compute and return partition function for the molecule.

This function is called by void solveBackward(cudaReal *q, int n)

Definition at line 307 of file Propagator.tpp.

References Pscf::Pspg::ThreadGrid::setThreadsLogical().

12.32.4.6 q() template<int D>

```
const cudaReal * Pscf::Pspg::Continuous::Propagator< D >::q (
              int i ) const  [inline]
```

Return q-field at specified step.

Parameters

i	step index

Definition at line 238 of file Propagator.h.

12.32.4.7 head() template<int D>

```
cudaReal * Pscf::Pspg::Continuous::Propagator< D >::head  [inline]
```

Return q-field at beginning of block (initial condition).

Definition at line 205 of file Propagator.h.

12.32.4.8 qtail() template<int D>

```
const cudaReal * Pscf::Pspg::Continuous::Propagator< D >::qtail  [inline]
```

Return q-field at end of block.

Definition at line 229 of file Propagator.h.

12.32.4.9 block() template<int D>

```
Block< D > & Pscf::Pspg::Continuous::Propagator< D >::block  [inline]
```

Get the associated Block object by reference.

Definition at line 253 of file Propagator.h.

12.32.4.10 isAllocated() template<int D>

```
bool Pscf::Pspg::Continuous::Propagator< D >::isAllocated  [inline]
```

Has memory been allocated for this propagator?

Definition at line 260 of file Propagator.h.

12.32.4.11 computeHead() template<int D>

```
void Pscf::Pspg::Continuous::Propagator< D >::computeHead  [protected]
```

Compute initial QField at head from tail QFields of sources.

Definition at line 83 of file Propagator.tpp.

References Pscf::Pspg::ThreadGrid::setThreadsLogical(), and UTIL_THROW.

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

- Propagator.h
- Propagator.tpp

12.33 Pscf::Pspg::Continuous::System< D > Class Template Reference

Main class in SCFT simulation of one system.

`#include <System.h>`

Public Types

- typedef RDField< D > Field

    Base class for WField and CField.
- typedef Propagator< D >::WField WField

    Monomer chemical potential field type.
- typedef Propagator< D >::CField CField

    Monomer concentration / volume fraction field type.

Public Member Functions

- System ()

    Constructor.
- ∼System ()

    Destructor.

Lifetime (Actions)

- void setOptions (int argc, char ∗∗argv)

    Process command line options.
- void **setOptionsOutside** (char ∗pArg, char ∗cArg)
- virtual void readParam (std::istream &in)

    Read input parameters (with opening and closing lines).
- void readParam ()

    Read input parameters from default param file.
- virtual void readParameters (std::istream &in)

    Read body of input parameters block (without opening and closing lines).
- void readCommands (std::istream &in)

    Read command script.
- void readCommands ()

    Read commands from default command file.
- void computeFreeEnergy ()

    Compute free energy density and pressure for current fields.
- void outputThermo (std::ostream &out)

    Output the thermodynamic properties to a file.

Chemical Potential Fields (W Fields)

- DArray< Field > & wFields ()

    Get an array of chemical potential fields, in a basis.
- Field & wField (int monomerId)

    Get chemical potential field for one monomer type, in a basis.
- DArray< WField > & wFieldsRGrid ()

    Get array of chemical potential fields, on an r-space grid.
- DArray< RDField< D > > & **wFieldsRGridPh** ()
- WField & wFieldRGrid (int monomerId)

    Get the chemical potential field for one monomer type, on a grid.
- RDField< D > & **wFieldRGridPh** (int monomerId)
- DArray< RDFieldDft< D > > & wFieldsKGrid ()

    Get array of chemical potential fields, in Fourier space.

- RDFieldDft< D > & wFieldKGrid (int monomerId)
  Get the chemical potential field for one monomer, in Fourier space.

Monomer Concentration / Volume Fraction Fields (C Fields)

- DArray< Field > & cFields ()
  Get an array of all monomer concentration fields, in a basis.
- Field & cField (int monomerId)
  Get the concentration field for one monomer type, in a basis.
- DArray< CField > & cFieldsRGrid ()
  Get array of all concentration fields (c fields), on a grid.
- DArray< CField > & **cFieldsRGridPh** ()
- CField & cFieldRGrid (int monomerId)
  Get the concentration (c field) for one monomer type, on a grid.
- CField & **cFieldRGridPh** (int monomerId)
- DArray< RDFieldDft< D > > & cFieldsKGrid ()
  Get all monomer concentration fields, in Fourier space (k-grid).
- RDFieldDft< D > & cFieldKGrid (int monomerId)
  Get the c field for one monomer type, in Fourier space (k-grid).
- double c (int monomerId)
  return homogeneous volume fraction for one monnomer type.

Accessors (access objects by reference)

- Mixture< D > & mixture ()
  Get Mixture by reference.
- Mesh< D > & mesh ()
  Get spatial discretization mesh by reference.
- UnitCell< D > & unitCell ()
  Get crystal unitCell (i.e., lattice type and parameters) by reference.
- ChiInteraction & interaction ()
  Get interaction (i.e., excess free energy model) by reference.
- AmIterator< D > & iterator ()
  Get the Iterator by reference.
- Basis< D > & basis ()
  Get basis object by reference.
- WaveList< D > & wavelist ()
  Get container for wavevector data.
- FieldIo< D > & fieldIo ()
  Get associated FieldIo object.
- FFT< D > & fft ()
  Get associated FFT object by reference.
- FCT< D > & **fct** ()
- Homogeneous::Mixture & homogeneous ()
  Get homogeneous mixture (for reference calculations).
- FileMaster & fileMaster ()
  Get FileMaster by reference.

Accessors (return values)

- std::string groupName ()
  Get the group name string.
- double fHelmholtz () const
  Get precomputed Helmholtz free energy per monomer / kT.
- double pressure () const

Get precomputed pressure x monomer volume kT.

- bool hasWFields () const

  Have monomer chemical potential fields (w fields) been set?

- bool hasCFields () const

  Have monomer concentration fields (c fields) been computed?

12.33.1 Detailed Description

template<int D>
class Pscf::Pspg::Continuous::System< D >

Main class in SCFT simulation of one system.
Definition at line 22 of file pgc/iterator/Iterator.h.

12.33.2 Member Typedef Documentation

12.33.2.1 Field template<int D>

`typedef RDField<D> Pscf::Pspg::Continuous::System< D >::Field`
Base class for WField and CField.
Definition at line 50 of file pgc/System.h.

12.33.2.2 WField template<int D>

`typedef Propagator<D>::WField Pscf::Pspg::Continuous::System< D >::WField`
Monomer chemical potential field type.
Definition at line 53 of file pgc/System.h.

12.33.2.3 CField template<int D>

`typedef Propagator<D>::CField Pscf::Pspg::Continuous::System< D >::CField`
Monomer concentration / volume fraction field type.
Definition at line 56 of file pgc/System.h.

12.33.3 Constructor & Destructor Documentation

12.33.3.1 System() template<int D>

`Pscf::Pspg::Continuous::System< D >::System`
Constructor.
Definition at line 41 of file pgc/System.tpp.
References Pscf::Pspg::ThreadGrid::init().

12.33.3.2 ∼System() template<int D>

`Pscf::Pspg::Continuous::System< D >::∼System`
Destructor.
Definition at line 74 of file pgc/System.tpp.

12.33.4 Member Function Documentation

12.33.4.1 setOptions() `template<int D>`

```
void Pscf::Pspg::Continuous::System< D >::setOptions (
            int argc,
            char ** argv )
```

Process command line options.

Definition at line 86 of file pgc/System.tpp.

References Util::Log::file(), Util::ParamComponent::setEcho(), Pscf::Pspg::ThreadGrid::setThreadsPerBlock(), and UTIL_THROW.

12.33.4.2 readParam() [1/2] `template<int D>`

```
void Pscf::Pspg::Continuous::System< D >::readParam (
            std::istream & in )  [virtual]
```

Read input parameters (with opening and closing lines).

Parameters

in	input parameter stream

Definition at line 202 of file pgc/System.tpp.

12.33.4.3 readParam() [2/2] `template<int D>`

```
void Pscf::Pspg::Continuous::System< D >::readParam
```

Read input parameters from default param file.

Definition at line 190 of file pgc/System.tpp.

12.33.4.4 readParameters() `template<int D>`

```
void Pscf::Pspg::Continuous::System< D >::readParameters (
            std::istream & in )  [virtual]
```

Read body of input parameters block (without opening and closing lines).

Parameters

in	input parameter stream

Read crystallographic unit cell (used only to create basis)

Definition at line 213 of file pgc/System.tpp.

12.33.4.5 readCommands() [1/2] `template<int D>`

```
void Pscf::Pspg::Continuous::System< D >::readCommands (
            std::istream & in )
```

Read command script.

Parameters

in	command script file.

Definition at line 336 of file pgc/System.tpp.

References Util::Log::file(), and UTIL_CHECK.

12.33.4.6 readCommands() [2/2] template<int D>

void Pscf::Pspg::Continuous::System< D >::readCommands

Read commands from default command file.

Definition at line 321 of file pgc/System.tpp.

References UTIL_THROW.

12.33.4.7 computeFreeEnergy() template<int D>

void Pscf::Pspg::Continuous::System< D >::computeFreeEnergy

Compute free energy density and pressure for current fields.

This function should be called after a successful call of iterator().solve(). Resulting values are returned by the free←Energy() and pressure() accessor functions.

Definition at line 886 of file pgc/System.tpp.

References Pscf::PolymerTmpl< Block< D > >::length(), Pscf::Species::mu(), Pscf::Pspg::ThreadGrid::nBlocks(), Pscf::Pspg::ThreadGrid::nThreads(), Pscf::Species::phi(), and Pscf::Pspg::ThreadGrid::setThreadsLogical().

12.33.4.8 outputThermo() template<int D>

void Pscf::Pspg::Continuous::System< D >::outputThermo (
std::ostream & *out*)

Output the thermodynamic properties to a file.

This function outputs Helmholtz free energy per monomer, pressure (in units of kT per monomer volume), and the volume fraction and chemical potential of each species.

/param out

Definition at line 1011 of file pgc/System.tpp.

12.33.4.9 wFields() template<int D>

DArray< RDField< D > > & Pscf::Pspg::Continuous::System< D >::wFields [inline]

Get an array of chemical potential fields, in a basis.

This function returns an array in which each element is an array containing the coefficients of the chemical potential field (w field) in a symmetry-adapted basis for one monomer type. The array capacity is the number of monomer types.

Definition at line 643 of file pgc/System.h.

12.33.4.10 wField() template<int D>

RDField< D > & Pscf::Pspg::Continuous::System< D >::wField (
int *monomerId*) [inline]

Get chemical potential field for one monomer type, in a basis.

This function returns an array containing coefficients of the chemical potential field (w field) in a symmetry-adapted basis for a specified monomer type.

Parameters

monomer←Id	integer monomer type index

Definition at line 650 of file pgc/System.h.

12.33.4.11 wFieldsRGrid() template<int D>

DArray< typename System< D >::WField > & Pscf::Pspg::Continuous::System< D >::wFieldsRGrid [inline]

Get array of chemical potential fields, on an r-space grid.

This function returns an array in which each element is a WField object containing values of the chemical potential field (w field) on a regular grid for one monomer type. The array capacity is the number of monomer types.

Definition at line 657 of file pgc/System.h.

12.33.4.12 wFieldRGrid() template<int D>

System< D >::WField & Pscf::Pspg::Continuous::System< D >::wFieldRGrid (int *monomerId*) [inline]

Get the chemical potential field for one monomer type, on a grid.

Parameters

monomerId	integer monomer type index

Definition at line 672 of file pgc/System.h.

12.33.4.13 wFieldsKGrid() template<int D>

DArray< RDFieldDft< D > > & Pscf::Pspg::Continuous::System< D >::wFieldsKGrid [inline]

Get array of chemical potential fields, in Fourier space.

The array capacity is equal to the number of monomer types.

Definition at line 684 of file pgc/System.h.

12.33.4.14 wFieldKGrid() template<int D>

RDFieldDft< D > & Pscf::Pspg::Continuous::System< D >::wFieldKGrid (int *monomerId*) [inline]

Get the chemical potential field for one monomer, in Fourier space.

Parameters

monomerId	integer monomer type index

Definition at line 690 of file pgc/System.h.

12.33.4.15 cFields() template<int D>

DArray< RDField< D > > & Pscf::Pspg::Continuous::System< D >::cFields [inline]

Get an array of all monomer concentration fields, in a basis.

This function returns an array in which each element is an array containing the coefficients of the monomer concentration field (cfield) for one monomer type in a symmetry-adapted basis. The array capacity is equal to the number of monomer types.

Definition at line 697 of file pgc/System.h.

12.33.4.16 cField() template<int D>

RDField< D > & Pscf::Pspg::Continuous::System< D >::cField (int *monomerId*) [inline]

Get the concentration field for one monomer type, in a basis.

This function returns an array containing the coefficients of the monomer concentration / volume fraction field (c field) for a specific monomer type.

Parameters

monomerId	integer monomer type index

Definition at line 704 of file pgc/System.h.

12.33.4.17 cFieldsRGrid() `template<int D>`

`DArray< typename System< D >::CField > & Pscf::Pspg::Continuous::System< D >::cFieldsRGrid [inline]`

Get array of all concentration fields (c fields), on a grid.

This function returns an array in which each element is the monomer concentration field for one monomer type on a regular grid (an r-grid).

Definition at line 711 of file pgc/System.h.

12.33.4.18 cFieldRGrid() `template<int D>`

`System< D >::CField & Pscf::Pspg::Continuous::System< D >::cFieldRGrid ( int monomerId ) [inline]`

Get the concentration (c field) for one monomer type, on a grid.

Parameters

monomerId	integer monomer type index

Definition at line 720 of file pgc/System.h.

12.33.4.19 cFieldsKGrid() `template<int D>`

`DArray< RDFieldDft< D > > & Pscf::Pspg::Continuous::System< D >::cFieldsKGrid [inline]`

Get all monomer concentration fields, in Fourier space (k-grid).

This function returns an array in which each element is the discrete Fourier transform (DFT) of the concentration field (c field) for on monomer type.

Definition at line 726 of file pgc/System.h.

12.33.4.20 cFieldKGrid() `template<int D>`

`RDFieldDft< D > & Pscf::Pspg::Continuous::System< D >::cFieldKGrid ( int monomerId ) [inline]`

Get the c field for one monomer type, in Fourier space (k-grid).

This function returns the discrete Fourier transform (DFT) of the concentration field (c field) for monomer type index monomerId.

Parameters

monomerId	integer monomer type index

Definition at line 732 of file pgc/System.h.

12.33.4.21 c() template<int D>

`double Pscf::Pspg::Continuous::System< D >::c ( int monomerId ) [inline]`

return homogeneous volume fraction for one monnomer type.
Definition at line 738 of file pgc/System.h.

12.33.4.22 mixture() template<int D>

`Mixture< D > & Pscf::Pspg::Continuous::System< D >::mixture [inline]`

Get Mixture by reference.
Definition at line 557 of file pgc/System.h.

12.33.4.23 mesh() template<int D>

`Mesh< D > & Pscf::Pspg::Continuous::System< D >::mesh [inline]`

Get spatial discretization mesh by reference.
Definition at line 564 of file pgc/System.h.

12.33.4.24 unitCell() template<int D>

`UnitCell< D > & Pscf::Pspg::Continuous::System< D >::unitCell [inline]`

Get crystal unitCell (i.e., lattice type and parameters) by reference.
Definition at line 571 of file pgc/System.h.

12.33.4.25 interaction() template<int D>

`ChiInteraction & Pscf::Pspg::Continuous::System< D >::interaction [inline]`

Get interaction (i.e., excess free energy model) by reference.
Definition at line 578 of file pgc/System.h.
References UTIL_ASSERT.

12.33.4.26 iterator() template<int D>

`AmIterator< D > & Pscf::Pspg::Continuous::System< D >::iterator [inline]`

Get the Iterator by reference.
Definition at line 621 of file pgc/System.h.
References UTIL_ASSERT.

12.33.4.27 basis() template<int D>

`Basis< D > & Pscf::Pspg::Continuous::System< D >::basis [inline]`

Get basis object by reference.
Definition at line 599 of file pgc/System.h.
References UTIL_ASSERT.

12.33.4.28 wavelist() template<int D>

`WaveList< D > & Pscf::Pspg::Continuous::System< D >::wavelist [inline]`

Get container for wavevector data.

Definition at line 607 of file pgc/System.h.

12.33.4.29 fieldIo() template<int D>

FieldIo< D > & Pscf::Pspg::Continuous::System< D >::fieldIo [inline]

Get associated FieldIo object.

Definition at line 614 of file pgc/System.h.

12.33.4.30 fft() template<int D>

FFT< D > & Pscf::Pspg::Continuous::System< D >::fft [inline]

Get associated FFT object by reference.

Definition at line 586 of file pgc/System.h.

12.33.4.31 homogeneous() template<int D>

Homogeneous::Mixture & Pscf::Pspg::Continuous::System< D >::homogeneous [inline]

Get homogeneous mixture (for reference calculations).

Definition at line 629 of file pgc/System.h.

12.33.4.32 fileMaster() template<int D>

FileMaster & Pscf::Pspg::Continuous::System< D >::fileMaster [inline]

Get FileMaster by reference.

Definition at line 636 of file pgc/System.h.

12.33.4.33 groupName() template<int D>

std::string Pscf::Pspg::Continuous::System< D >::groupName [inline]

Get the group name string.

Definition at line 745 of file pgc/System.h.

12.33.4.34 fHelmholtz() template<int D>

double Pscf::Pspg::Continuous::System< D >::fHelmholtz [inline]

Get precomputed Helmholtz free energy per monomer / kT.

The value retrieved by this function is computed by the computeFreeEnergy() function.

Definition at line 752 of file pgc/System.h.

12.33.4.35 pressure() template<int D>

double Pscf::Pspg::Continuous::System< D >::pressure [inline]

Get precomputed pressure x monomer volume kT.

The value retrieved by this function is computed by the computeFreeEnergy() function.

Definition at line 759 of file pgc/System.h.

12.33.4.36 hasWFields() template<int D>

bool Pscf::Pspg::Continuous::System< D >::hasWFields [inline]

Have monomer chemical potential fields (w fields) been set?

A true value is returned if and only if values have been set on a real space grid. The READ_W_BASIS command must immediately convert from a basis to a grid to satisfy this requirement.

Definition at line 766 of file pgc/System.h.

12.33.4.37 hasCFields() `template<int D>`

`bool Pscf::Pspg::Continuous::System< D >::hasCFields [inline]`

Have monomer concentration fields (c fields) been computed?

A true value is returned if and only if monomer concentration fields have been computed by solving the modified diffusion equation for the current w fields, and values are known on a grid (cFieldsRGrid).

Definition at line 773 of file pgc/System.h.

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

- pgc/iterator/Iterator.h
- pgc/System.h
- pgc/System.tpp

12.34 Pscf::Pspg::DField< Data > Class Template Reference

Dynamic array with aligned data, for use with cufftw library/device code.

`#include <DField.h>`

Public Member Functions

- DField ()

  Default constructor.
- virtual ~DField ()

  Destructor.
- void allocate (int capacity)

  Allocate the underlying C array.
- void deallocate ()

  Dellocate the underlying C array.
- bool isAllocated () const

  Return true if the Field has been allocated, false otherwise.
- int capacity () const

  Return allocated size.
- Data * cDField ()

  Return pointer to underlying C array.
- const Data * cDField () const

  Return pointer to const to underlying C array.

Protected Attributes

- Data * data_

  Serialize a Field to/from an Archive.
- int capacity_

  Allocated size of the data_ array.

12.34.1 Detailed Description

template<typename Data>
class Pscf::Pspg::DField< Data >

Dynamic array with aligned data, for use with cufftw library/device code.

This class does not offer memory access via operator[]

Definition at line 25 of file DField.h.

12.34.2 Constructor & Destructor Documentation

12.34.2.1 DField() `template<typename Data >`

`Pscf::Pspg::DField< Data >::DField`

Default constructor.

Definition at line 25 of file DField.tpp.

12.34.2.2 ∼DField() `template<typename Data >`

`Pscf::Pspg::DField< Data >::~DField [virtual]`

Destructor.

Deletes underlying C array, if allocated previously.

Definition at line 34 of file DField.tpp.

12.34.3 Member Function Documentation

12.34.3.1 allocate() `template<typename Data >`

```
void Pscf::Pspg::DField< Data >::allocate (
            int capacity )
```

Allocate the underlying C array.

Exceptions

Exception	if the Field is already allocated.

Parameters

capacity	number of elements to allocate.

Definition at line 50 of file DField.tpp.

Referenced by Pscf::Pspg::RDFieldDft< D >::allocate(), and Pscf::Pspg::RDField< D >::allocate().

12.34.3.2 deallocate() `template<typename Data >`

`void Pscf::Pspg::DField< Data >::deallocate`

Dellocate the underlying C array.

Exceptions

Exception	if the Field is not allocated.

Definition at line 68 of file DField.tpp.

12.34.3.3 isAllocated() `template<typename Data >`

`bool Pscf::Pspg::DField< Data >::isAllocated [inline]`

Return true if the Field has been allocated, false otherwise.

Definition at line 133 of file DField.h.

12.34.3.4 capacity() `template<typename Data >`

`int Pscf::Pspg::DField< Data >::capacity [inline]`

Return allocated size.

Returns

Number of elements allocated in array.

Definition at line 112 of file DField.h.

12.34.3.5 cDField() [1/2] `template<typename Data >`

`const Data * Pscf::Pspg::DField< Data >::cDField [inline]`

Return pointer to underlying C array.

Definition at line 119 of file DField.h.

Referenced by Pscf::Pspg::Continuous::Mixture< D >::compute(), and Pscf::Pspg::Continuous::Block< D >::setup↩ Solver().

12.34.3.6 cDField() [2/2] `template<typename Data >`

`const Data* Pscf::Pspg::DField< Data >::cDField ( ) const`

Return pointer to const to underlying C array.

12.34.4 Member Data Documentation

12.34.4.1 data_ `template<typename Data >`

`Data* Pscf::Pspg::DField< Data >::data_ [protected]`

Serialize a Field to/from an Archive.

Parameters

ar	archive
version	archive version id Pointer to an array of Data elements.

Definition at line 100 of file DField.h.

12.34.4.2 capacity_ `template<typename Data >`

`int Pscf::Pspg::DField< Data >::capacity_ [protected]`

Allocated size of the data_ array.

Definition at line 103 of file DField.h.

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

- DField.h
- DField.tpp

12.35 Pscf::Pspg::Discrete::AmIterator< D > Class Template Reference

Anderson mixing iterator for the pseudo spectral method.

```
#include <AmIterator.h>
```

Inheritance diagram for Pscf::Pspg::Discrete::AmIterator< D >:

classPscf_1_1Pspg_1_1Discrete_1_1AmIterator-eps-converted-to.pc

Public Member Functions

- AmIterator ()

  Default constructor.
- ~AmIterator ()

  Destructor.
- void readParameters (std::istream &in)

  Read all parameters and initialize.

Additional Inherited Members

12.35.1 Detailed Description

template<int D>
class Pscf::Pspg::Discrete::AmIterator< D >

Anderson mixing iterator for the pseudo spectral method.
Definition at line 28 of file pgd/iterator/AmIterator.h.

12.35.2 Constructor & Destructor Documentation

12.35.2.1 AmIterator() `template<int D>`

`Pscf::Pspg::Discrete::AmIterator< D >::AmIterator`

Default constructor.
Definition at line 21 of file pgd/iterator/AmIterator.tpp.
References Util::ParamComposite::setClassName().

12.35.2.2 ~AmIterator() `template<int D>`

`Pscf::Pspg::Discrete::AmIterator< D >::~AmIterator`

Destructor.
Definition at line 45 of file pgd/iterator/AmIterator.tpp.

12.35.3 Member Function Documentation

12.35.3.1 readParameters() `template<int D>`

`void Pscf::Pspg::Discrete::AmIterator< D >::readParameters ( std::istream & in ) [virtual]`

Read all parameters and initialize.

Parameters

in	input filestream

Reimplemented from Util::ParamComposite.

Definition at line 50 of file pgd/iterator/AmIterator.tpp.

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

- pgd/iterator/AmIterator.h
- pgd/iterator/AmIterator.tpp

12.36 Pscf::Pspg::Discrete::Bond< D > Class Template Reference

Bond within a branched polymer.

`#include <Bond.h>`

Inheritance diagram for Pscf::Pspg::Discrete::Bond< D >:

classPscf_1_1Pspg_1_1Discrete_1_1Bond-eps-converted-to.pdf

Public Member Functions

- bool bondtype () const

  Get the type of this bond.
- int length () const

  Get the number of segments in this bond.

12.36.1 Detailed Description

template<int D>
class Pscf::Pspg::Discrete::Bond< D >

Bond within a branched polymer.

(discrete chain model)

Derived from BondTmpl< DPropagator<D> >. A BondTmpl< DPropagator<D> > has two SPropagator<D> members and is derived from BondDescriptor.

Definition at line 37 of file Bond.h.

12.36.2 Member Function Documentation

12.36.2.1 bondtype() `template<int D>`

`bool Pscf::BondDescriptor::bondtype [inline]`

Get the type of this bond.
0 for block-bond; 1 for joint-bond
Definition at line 189 of file BondDescriptor.h.

12.36.2.2 length() `template<int D>`

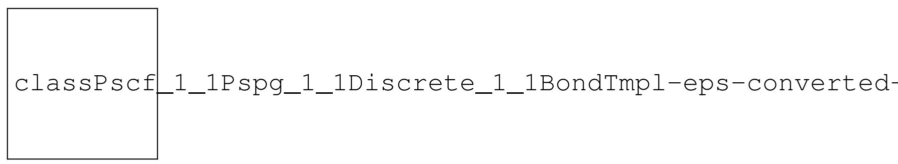
`int Pscf::BondDescriptor::length [inline]`

Get the number of segments in this bond.
For a block bond, the return value should be a positive integer number.
Definition at line 181 of file BondDescriptor.h.
The documentation for this class was generated from the following files:

- Bond.h
- Bond.tpp

12.37 Pscf::Pspg::Discrete::BondTmpl< TP > Class Template Reference

Inheritance diagram for Pscf::Pspg::Discrete::BondTmpl< TP >:



Public Member Functions

- virtual void setKuhn (double kuhn1, double kuhn2)

  non-inline
- TP & propagator (int directionId)

  inline

12.37.1 Detailed Description

template<class TP>
class Pscf::Pspg::Discrete::BondTmpl< TP >

Definition at line 18 of file BondTmpl.h.

12.37.2 Member Function Documentation

12.37.2.1 setKuhn() `template<class TP >`

```
void Pscf::Pspg::Discrete::BondTmpl< TP >::setKuhn (
            double kuhn1,
            double kuhn2 ) [virtual]
```

non-inline
Definition at line 68 of file BondTmpl.h.

12.37.2.2 propagator() `template<class TP >`

`TP & Pscf::Pspg::Discrete::BondTmpl< TP >::propagator ( int directionId ) [inline]`

inline

Definition at line 75 of file BondTmpl.h.

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

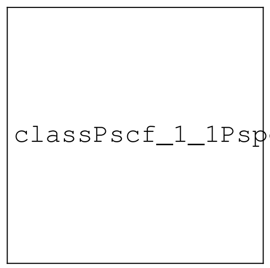
- BondTmpl.h

12.38 Pscf::Pspg::Discrete::DMixture< D > Class Template Reference

Solver for a mixture of polymers (Discrete chain model).

`#include <DMixture.h>`

Inheritance diagram for Pscf::Pspg::Discrete::DMixture< D >:



Public Member Functions

- void readParameters (std::istream &in)

  Read parameters from file and initialize.

Protected Member Functions

- template<typename Type >
  ScalarParam< Type > & read (std::istream &in, const char ∗label, Type &value)

  Add and read a new required ScalarParam < Type > object.
- template<typename Type >
  ScalarParam< Type > & readOptional (std::istream &in, const char ∗label, Type &value)

  Add and read a new optional ScalarParam < Type > object.

Additional Inherited Members

12.38.1 Detailed Description

template<int D>
class Pscf::Pspg::Discrete::DMixture< D >

Solver for a mixture of polymers (Discrete chain model).

A Mixture contains a list of Polymer and Solvent objects. Each such object can solve the single-molecule statistical mechanics problem for an ideal gas of the associated species in a set of specified chemical potential fields, and thereby compute concentrations and single-molecule partition functions. A Mixture is thus both a chemistry descriptor and an ideal-gas solver.

A Mixture is associated with a Mesh<D> object, which models a spatial discretization mesh.

Definition at line 42 of file DMixture.h.

12.38.2 Member Function Documentation

12.38.2.1 readParameters() `template<int D>`

```
void Pscf::Pspg::Discrete::DMixture< D >::readParameters (
            std::istream & in ) [virtual]
```

Read parameters from file and initialize.

Parameters

in	input parameter file

Reimplemented from Pscf::Pspg::Discrete::DMixtureTmpl< DPolymer< D >, Solvent< D > >.
Definition at line 40 of file DMixture.tpp.
References UTIL_CHECK.

12.38.2.2 read() `template<int D>`

```
template<typename Type >
ScalarParam< Type > & Util::ParamComposite::read (
            typename Type ) [protected]
```

Add and read a new required ScalarParam < Type > object.
This is equivalent to ScalarParam<Type>(in, label, value, true).

Parameters

in	input stream for reading
label	Label string
value	reference to new ScalarParam< Type >

Definition at line 1156 of file ParamComposite.h.

12.38.2.3 readOptional() `template<int D>`

```
template<typename Type >
ScalarParam< Type > & Util::ParamComposite::readOptional (
            typename Type ) [inline], [protected]
```

Add and read a new optional ScalarParam < Type > object.
This is equivalent to ScalarParam<Type>(in, label, value, false).

Parameters

in	input stream for reading
label	Label string
value	reference to new ScalarParam< Type >

Definition at line 1164 of file ParamComposite.h.
The documentation for this class was generated from the following files:

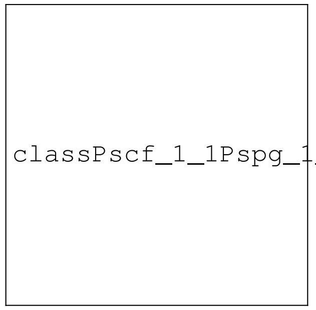
- DMixture.h
- DMixture.tpp

12.39 Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS > Class Template Reference

A mixture of polymer and solvent species.

`#include <DMixtureTmpl.h>`

Inheritance diagram for Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >:



Public Types

- typedef TP DPolymer

  Polymer species solver type.
- typedef TS Solvent

  Solvent species solver type.

Public Member Functions

- DMixtureTmpl ()

  Constructor.
- ~DMixtureTmpl ()

  Destructor.
- virtual void readParameters (std::istream &in)

  Read parameters from file and initialize.

Accessors (by non-const reference)

- Monomer & monomer (int id)

  Get a Monomer type descriptor.
- DPolymer & polymer (int id)

  Get a polymer object.
- Solvent & solvent (int id)

  Set a solvent solver object.

Accessors (by value)

- int nMonomer () const

  Get number of monomer types.
- int nPolymer () const

  Get number of polymer species.
- int nSolvent () const

  Get number of solvent (point particle) species.

Additional Inherited Members

12.39.1 Detailed Description

template<class TP, class TS>
class Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >

A mixture of polymer and solvent species.
Definition at line 30 of file DMixtureTmpl.h.

12.39.2 Member Typedef Documentation

12.39.2.1 DPolymer `template<class TP , class TS >`

`typedef TP Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >::DPolymer`

Polymer species solver type.
Definition at line 38 of file DMixtureTmpl.h.

12.39.2.2 Solvent `template<class TP , class TS >`

`typedef TS Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >::Solvent`

Solvent species solver type.
Definition at line 43 of file DMixtureTmpl.h.

12.39.3 Constructor & Destructor Documentation

12.39.3.1 DMixtureTmpl() `template<class TP , class TS >`

`Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >::DMixtureTmpl`

Constructor.
Definition at line 189 of file DMixtureTmpl.h.

12.39.3.2 ∼DMixtureTmpl() `template<class TP , class TS >`

`Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >::~DMixtureTmpl`

Destructor.
Definition at line 204 of file DMixtureTmpl.h.

12.39.4 Member Function Documentation

12.39.4.1 readParameters() `template<class TP , class TS >`

```
void Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >::readParameters (
            std::istream & in )  [virtual]
```

Read parameters from file and initialize.

Parameters

in	input parameter file

Reimplemented from Util::ParamComposite.

Reimplemented in Pscf::Pspg::Discrete::DMixture< D >.
Definition at line 212 of file DMixtureTmpl.h.

12.39.4.2 monomer() template<class TP , class TS >

Monomer & Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >::monomer (
int *id*) [inline]

Get a Monomer type descriptor.

Parameters

id	integer monomer type index (0 <= id < nMonomer)

Definition at line 166 of file DMixtureTmpl.h.

12.39.4.3 polymer() template<class TP , class TS >

TP & Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >::polymer (
int *id*) [inline]

Get a polymer object.

Parameters

id	integer polymer species index (0 <= id < nPolymer)

Definition at line 172 of file DMixtureTmpl.h.

12.39.4.4 solvent() template<class TP , class TS >

TS & Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >::solvent (
int *id*) [inline]

Set a solvent solver object.

Parameters

id	integer solvent species index (0 <= id < nSolvent)

Definition at line 178 of file DMixtureTmpl.h.

12.39.4.5 nMonomer() template<class TP , class TS >

int Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >::nMonomer [inline]

Get number of monomer types.
Definition at line 148 of file DMixtureTmpl.h.

12.39.4.6 nPolymer() template<class TP , class TS >

int Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >::nPolymer [inline]

Get number of polymer species.
Definition at line 154 of file DMixtureTmpl.h.

12.39.4.7 nSolvent() `template<class TP , class TS >`

`int Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >::nSolvent [inline]`

Get number of solvent (point particle) species.
Definition at line 160 of file DMixtureTmpl.h.
The documentation for this class was generated from the following file:

- DMixtureTmpl.h

12.40 Pscf::Pspg::Discrete::DPolymer< D > Class Template Reference

Descriptor and solver for a branched polymer species (Discrete chain model).
`#include <DPolymer.h>`
Inheritance diagram for Pscf::Pspg::Discrete::DPolymer< D >:

classPscf_1_1Pspg_1_1Discrete_1_1DPolymer-eps-converted-to.pc

Protected Member Functions

- void setClassName (const char *className)

  Set class name string.

Additional Inherited Members

12.40.1 Detailed Description

template<int D>
class Pscf::Pspg::Discrete::DPolymer< D >

Descriptor and solver for a branched polymer species (Discrete chain model).
The block-bond concentrations stored in the constituent Bond<D> objects contain the block-bond concentrations (i.e., volume fractions) computed in the most recent call of the compute function.
The phi() and mu() accessor functions, which are inherited from DPolymerTmp< Bond<D> >, return the value of phi (spatial average volume fraction of a species) or mu (chemical potential) computed in the last call of the compute function. If the ensemble for this species is closed, phi is read from the parameter file and mu is computed. If the ensemble is open, mu is read from the parameter file and phi is computed.
Definition at line 36 of file DPolymer.h.

12.40.2 Member Function Documentation

12.40.2.1 setClassName() `template<int D>`

`void Util::ParamComposite::setClassName [protected]`

Set class name string.
Should be set in subclass constructor.

Definition at line 377 of file ParamComposite.cpp.
The documentation for this class was generated from the following files:

- DPolymer.h
- DPolymer.tpp

12.41 Pscf::Pspg::Discrete::DPropagator< D > Class Template Reference

CKE solver for one-direction of one bond.

```
#include <DPropagator.h>
```

Inheritance diagram for Pscf::Pspg::Discrete::DPropagator< D >:

classPscf_1_1Pspg_1_1Discrete_1_1DPropagator-eps-convert

Additional Inherited Members

12.41.1 Detailed Description

template<int D>
class Pscf::Pspg::Discrete::DPropagator< D >

CKE solver for one-direction of one bond.
Definition at line 38 of file DPropagator.h.
The documentation for this class was generated from the following files:

- DPropagator.h
- DPropagator.tpp

12.42 Pscf::Pspg::Discrete::Iterator< D > Class Template Reference

Base class for iterative solvers for SCF equations.

```
#include <Iterator.h>
```

Inheritance diagram for Pscf::Pspg::Discrete::Iterator< D >:

classPscf_1_1Pspg_1_1Discrete_1_1Iterator-eps-converted-to.pdf

Additional Inherited Members

12.42.1 Detailed Description

template<int D>
class Pscf::Pspg::Discrete::Iterator< D >

Base class for iterative solvers for SCF equations.
Definition at line 24 of file pgd/iterator/Iterator.h.
The documentation for this class was generated from the following files:

- pgd/iterator/Iterator.h
- pgd/iterator/Iterator.tpp

12.43 Pscf::Pspg::Discrete::System< D > Class Template Reference

Public Member Functions

- FileMaster & fileMaster ()

    Accessors (access objects by reference)

12.43.1 Detailed Description

template<int D>
class Pscf::Pspg::Discrete::System< D >

Definition at line 14 of file pgd/iterator/Iterator.h.

12.43.2 Member Function Documentation

12.43.2.1 fileMaster() `template<int D>`

`FileMaster & Pscf::Pspg::Discrete::System< D >::fileMaster [inline]`

Accessors (access objects by reference)
Definition at line 185 of file pgd/System.h.
The documentation for this class was generated from the following files:

- pgd/iterator/Iterator.h
- pgd/System.h
- pgd/System.tpp

12.44 Pscf::Pspg::FFT< D > Class Template Reference

Fourier transform wrapper for real data.
`#include <FFT.h>`

Public Member Functions

- FFT ()

    Default constructor.
- virtual ∼FFT ()

    Destructor.
- void setup (RDField< D > &rDField, RDFieldDft< D > &kDField)

    Check and setup grid dimensions if necessary.
- void forwardTransform (RDField< D > &in, RDFieldDft< D > &out)

    Compute forward (real-to-complex) Fourier transform.
- void inverseTransform (RDFieldDft< D > &in, RDField< D > &out)

Compute inverse (complex-to-real) Fourier transform.

- const IntVec< D > & meshDimensions () const

  Return the dimensions of the grid for which this was allocated.

12.44.1 Detailed Description

template<int D>
class Pscf::Pspg::FFT< D >

Fourier transform wrapper for real data.
Definition at line 33 of file FFT.h.

12.44.2 Constructor & Destructor Documentation

12.44.2.1 FFT() `template<int D>`

`Pscf::Pspg::FFT< D >::FFT`

Default constructor.
Definition at line 25 of file FFT.tpp.

12.44.2.2 ∼FFT() `template<int D>`

`Pscf::Pspg::FFT< D >::~FFT [virtual]`

Destructor.
Definition at line 38 of file FFT.tpp.

12.44.3 Member Function Documentation

12.44.3.1 setup() `template<int D>`

```
void Pscf::Pspg::FFT< D >::setup (
            RDField< D > & rDField,
            RDFieldDft< D > & kDField )
```

Check and setup grid dimensions if necessary.

Parameters

rDField	real data on r-space grid (device mem)
kDField	complex data on k-space grid (device mem)

Definition at line 52 of file FFT.tpp.
References Pscf::Pspg::DField< cudaReal >::capacity(), Pscf::Pspg::DField< cudaComplex >::capacity(), Pscf::Pspg::RDFieldDft< D >::meshDimensions(), Pscf::Pspg::RDField< D >::meshDimensions(), and UTIL_CHECK.

12.44.3.2 forwardTransform() `template<int D>`

```
void Pscf::Pspg::FFT< D >::forwardTransform (
            RDField< D > & in,
            RDFieldDft< D > & out )
```

Compute forward (real-to-complex) Fourier transform.

Parameters

in	array of real values on r-space grid (device mem)
out	array of complex values on k-space grid (device mem)

Definition at line 88 of file FFT.tpp.
References Pscf::Pspg::DField< cudaReal >::capacity(), Pscf::Pspg::DField< cudaComplex >::capacity(), Pscf::Pspg::DField< cudaReal >::cDField(), Pscf::Pspg::DField< cudaComplex >::cDField(), Pscf::Pspg::ThreadGrid::setThreadsLogical(), and UTIL_CHECK.

12.44.3.3 inverseTransform() `template<int D>`

```
void Pscf::Pspg::FFT< D >::inverseTransform (
            RDFieldDft< D > & in,
            RDField< D > & out )
```

Compute inverse (complex-to-real) Fourier transform.

Parameters

in	array of complex values on k-space grid (device mem)
out	array of real values on r-space grid (device mem)

Definition at line 127 of file FFT.tpp.
References Pscf::Pspg::DField< cudaComplex >::cDField(), and Pscf::Pspg::DField< cudaReal >::cDField().

12.44.3.4 meshDimensions() `template<int D>`

```
const IntVec< D > & Pscf::Pspg::FFT< D >::meshDimensions  [inline]
```

Return the dimensions of the grid for which this was allocated.
Definition at line 126 of file FFT.h.
The documentation for this class was generated from the following files:

- FFT.h
- FFT.tpp

12.45 Pscf::Pspg::FFTBatched< D > Class Template Reference

Fourier transform wrapper for real data.
`#include <FFTBatched.h>`

Public Member Functions

- FFTBatched ()

    Default constructor.
- virtual ~FFTBatched ()

    Destructor.
- void setup (RDField< D > &rDField, RDFieldDft< D > &kDField)

    Check and setup grid dimensions if necessary.
- void forwardTransform (RDField< D > &in, RDFieldDft< D > &out)

    Compute forward (real-to-complex) Fourier transform.
- void inverseTransform (RDFieldDft< D > &in, RDField< D > &out)

Compute inverse (complex-to-real) Fourier transform.

- const IntVec< D > & meshDimensions () const

  Return the dimensions of the grid for which this was allocated.

12.45.1 Detailed Description

template<int D>
class Pscf::Pspg::FFTBatched< D >

Fourier transform wrapper for real data.
Definition at line 36 of file FFTBatched.h.

12.45.2 Constructor & Destructor Documentation

12.45.2.1 FFTBatched() `template<int D>`

`Pscf::Pspg::FFTBatched< D >::FFTBatched`

Default constructor.
Definition at line 39 of file FFTBatched.tpp.

12.45.2.2 ~FFTBatched() `template<int D>`

`Pscf::Pspg::FFTBatched< D >::~FFTBatched [virtual]`

Destructor.
Definition at line 52 of file FFTBatched.tpp.

12.45.3 Member Function Documentation

12.45.3.1 setup() `template<int D>`

```
void Pscf::Pspg::FFTBatched< D >::setup (
            RDField< D > & rDField,
            RDFieldDft< D > & kDField )
```

Check and setup grid dimensions if necessary.

Parameters

rDField	real data on r-space grid (device mem)
kDField	complex data on k-space grid (device mem)

Definition at line 66 of file FFTBatched.tpp.
References Pscf::Pspg::RDFieldDft< D >::meshDimensions(), Pscf::Pspg::RDField< D >::meshDimensions(), and UTIL_CHECK.

12.45.3.2 forwardTransform() `template<int D>`

```
void Pscf::Pspg::FFTBatched< D >::forwardTransform (
            RDField< D > & in,
            RDFieldDft< D > & out )
```

Compute forward (real-to-complex) Fourier transform.

Parameters

in	array of real values on r-space grid (device mem)
out	array of complex values on k-space grid (device mem)

Definition at line 241 of file FFTBatched.tpp.

References Pscf::Pspg::DField< cudaReal >::capacity(), Pscf::Pspg::DField< cudaComplex >::capacity(), Pscf::↩ Pspg::DField< cudaReal >::cDField(), Pscf::Pspg::DField< cudaComplex >::cDField(), Pscf::Pspg::ThreadGrid::set↩ ThreadsLogical(), and UTIL_CHECK.

12.45.3.3 inverseTransform() `template<int D>`

```
void Pscf::Pspg::FFTBatched< D >::inverseTransform (
            RDFieldDft< D > & in,
            RDField< D > & out )
```

Compute inverse (complex-to-real) Fourier transform.

Parameters

in	array of complex values on k-space grid (device mem)
out	array of real values on r-space grid (device mem)

Definition at line 315 of file FFTBatched.tpp.

References Pscf::Pspg::DField< cudaComplex >::cDField(), and Pscf::Pspg::DField< cudaReal >::cDField().

12.45.3.4 meshDimensions() `template<int D>`

```
const IntVec< D > & Pscf::Pspg::FFTBatched< D >::meshDimensions  [inline]
```

Return the dimensions of the grid for which this was allocated.

Definition at line 123 of file FFTBatched.h.

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

- FFTBatched.h
- FFTBatched.tpp

12.46 Pscf::Pspg::FieldIo< D > Class Template Reference

File input/output operations for fields in several file formats.

```
#include <FieldIo.h>
```

Public Member Functions

- FieldIo ()

  Constructor.
- ∼FieldIo ()

  Destructor.
- void associate (UnitCell< D > &unitCell, Mesh< D > &mesh, FFT< D > &fft, std::string &groupName, Basis< D > &basis, FileMaster &fileMaster)

  Get and store addresses of associated objects.

Field File IO

- void readFieldsBasis (std::istream &in, DArray< RDField< D > > &fields)

Read concentration or chemical potential field components from file.
- void readFieldsBasis (std::string filename, DArray< RDField< D > > &fields)

  Read concentration or chemical potential field components from file.
- void writeFieldsBasis (std::ostream &out, DArray< RDField< D > > const &fields)

  Write concentration or chemical potential field components to file.
- void writeFieldsBasis (std::string filename, DArray< RDField< D > > const &fields)

  Write concentration or chemical potential field components to file.
- void readFieldsRGrid (std::istream &in, DArray< RDField< D > > &fields)

  Read array of RField objects (fields on an r-space grid) from file.
- void readFieldsRGrid (std::string filename, DArray< RDField< D > > &fields)

  Read array of RField objects (fields on an r-space grid) from file.
- void writeFieldsRGrid (std::ostream &out, DArray< RDField< D > > const &fields)

  Write array of RField objects (fields on an r-space grid) to file.
- void writeFieldsRGrid (std::string filename, DArray< RDField< D > > const &fields)

  Write array of RField objects (fields on an r-space grid) to file.
- void readFieldsKGrid (std::istream &in, DArray< RDFieldDft< D > > &fields)

  Read array of RFieldDft objects (k-space fields) from file.
- void readFieldsKGrid (std::string filename, DArray< RDFieldDft< D > > &fields)

  Read array of RFieldDft objects (k-space fields) from file.
- void writeFieldsKGrid (std::ostream &out, DArray< RDFieldDft< D > > const &fields)

  Write array of RFieldDft objects (k-space fields) to file.
- void writeFieldsKGrid (std::string filename, DArray< RDFieldDft< D > > const &fields)

  Write array of RFieldDft objects (k-space fields) to a file.
- void writeFieldHeader (std::ostream &out, int nMonomer) const

  Write header for field file (fortran pscf format)

Field Format Conversion

- void convertBasisToKGrid (RDField< D > const &components, RDFieldDft< D > &dft)

  Convert field from symmetrized basis to Fourier transform (k-grid).
- void convertBasisToKGrid (DArray< RDField< D > > &in, DArray< RDFieldDft< D > > &out)

  Convert fields from symmetrized basis to Fourier transform (kgrid).
- void convertKGridToBasis (RDFieldDft< D > const &dft, RDField< D > &components)

  Convert field from Fourier transform (k-grid) to symmetrized basis.
- void convertKGridToBasis (DArray< RDFieldDft< D > > &in, DArray< RDField< D > > &out)

  Convert fields from Fourier transform (kgrid) to symmetrized basis.
- void convertBasisToRGrid (DArray< RDField< D > > &in, DArray< RDField< D > > &out)

  Convert fields from symmetrized basis to spatial grid (rgrid).
- void convertRGridToBasis (DArray< RDField< D > > &in, DArray< RDField< D > > &out)

  Convert fields from spatial grid (rgrid) to symmetrized basis.

12.46.1 Detailed Description

template< int D >
class Pscf::Pspg::FieldIo< D >

File input/output operations for fields in several file formats.
Definition at line 35 of file FieldIo.h.

12.46.2 Constructor & Destructor Documentation

12.46.2.1 FieldIo() `template<int D>`

```
Pscf::Pspg::FieldIo< D >::FieldIo
```

Constructor.

Definition at line 34 of file FieldIo.tpp.

12.46.2.2 ∼FieldIo() `template<int D>`

```
Pscf::Pspg::FieldIo< D >::~FieldIo
```

Destructor.

Definition at line 47 of file FieldIo.tpp.

12.46.3 Member Function Documentation

12.46.3.1 associate() `template<int D>`

```
void Pscf::Pspg::FieldIo< D >::associate (
            UnitCell< D > & unitCell,
            Mesh< D > & mesh,
            FFT< D > & fft,
            std::string & groupName,
            Basis< D > & basis,
            FileMaster & fileMaster )
```

Get and store addresses of associated objects.

Parameters

unitCell	associated crystallographic UnitCell<D>
mesh	associated spatial discretization Mesh<D>
fft	associated FFT object for fast transforms
groupName	space group name string
basis	associated Basis object
fileMaster	associated FileMaster (for file paths)

Definition at line 54 of file FieldIo.tpp.

12.46.3.2 readFieldsBasis() `[1/2]` `template<int D>`

```
void Pscf::Pspg::FieldIo< D >::readFieldsBasis (
            std::istream & in,
            DArray< RDField< D > > & fields )
```

Read concentration or chemical potential field components from file.

This function reads components in a symmetry adapted basis from file in.

The capacity of DArray fields is equal to nMonomer, and element fields[i] is a DArray containing components of the field associated with monomer type i.

Parameters

in	input stream (i.e., input file)
fields	array of fields (symmetry adapted basis components)

Definition at line 70 of file FieldIo.tpp.

References Util::DArray< Data >::allocate(), and UTIL_CHECK.

12.46.3.3 readFieldsBasis() [2/2]

```
template<int D>
void Pscf::Pspg::FieldIo< D >::readFieldsBasis (
    std::string filename,
    DArray< RDField< D > > & fields )
```

Read concentration or chemical potential field components from file.

This function opens an input file with the specified filename, reads components in symmetry-adapted form from that file, and closes the file.

Parameters

filename	name of input file
fields	array of fields (symmetry adapted basis components)

Definition at line 147 of file FieldIo.tpp.

12.46.3.4 writeFieldsBasis() [1/2]

```
template<int D>
void Pscf::Pspg::FieldIo< D >::writeFieldsBasis (
    std::ostream & out,
    DArray< RDField< D > > const & fields )
```

Write concentration or chemical potential field components to file.

This function writes components in a symmetry adapted basis.

Parameters

out	output stream (i.e., output file)
fields	array of fields (symmetry adapted basis components)

Definition at line 158 of file FieldIo.tpp.

References Util::DArray< Data >::allocate(), and UTIL_CHECK.

12.46.3.5 writeFieldsBasis() [2/2]

```
template<int D>
void Pscf::Pspg::FieldIo< D >::writeFieldsBasis (
    std::string filename,
    DArray< RDField< D > > const & fields )
```

Write concentration or chemical potential field components to file.

This function opens an output file with the specified filename, writes components in symmetry-adapted form to that file, and then closes the file.

Parameters

filename	name of input file
fields	array of fields (symmetry adapted basis components)

Definition at line 206 of file FieldIo.tpp.

12.46.3.6 readFieldsRGrid() [1/2]

```
template<int D>
```

```
void Pscf::Pspg::FieldIo< D >::readFieldsRGrid (
            std::istream & in,
            DArray< RDField< D > > & fields )
```

Read array of RField objects (fields on an r-space grid) from file.

The capacity of array fields is equal to nMonomer, and element fields[i] is the RField<D> associated with monomer type i.

Parameters

in	input stream (i.e., input file)
fields	array of RField fields (r-space grid)

Definition at line 216 of file FieldIo.tpp.

References Util::DArray< Data >::allocate(), and UTIL_CHECK.

12.46.3.7 readFieldsRGrid() [2/2] template<int D>

```
void Pscf::Pspg::FieldIo< D >::readFieldsRGrid (
            std::string filename,
            DArray< RDField< D > > & fields )
```

Read array of RField objects (fields on an r-space grid) from file.

The capacity of array fields is equal to nMonomer, and element fields[i] is the RField<D> associated with monomer type i.

This function opens an input file with the specified filename, reads fields in RField<D> real-space grid format from that file, and then closes the file.

Parameters

filename	name of input file
fields	array of RField fields (r-space grid)

Definition at line 281 of file FieldIo.tpp.

12.46.3.8 writeFieldsRGrid() [1/2] template<int D>

```
void Pscf::Pspg::FieldIo< D >::writeFieldsRGrid (
            std::ostream & out,
            DArray< RDField< D > > const & fields )
```

Write array of RField objects (fields on an r-space grid) to file.

Parameters

out	output stream (i.e., output file)
fields	array of RField fields (r-space grid)

Definition at line 291 of file FieldIo.tpp.

References Util::DArray< Data >::allocate(), and UTIL_CHECK.

12.46.3.9 writeFieldsRGrid() [2/2] template<int D>

```
void Pscf::Pspg::FieldIo< D >::writeFieldsRGrid (
            std::string filename,
```

```
DArray< RDField< D > > const & fields )
```

Write array of RField objects (fields on an r-space grid) to file.

This function opens an output file with the specified filename, writes fields in RField<D> real-space grid format to that file, and then closes the file.

Parameters

filename	name of output file
fields	array of RField fields (r-space grid)

Definition at line 351 of file FieldIo.tpp.

12.46.3.10 readFieldsKGrid() [1/2]

```
template<int D>
void Pscf::Pspg::FieldIo< D >::readFieldsKGrid (
            std::istream & in,
            DArray< RDFieldDft< D > > & fields )
```

Read array of RFieldDft objects (k-space fields) from file.

The capacity of the array is equal to nMonomer, and element fields[i] is the discrete Fourier transform of the field for monomer type i.

Parameters

in	input stream (i.e., input file)
fields	array of RFieldDft fields (k-space grid)

Definition at line 361 of file FieldIo.tpp.

References Util::DArray< Data >::allocate(), and UTIL_CHECK.

12.46.3.11 readFieldsKGrid() [2/2]

```
template<int D>
void Pscf::Pspg::FieldIo< D >::readFieldsKGrid (
            std::string filename,
            DArray< RDFieldDft< D > > & fields )
```

Read array of RFieldDft objects (k-space fields) from file.

This function opens a file with name filename, reads discrete Fourier components (Dft) of fields from that file, and closes the file.

The capacity of the array is equal to nMonomer, and element fields[i] is the discrete Fourier transform of the field for monomer type i.

Parameters

filename	name of input file
fields	array of RFieldDft fields (k-space grid)

Definition at line 413 of file FieldIo.tpp.

12.46.3.12 writeFieldsKGrid() [1/2]

```
template<int D>
void Pscf::Pspg::FieldIo< D >::writeFieldsKGrid (
            std::ostream & out,
            DArray< RDFieldDft< D > > const & fields )
```

Write array of RFieldDft objects (k-space fields) to file.
The capacity of the array fields is equal to nMonomer. Element fields[i] is the discrete Fourier transform of the field for monomer type i.

Parameters

out	output stream (i.e., output file)
fields	array of RFieldDft fields

Definition at line 423 of file FieldIo.tpp.
References Util::DArray< Data >::allocate(), and UTIL_CHECK.

12.46.3.13 writeFieldsKGrid() [2/2] template<int D>

```
void Pscf::Pspg::FieldIo< D >::writeFieldsKGrid (
            std::string filename,
            DArray< RDFieldDft< D > > const & fields )
```

Write array of RFieldDft objects (k-space fields) to a file.
This function opens a file with name filename, writes discrete Fourier transform components (DFT) components of fields to that file, and closes the file.

Parameters

filename	name of output file.
fields	array of RFieldDft fields (k-space grid)

Definition at line 469 of file FieldIo.tpp.

12.46.3.14 writeFieldHeader() template<int D>

```
void Pscf::Pspg::FieldIo< D >::writeFieldHeader (
            std::ostream & out,
            int nMonomer ) const
```

Write header for field file (fortran pscf format)

Parameters

out	output stream (i.e., output file)
nMonomer	number of monomer types

Definition at line 509 of file FieldIo.tpp.
References Pscf::writeUnitCellHeader().

12.46.3.15 convertBasisToKGrid() [1/2] template<int D>

```
void Pscf::Pspg::FieldIo< D >::convertBasisToKGrid (
            RDField< D > const & components,
            RDFieldDft< D > & dft )
```

Convert field from symmetrized basis to Fourier transform (k-grid).

Parameters

Parameters

components	coefficients of symmetry-adapted basis functions
dft	discrete Fourier transform of a real field

Definition at line 522 of file FieldIo.tpp.
References Pscf::Basis< D >::Star::beginId, Pscf::Basis< D >::Star::cancel, Pscf::Pspg::DField< cudaReal >::cD↩ Field(), Pscf::Pspg::DField< cudaComplex >::cDField(), Pscf::Pspg::RDFieldDft< D >::dftDimensions(), Pscf::Basis< D >::Star::invertFlag, Pscf::Mesh< D >::rank(), Pscf::Mesh< D >::size(), UTIL_CHECK, and UTIL_THROW.

12.46.3.16 convertBasisToKGrid() [2/2]

```
template<int D>
void Pscf::Pspg::FieldIo< D >::convertBasisToKGrid (
            DArray< RDField< D > > & in,
            DArray< RDFieldDft< D > > & out )
```

Convert fields from symmetrized basis to Fourier transform (kgrid).
The in and out parameters are arrays of fields, in which element number i is the field associated with monomer type i.

Parameters

in	components of fields in symmetry adapted basis
out	fields defined as discrete Fourier transforms (k-grid)

Definition at line 759 of file FieldIo.tpp.
References UTIL_ASSERT.

12.46.3.17 convertKGridToBasis() [1/2]

```
template<int D>
void Pscf::Pspg::FieldIo< D >::convertKGridToBasis (
            RDFieldDft< D > const & dft,
            RDField< D > & components )
```

Convert field from Fourier transform (k-grid) to symmetrized basis.

Parameters

dft	complex DFT (k-grid) representation of a field.
components	coefficients of symmetry-adapted basis functions.

Definition at line 649 of file FieldIo.tpp.
References Pscf::Basis< D >::Star::beginId, Pscf::Basis< D >::Star::cancel, Pscf::Pspg::DField< cudaReal >::cD↩ Field(), Pscf::Pspg::DField< cudaComplex >::cDField(), Pscf::Pspg::RDFieldDft< D >::dftDimensions(), Pscf::Basis< D >::Star::endId, Pscf::Basis< D >::Star::invertFlag, Pscf::Mesh< D >::rank(), Pscf::Basis< D >::Star::size, UTIL_↩ CHECK, and UTIL_THROW.

12.46.3.18 convertKGridToBasis() [2/2]

```
template<int D>
void Pscf::Pspg::FieldIo< D >::convertKGridToBasis (
            DArray< RDFieldDft< D > > & in,
            DArray< RDField< D > > & out )
```

Convert fields from Fourier transform (kgrid) to symmetrized basis.
The in and out parameters are each an array of fields, in which element i is the field associated with monomer type i.

Parameters

in	fields defined as discrete Fourier transforms (k-grid)
out	components of fields in symmetry adapted basis

Definition at line 854 of file FieldIo.tpp.
References UTIL_ASSERT.

12.46.3.19 convertBasisToRGrid() `template<int D>`

```
void Pscf::Pspg::FieldIo< D >::convertBasisToRGrid (
            DArray< RDField< D > > & in,
            DArray< RDField< D > > & out )
```

Convert fields from symmetrized basis to spatial grid (rgrid).

Parameters

in	fields in symmetry adapted basis form
out	fields defined on real-space grid

Definition at line 869 of file FieldIo.tpp.
References Util::DArray< Data >::allocate(), Util::DArray< Data >::deallocate(), and UTIL_ASSERT.

12.46.3.20 convertRGridToBasis() `template<int D>`

```
void Pscf::Pspg::FieldIo< D >::convertRGridToBasis (
            DArray< RDField< D > > & in,
            DArray< RDField< D > > & out )
```

Convert fields from spatial grid (rgrid) to symmetrized basis.

Parameters

in	fields defined on real-space grid
out	fields in symmetry adapted basis form

Definition at line 893 of file FieldIo.tpp.
References Util::DArray< Data >::allocate(), Util::DArray< Data >::deallocate(), and UTIL_ASSERT.
The documentation for this class was generated from the following files:

- FieldIo.h
- FieldIo.tpp

12.47 Pscf::Pspg::HistMat< Data > Class Template Reference

Inheritance diagram for Pscf::Pspg::HistMat< Data >:

classPscf_1_1Pspg_1_1HistMat-eps-converted-to.pdf

Additional Inherited Members

12.47.1 Detailed Description

template<typename Data>
class Pscf::Pspg::HistMat< Data >

Definition at line 14 of file HistMat.h.
The documentation for this class was generated from the following file:

- HistMat.h

12.48 Pscf::Pspg::RDField< D > Class Template Reference

Field of real single precision values on an FFT mesh on a device.

```
#include <RDField.h>
```

Inheritance diagram for Pscf::Pspg::RDField< D >:

classPscf_1_1Pspg_1_1RDField-eps-converted-to.pdf

Public Member Functions

- RDField ()

  Default constructor.
- RDField (RDField< D > const &other)

  Copy constructor.
- virtual ∼RDField ()

  Destructor.
- RDField & operator= (RDField< D > const &other)

  Assignment operator.
- void allocate (const IntVec< D > &meshDimensions)

  Allocate the underlying C array for an FFT grid.
- const IntVec< D > & meshDimensions () const

  Return mesh dimensions by constant reference.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize a Field to/from an Archive.

Additional Inherited Members

12.48.1 Detailed Description

template<int D>
class Pscf::Pspg::RDField< D >

Field of real single precision values on an FFT mesh on a device.
cudaReal = float
Definition at line 33 of file RDField.h.

12.48.2 Constructor & Destructor Documentation

12.48.2.1 RDField() [1/2] `template<int D>`

`Pscf::Pspg::RDField< D >::RDField`

Default constructor.
Definition at line 22 of file RDField.tpp.

12.48.2.2 RDField() [2/2] `template<int D>`

```
Pscf::Pspg::RDField< D >::RDField (
            RDField< D > const & other )
```

Copy constructor.
Allocates new memory and copies all elements by value.
uses memcpy! slow!

Parameters

other	the RField to be copied.

Definition at line 41 of file RDField.tpp.
References Pscf::Pspg::DField< cudaReal >::capacity_, Pscf::Pspg::DField< cudaReal >::cDField(), Pscf::Pspg::DField< cudaReal >::data_, Pscf::Pspg::DField< cudaReal >::isAllocated(), and UTIL_THROW.

12.48.2.3 ∼RDField() `template<int D>`

`Pscf::Pspg::RDField< D >::∼RDField [virtual]`

Destructor.
Deletes underlying C array, if allocated previously.
Definition at line 30 of file RDField.tpp.

12.48.3 Member Function Documentation

12.48.3.1 operator=() `template<int D>`

```
RDField< D > & Pscf::Pspg::RDField< D >::operator= (
            RDField< D > const & other )
```

Assignment operator.
If this Field is not allocated, launch a kernel to swap memory.
If this and the other Field are both allocated, the capacities must be exactly equal. If so, this method copies all elements.
uses memcpy! slow!

Parameters

other	the RHS RField

Definition at line 67 of file RDField.tpp.
References Pscf::Pspg::DField< cudaReal >::capacity(), Pscf::Pspg::DField< cudaReal >::capacity_, Pscf::Pspg::↩
DField< cudaReal >::cDField(), Pscf::Pspg::DField< cudaReal >::isAllocated(), and UTIL_THROW.

12.48.3.2 allocate() `template<int D>`

```
void Pscf::Pspg::RDField< D >::allocate (
            const IntVec< D > & meshDimensions )
```

Allocate the underlying C array for an FFT grid.

Exceptions

Exception	if the RField is already allocated.

Parameters

meshDimensions	vector containing number of grid points in each direction

Definition at line 109 of file RDField.h.
References Pscf::Pspg::DField< Data >::allocate(), and UTIL_CHECK.
Referenced by Pscf::Pspg::Continuous::AmIterator< D >::minimizeCoeff().

12.48.3.3 meshDimensions() `template<int D>`

```
const IntVec< D > & Pscf::Pspg::RDField< D >::meshDimensions  [inline]
```

Return mesh dimensions by constant reference.
Definition at line 124 of file RDField.h.
Referenced by Pscf::Pspg::FFT< D >::setup(), and Pscf::Pspg::FFTBatched< D >::setup().

12.48.3.4 serialize() `template<int D>`

```
template<class Archive >
void Pscf::Pspg::RDField< D >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize a Field to/from an Archive.
Temporarily uses a memcpy

Parameters

ar	archive
version	archive version id

Definition at line 132 of file RDField.h.
References UTIL_THROW.
The documentation for this class was generated from the following files:

- RDField.h

- RDField.tpp

12.49 Pscf::Pspg::RDFieldDft< D > Class Template Reference

Fourier transform of a real field on an FFT mesh.

```
#include <RDFieldDft.h>
```

Inheritance diagram for Pscf::Pspg::RDFieldDft< D >:

classPscf_1_1Pspg_1_1RDFieldDft-eps-converted-to.pdf

Public Member Functions

- RDFieldDft ()

  Default constructor.
- RDFieldDft (const RDFieldDft< D > &other)

  Copy constructor.
- virtual ∼RDFieldDft ()

  Destructor.
- RDFieldDft< D > & operator= (const RDFieldDft< D > &other)

  Assignment operator.
- void allocate (const IntVec< D > &meshDimensions)

  Allocate the underlying C array for an FFT grid.
- const IntVec< D > & meshDimensions () const

  Return vector of mesh dimensions by constant reference.
- const IntVec< D > & dftDimensions () const

  Return vector of dft (Fourier) grid dimensions by constant reference.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize a Field to/from an Archive.

Additional Inherited Members

12.49.1 Detailed Description

template<int D>
class Pscf::Pspg::RDFieldDft< D >

Fourier transform of a real field on an FFT mesh.
Definition at line 30 of file RDFieldDft.h.

12.49.2 Constructor & Destructor Documentation

12.49.2.1 RDFieldDft() `[1/2]` `template<int D>`

`Pscf::Pspg::RDFieldDft< D >::RDFieldDft`

Default constructor.
Definition at line 22 of file RDFieldDft.tpp.

12.49.2.2 RDFieldDft() [2/2]

`template<int D>`

```
Pscf::Pspg::RDFieldDft< D >::RDFieldDft (
        const RDFieldDft< D > & other )
```

Copy constructor.

Allocates new memory and copies all elements by value.

Parameters

other	the RFieldDft to be copied.

Definition at line 41 of file RDFieldDft.tpp.

References Pscf::Pspg::DField< cudaComplex >::capacity_, Pscf::Pspg::DField< cudaComplex >::cDField(), Pscf::Pspg::DField< cudaComplex >::data_, Pscf::Pspg::DField< cudaComplex >::isAllocated(), and UTIL_THROW.

12.49.2.3 ∼RDFieldDft()

`template<int D>`

```
Pscf::Pspg::RDFieldDft< D >::~RDFieldDft [virtual]
```

Destructor.

Deletes underlying C array, if allocated previously.

Definition at line 30 of file RDFieldDft.tpp.

12.49.3 Member Function Documentation

12.49.3.1 operator=()

`template<int D>`

```
RDFieldDft< D > & Pscf::Pspg::RDFieldDft< D >::operator= (
        const RDFieldDft< D > & other )
```

Assignment operator.

If this Field is not allocated, allocates and copies all elements.

If this and the other Field are both allocated, the capacities must be exactly equal. If so, this method copies all elements.

Parameters

other	the RHS Field

Definition at line 66 of file RDFieldDft.tpp.

References Pscf::Pspg::DField< cudaComplex >::capacity(), Pscf::Pspg::DField< cudaComplex >::capacity_, Pscf::Pspg::DField< cudaComplex >::cDField(), Pscf::Pspg::DField< cudaComplex >::isAllocated(), and UTIL_THROW.

12.49.3.2 allocate()

`template<int D>`

```
void Pscf::Pspg::RDFieldDft< D >::allocate (
        const IntVec< D > & meshDimensions )
```

Allocate the underlying C array for an FFT grid.

Exceptions

Exception	if the RFieldDft is already allocated.

Parameters

meshDimensions	vector containing number of grid points in each direction

Definition at line 116 of file RDFieldDft.h.
References Pscf::Pspg::DField< Data >::allocate(), and UTIL_CHECK.

12.49.3.3 meshDimensions() `template<int D>`

`const IntVec< D > & Pscf::Pspg::RDFieldDft< D >::meshDimensions [inline]`

Return vector of mesh dimensions by constant reference.
Definition at line 137 of file RDFieldDft.h.
Referenced by Pscf::Pspg::FFT< D >::setup(), and Pscf::Pspg::FFTBatched< D >::setup().

12.49.3.4 dftDimensions() `template<int D>`

`const IntVec< D > & Pscf::Pspg::RDFieldDft< D >::dftDimensions [inline]`

Return vector of dft (Fourier) grid dimensions by constant reference.
The last element of dftDimensions() and meshDimensions() differ by about a factor of two: dftDimension()[D-1] = meshDimensions()/2 + 1. For D > 1, other elements are equal.
Definition at line 144 of file RDFieldDft.h.
Referenced by Pscf::Pspg::FieldIo< D >::convertBasisToKGrid(), and Pscf::Pspg::FieldIo< D >::convertKGridTo↩ Basis().

12.49.3.5 serialize() `template<int D>`

```
template<class Archive >
void Pscf::Pspg::RDFieldDft< D >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize a Field to/from an Archive.

Parameters

ar	archive
version	archive version id

Definition at line 152 of file RDFieldDft.h.
References UTIL_THROW.
The documentation for this class was generated from the following files:

- RDFieldDft.h
- RDFieldDft.tpp

12.50 Pscf::Pspg::Solvent< D > Class Template Reference

Class representing a solvent species.
`#include <Solvent.h>`
Inheritance diagram for Pscf::Pspg::Solvent< D >:

classPscf_1_1Pspg_1_1Solvent-eps-converted-to.pdf

Public Types

- typedef RDField< D > CField
    Monomer concentration field.
- typedef RDField< D > WField
    Monomer chemical potential field.

Public Member Functions

- Solvent ()
    Constructor.
- ~Solvent ()
    Constructor.
- virtual void compute (WField const &wField)
    Compute monomer concentration field and phi and/or mu.
- const CField & concentration () const
    Get monomer concentration field for this solvent.

Additional Inherited Members

12.50.1 Detailed Description

template<int D>
class Pscf::Pspg::Solvent< D >

Class representing a solvent species.
Definition at line 27 of file Solvent.h.

12.50.2 Member Typedef Documentation

12.50.2.1 CField `template<int D>`

`typedef RDField<D> Pscf::Pspg::Solvent< D >::CField`
Monomer concentration field.
Definition at line 33 of file Solvent.h.

12.50.2.2 WField `template<int D>`

`typedef RDField<D> Pscf::Pspg::Solvent< D >::WField`
Monomer chemical potential field.
Definition at line 38 of file Solvent.h.

12.50.3 Constructor & Destructor Documentation

12.50.3.1 Solvent() template<int D>

`Pscf::Pspg::Solvent< D >::Solvent ( ) [inline]`

Constructor.

Definition at line 43 of file Solvent.h.

12.50.3.2 ∼Solvent() template<int D>

`Pscf::Pspg::Solvent< D >::∼Solvent ( ) [inline]`

Constructor.

Definition at line 50 of file Solvent.h.

12.50.4 Member Function Documentation

12.50.4.1 compute() template<int D>

```
virtual void Pscf::Pspg::Solvent< D >::compute (
            WField const & wField ) [inline], [virtual]
```

Compute monomer concentration field and phi and/or mu.

Pure virtual function: Must be implemented by subclasses. Upon return, concentration field, phi and mu are all set.

Parameters

wField	monomer chemical potential field.

Definition at line 62 of file Solvent.h.

12.50.4.2 concentration() template<int D>

`const CField& Pscf::Pspg::Solvent< D >::concentration ( ) const [inline]`

Get monomer concentration field for this solvent.

Definition at line 67 of file Solvent.h.

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

- Solvent.h

12.51 Pscf::Pspg::WaveList< D > Class Template Reference

12.51.1 Detailed Description

template<int D>
class Pscf::Pspg::WaveList< D >

Definition at line 27 of file WaveList.h.

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

- WaveList.h
- WaveList.tpp

12.52 Pscf::RealVec< D, T > Class Template Reference

A RealVec<D, T> is D-component vector with elements of floating type T.

`#include <RealVec.h>`

Inheritance diagram for Pscf::RealVec< D, T >:

classPscf_1_1RealVec-eps-converted-to.pdf

Constructors

- static const int Width = 25

  Width of field per Cartesian coordinate in stream IO.
- static const int Precision = 17

  Precision in stream IO of RealVec<D, T> coordinates.
- RealVec ()

  Default constructor.
- RealVec (const RealVec< D, T > &v)

  Copy constructor.
- RealVec (T const *v)

  Construct from C array.
- RealVec (T s)

  Constructor, initialize all elements to a scalar value.

Additional Inherited Members

12.52.1 Detailed Description

template<int D, typename T = double>
class Pscf::RealVec< D, T >

A RealVec<D, T> is D-component vector with elements of floating type T.
Default of type T is T = double.
Definition at line 27 of file RealVec.h.

12.52.2 Constructor & Destructor Documentation

12.52.2.1 RealVec() `[1/4]` `template<int D, typename T = double>`

`Pscf::RealVec< D, T >::RealVec ( ) [inline]`

Default constructor.
Definition at line 38 of file RealVec.h.

12.52.2.2 RealVec() `[2/4]` `template<int D, typename T = double>`

`Pscf::RealVec< D, T >::RealVec (`
`const RealVec< D, T > & v ) [inline]`

Copy constructor.

Parameters

v	RealVec<D, T> to be copied

Definition at line 47 of file RealVec.h.

12.52.2.3 RealVec() [3/4] template<int D, typename T = double>

```
Pscf::RealVec< D, T >::RealVec (
            T const * v ) [inline]
```

Construct from C array.

Parameters

v	C array to be copied

Definition at line 56 of file RealVec.h.

12.52.2.4 RealVec() [4/4] template<int D, typename T = double>

```
Pscf::RealVec< D, T >::RealVec (
            T s ) [inline], [explicit]
```

Constructor, initialize all elements to a scalar value.

Parameters

s	scalar initial value for all elements.

Definition at line 65 of file RealVec.h.

12.52.3 Member Data Documentation

12.52.3.1 Width template<int D, typename T = double>

```
const int Pscf::RealVec< D, T >::Width = 25 [static]
```

Width of field per Cartesian coordinate in stream IO.

Definition at line 70 of file RealVec.h.

12.52.3.2 Precision template<int D, typename T = double>

```
const int Pscf::RealVec< D, T >::Precision = 17 [static]
```

Precision in stream IO of RealVec<D, T> coordinates.

Definition at line 73 of file RealVec.h.

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

- RealVec.h

12.53 Pscf::SolventTmpl< TP > Class Template Reference

Template for a class representing a solvent species.

```
#include <SolventTmpl.h>
```

Inheritance diagram for Pscf::SolventTmpl< TP >:

classPscf_1_1SolventTmpl-eps-converted-to.pdf

Public Types

- typedef TP::CField CField

    Monomer concentration field.
- typedef TP::WField WField

    Monomer chemical potential field.

Public Member Functions

- SolventTmpl ()

    Constructor.
- ∼SolventTmpl ()

    Constructor.
- void setMonomerId (int monomerId)

    Set the monomer id.
- void setSize (double size)

    Set the size of this block.
- virtual void compute (WField const &wField)=0

    Compute monomer concentration field and phi and/or mu.

Accessors (getters)

- CField **concentration_**
- int monomerId () const

    Get the monomer type id.
- double size () const

    Get the size (number of monomers) in this block.
- const CField & concentration () const

    Get monomer concentration field for this solvent.

Additional Inherited Members

12.53.1 Detailed Description

template<class TP>
class Pscf::SolventTmpl< TP >

Template for a class representing a solvent species.
Template argument TP is a propagator class. This is only used to define the data types for concentration and chemical potential fields.
Definition at line 29 of file SolventTmpl.h.

12.53.2 Member Typedef Documentation

12.53.2.1 CField `template<class TP >`

`typedef TP::CField Pscf::SolventTmpl< TP >::CField`

Monomer concentration field.

Definition at line 36 of file SolventTmpl.h.

12.53.2.2 WField `template<class TP >`

`typedef TP::WField Pscf::SolventTmpl< TP >::WField`

Monomer chemical potential field.

Definition at line 41 of file SolventTmpl.h.

12.53.3 Constructor & Destructor Documentation

12.53.3.1 SolventTmpl() `template<class TP >`

`Pscf::SolventTmpl< TP >::SolventTmpl ( ) [inline]`

Constructor.

Definition at line 46 of file SolventTmpl.h.

12.53.3.2 ∼SolventTmpl() `template<class TP >`

`Pscf::SolventTmpl< TP >::∼SolventTmpl ( ) [inline]`

Constructor.

Definition at line 52 of file SolventTmpl.h.

12.53.4 Member Function Documentation

12.53.4.1 setMonomerId() `template<class TP >`

```
void Pscf::SolventTmpl< TP >::setMonomerId (
            int monomerId )
```

Set the monomer id.

Parameters

monomer↩ Id	integer id of monomer type (>=0)

12.53.4.2 setSize() `template<class TP >`

```
void Pscf::SolventTmpl< TP >::setSize (
            double size )
```

Set the size of this block.

The ``size'' is steric volume / reference volume.

Parameters

size	block size

12.53.4.3 compute() `template<class TP >`

`virtual void Pscf::SolventTmpl< TP >::compute ( WField const & wField ) [pure virtual]`

Compute monomer concentration field and phi and/or mu.

Pure virtual function: Must be implemented by subclasses. Upon return, concentration field, phi and mu are all set.

Parameters

wField	monomer chemical potential field.

12.53.4.4 monomerId() `template<class TP >`

`int Pscf::SolventTmpl< TP >::monomerId [inline]`

Get the monomer type id.

Definition at line 133 of file SolventTmpl.h.

12.53.4.5 size() `template<class TP >`

`double Pscf::SolventTmpl< TP >::size [inline]`

Get the size (number of monomers) in this block.

Definition at line 140 of file SolventTmpl.h.

12.53.4.6 concentration() `template<class TP >`

`const SolventTmpl< TP >::CField & Pscf::SolventTmpl< TP >::concentration`

Get monomer concentration field for this solvent.

Definition at line 147 of file SolventTmpl.h.

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

- SolventTmpl.h

12.54 Pscf::SpaceGroup< D > Class Template Reference

Crystallographic space group.

`#include <SpaceGroup.h>`

Inheritance diagram for Pscf::SpaceGroup< D >:

classPscf_1_1SpaceGroup-eps-converted-to.pdf

Public Member Functions

- bool hasInversionCenter (typename SpaceSymmetry< D >::Translation &center) const

Determines if this space group has an inversion center.

- void shiftOrigin (typename SpaceSymmetry< D >::Translation const &origin)

  Shift the origin of space used in the coordinate system.

- void checkMeshDimensions (IntVec< D > const &dimensions) const

  Check if input mesh dimensions are compatible with space group.

12.54.1 Detailed Description

template<int D>
class Pscf::SpaceGroup< D >

Crystallographic space group.
Definition at line 29 of file SpaceGroup.h.

12.54.2 Member Function Documentation

12.54.2.1 hasInversionCenter() `template<int D>`

```
bool Pscf::SpaceGroup< D >::hasInversionCenter (
            typename SpaceSymmetry< D >::Translation & center ) const
```

Determines if this space group has an inversion center.
Returns true if an inversion center exists, and false otherwise. If an inversion center exists, its location is returned as the output value of output argument "center".

Parameters

center	location of inversion center, if any (output)

Definition at line 24 of file SpaceGroup.tpp.

12.54.2.2 shiftOrigin() `template<int D>`

```
void Pscf::SpaceGroup< D >::shiftOrigin (
            typename SpaceSymmetry< D >::Translation const & origin )
```

Shift the origin of space used in the coordinate system.
This function modifies each symmetry elements in the group so as to refer to an equivalent symmetry defined using a new coordinate system with a shifted origin. The argument gives the coordinates of the origin of the new coordinate system as defined in the old coordinate system.

Parameters

origin	location of origin of the new coordinate system

Definition at line 52 of file SpaceGroup.tpp.

12.54.2.3 checkMeshDimensions() `template<int D>`

```
void Pscf::SpaceGroup< D >::checkMeshDimensions (
            IntVec< D > const & dimensions ) const
```

Check if input mesh dimensions are compatible with space group.
This function checks if a mesh with the specified dimensions is invariant under all operations of this space group, i.e., whether each crystal symmetry operation maps the position of every node of the mesh onto the position of another node.

It is only possible define how a symmetry operation transforms a function that is defined only on the nodes of mesh if the mesh is invariant under the symmetry operation, in this sense. An invariant mesh must thus be used necessary to describe a function whose values on the mesh nodes are invariant under all operations in the space group.

If the mesh is not invariant under all operations of the space group, an explanatory error message is printed and an Exception is thrown to halt execution.

The mesh for a unit cell within a Bravais lattice is assumed to be a regular orthogonal mesh in a space of reduced coordinates, which are the components of position defined using a Bravais basis (i.e., a basis of Bravais lattice basis vectors). Each element of the dimensions vector is equal to the number of grid points along a direction corresponding to a Bravais lattice vector. A Bravais basis is also used to define elements of the matrix representation of the point group operation and the translation vector in the representation of a crystal symmetry operation as an instance of class Pscf::SpaceSymmetry<D>.

Parameters

dimensions	vector of mesh dimensions

Definition at line 64 of file SpaceGroup.tpp.

References Util::Log::file(), UTIL_CHECK, and UTIL_THROW.

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

- SpaceGroup.h
- SpaceGroup.tpp

12.55 Pscf::SpaceSymmetry< D > Class Template Reference

A SpaceSymmetry represents a crystallographic space group symmetry.

`#include <SpaceSymmetry.h>`

Public Types

- typedef FMatrix< int, D, D > Rotation

  Typedef for matrix used to represent point group operation.
- typedef FArray< Rational, D > Translation

  Typedef for vector used to represent fractional translation.

Public Member Functions

- SpaceSymmetry ()

  Default constructor.
- SpaceSymmetry (const SpaceSymmetry< D > &other)

  Copy constructor.
- SpaceSymmetry< D > & operator= (const SpaceSymmetry< D > &other)

  Assignment operator.
- void normalize ()

  Shift components of translation to [0,1).
- SpaceSymmetry< D > inverse () const

  Compute and return the inverse of this symmetry element.
- SpaceSymmetry< D >::Rotation inverseRotation () const

  Compute and return the inverse of the rotation matrix.
- int determinant () const

  Compute and return the determinant of the rotation matrix.
- int & R (int i, int j)

Return an element of the matrix by reference.

- int R (int i, int j) const

  Return an element of the matrix by value.

- Rational & t (int i)

  Return a component of the translation by reference.

- Rational t (int i) const

  Return an element of the translation by value.

Static Public Member Functions

- static const SpaceSymmetry< D > & identity ()

  Return the identity element.

Friends

- bool operator== (const SpaceSymmetry< D > &A, const SpaceSymmetry< D > &B)

  Are two SpaceSymmetry objects equivalent?

- bool operator!= (const SpaceSymmetry< D > &A, const SpaceSymmetry< D > &B)

  Are two SpaceSymmetry objects not equivalent?

- SpaceSymmetry< D > operator* (const SpaceSymmetry< D > &A, const SpaceSymmetry< D > &B)

  *Return the product A*B of two symmetry objects.*

- IntVec< D > operator* (const IntVec< D > &V, const SpaceSymmetry< D > &S)

  *Return the IntVec<D> product V*S of an IntVec<D> and a rotation matrix.*

- IntVec< D > operator* (const SpaceSymmetry< D > &S, const IntVec< D > &V)

  *Return the IntVec<D> product S*V of a rotation matrix and an IntVec<D>.*

- std::ostream & operator<< (std::ostream &out, const SpaceSymmetry< D > &A)

  Output stream inserter for a SpaceSymmetry<D>

- std::istream & operator>> (std::istream &in, SpaceSymmetry< D > &A)

  Input stream extractor for a SpaceSymmetry<D>

12.55.1 Detailed Description

template<int D>
class Pscf::SpaceSymmetry< D >

A SpaceSymmetry represents a crystallographic space group symmetry.

Crystallographic space group symmetry operation combines a point group operation (e.g., 2, 3, and 4 fold rotations about axes, reflections, or inversion) with possible translations by a fraction of a unit cell.

Both the rotation matrix R and the translation t are represented using a basis of Bravais lattice basis vectors. Because Bravais basis vectors must map onto other lattice vectors, this implies that elements of all elements of the rotation matrix must be integers. To guarantee that the inverse of the rotation matrix is also a matrix of integers, we require that the determinant of the rotation matrix must be +1 or -1. The translation vector is represented by a vector of D rational numbers (i.e., fractions) of the form n/m with m = 2, 3, or 4 and n < m.

The basis used to describe a crytallographic group may be either a primitive or non-primitive unit cell. Thus, for example, the space group of a bcc crystal may be expressed either using a basis of 3 three orthogonal simple cubic unit vectors, with a translation t = (1/2, 1/2, 1/2), or as a point group using a set of three non-orthogonal basis vectors for the primitive unit cell.

Definition at line 23 of file SpaceSymmetry.h.

12.55.2 Member Typedef Documentation

12.55.2.1 Rotation `template<int D>`

`typedef FMatrix<int, D, D> Pscf::SpaceSymmetry< D >::Rotation`

Typedef for matrix used to represent point group operation.

Definition at line 143 of file SpaceSymmetry.h.

12.55.2.2 Translation `template<int D>`

`typedef FArray<Rational, D> Pscf::SpaceSymmetry< D >::Translation`

Typedef for vector used to represent fractional translation.

Definition at line 146 of file SpaceSymmetry.h.

12.55.3 Constructor & Destructor Documentation

12.55.3.1 SpaceSymmetry() [1/2] `template<int D>`

`Pscf::SpaceSymmetry< D >::SpaceSymmetry`

Default constructor.

All elements of the rotation matrix are initialized to zero.

Definition at line 22 of file SpaceSymmetry.tpp.

12.55.3.2 SpaceSymmetry() [2/2] `template<int D>`

```
Pscf::SpaceSymmetry< D >::SpaceSymmetry (
            const SpaceSymmetry< D > & other )
```

Copy constructor.

Definition at line 43 of file SpaceSymmetry.tpp.

12.55.4 Member Function Documentation

12.55.4.1 operator=() `template<int D>`

```
SpaceSymmetry< D > & Pscf::SpaceSymmetry< D >::operator= (
            const SpaceSymmetry< D > & other )
```

Assignment operator.

Definition at line 60 of file SpaceSymmetry.tpp.

12.55.4.2 normalize() `template<int D>`

`void Pscf::SpaceSymmetry< D >::normalize`

Shift components of translation to [0,1).

Definition at line 79 of file SpaceSymmetry.tpp.

References UTIL_ASSERT.

Referenced by Pscf::SpaceSymmetry< D >::inverse(), and Pscf::operator*().

12.55.4.3 inverse() `template<int D>`

`SpaceSymmetry< D > Pscf::SpaceSymmetry< D >::inverse`

Compute and return the inverse of this symmetry element.

Definition at line 124 of file SpaceSymmetry.tpp.

References Pscf::SpaceSymmetry< D >::normalize().

12.55.4.4 inverseRotation() `template<int D>`

`SpaceSymmetry<D>::Rotation Pscf::SpaceSymmetry< D >::inverseRotation ( ) const`

Compute and return the inverse of the rotation matrix.

12.55.4.5 determinant() `template<int D>`

`int Pscf::SpaceSymmetry< D >::determinant ( ) const`

Compute and return the determinant of the rotation matrix.

12.55.4.6 R() [1/2] `template<int D>`

```
int & Pscf::SpaceSymmetry< D >::R (
            int i,
            int j )  [inline]
```

Return an element of the matrix by reference.

Parameters

i	1st (row) index
j	2nd (column) index

Definition at line 313 of file SpaceSymmetry.h.

12.55.4.7 R() [2/2] `template<int D>`

```
int Pscf::SpaceSymmetry< D >::R (
            int i,
            int j ) const  [inline]
```

Return an element of the matrix by value.

Parameters

i	1st (row) index
j	2nd (column) index

Definition at line 321 of file SpaceSymmetry.h.

12.55.4.8 t() [1/2] `template<int D>`

```
Rational & Pscf::SpaceSymmetry< D >::t (
            int i )  [inline]
```

Return a component of the translation by reference.

Parameters

i	component index

Definition at line 329 of file SpaceSymmetry.h.

12.55.4.9 t() [2/2] `template<int D>`

```
Rational Pscf::SpaceSymmetry< D >::t (
```

```
int i ) const [inline]
```

Return an element of the translation by value.

Parameters

i	component index

Definition at line 337 of file SpaceSymmetry.h.

12.55.4.10 identity() template<int D>

```
const SpaceSymmetry< D > & Pscf::SpaceSymmetry< D >::identity [inline], [static]
```

Return the identity element.

Definition at line 345 of file SpaceSymmetry.h.

12.55.5 Friends And Related Function Documentation

12.55.5.1 operator== template<int D>

```
bool operator== (
        const SpaceSymmetry< D > & A,
        const SpaceSymmetry< D > & B ) [friend]
```

Are two SpaceSymmetry objects equivalent?

Parameters

A	first symmetry
B	second symmetry

Returns

True if A == B, false otherwise

Definition at line 357 of file SpaceSymmetry.h.

12.55.5.2 operator"!= template<int D>

```
bool operator!= (
        const SpaceSymmetry< D > & A,
        const SpaceSymmetry< D > & B ) [friend]
```

Are two SpaceSymmetry objects not equivalent?

Parameters

A	first symmetry
B	second symmetry

Returns

True if A != B, false otherwise

Definition at line 305 of file SpaceSymmetry.h.

12.55.5.3 operator* [1/3] template<int D>

```
SpaceSymmetry<D> operator* (
            const SpaceSymmetry< D > & A,
            const SpaceSymmetry< D > & B ) [friend]
```

Return the product A*B of two symmetry objects.

Parameters

A	first symmetry
B	second symmetry

Returns

product A*B

Definition at line 378 of file SpaceSymmetry.h.

12.55.5.4 operator* [2/3] template<int D>

```
IntVec<D> operator* (
            const IntVec< D > & V,
            const SpaceSymmetry< D > & S ) [friend]
```

Return the IntVec< D > product V*S of an IntVec< D > and a rotation matrix.
The product is defined to be the matrix product of the integer vector and the space group rotation matrix S.R * V.

Parameters

V	integer vector
S	symmetry operation

Returns

product V*S

Definition at line 428 of file SpaceSymmetry.h.

12.55.5.5 operator* [3/3] template<int D>

```
IntVec<D> operator* (
            const SpaceSymmetry< D > & S,
            const IntVec< D > & V ) [friend]
```

Return the IntVec< D > product S*V of a rotation matrix and an IntVec< D >.
The product is defined to be the matrix product of the rotation matrix and the integer vector S.R * V.

Parameters

S	symmetry operation
V	integer vector

Returns

product S∗V

Definition at line 411 of file SpaceSymmetry.h.

12.55.5.6 operator<< template<int D>

```
std::ostream& operator<< (
            std::ostream & out,
            const SpaceSymmetry< D > & A ) [friend]
```

Output stream inserter for a SpaceSymmetry<D>

Parameters

out	output stream
A	SpaceSymmetry<D> object to be output

Returns

modified output stream

Definition at line 445 of file SpaceSymmetry.h.

12.55.5.7 operator>> template<int D>

```
std::istream& operator>> (
            std::istream & in,
            SpaceSymmetry< D > & A ) [friend]
```

Input stream extractor for a SpaceSymmetry<D>

Parameters

in	input stream
A	SpaceSymmetry<D> object to be input

Returns

modified input stream

Definition at line 465 of file SpaceSymmetry.h.
The documentation for this class was generated from the following files:

- SpaceSymmetry.h
- SpaceSymmetry.tpp

12.56 Pscf::Species Class Reference

Base class for a molecular species (polymer or solvent).

```
#include <Species.h>
```

Inheritance diagram for Pscf::Species:

classPscf_1_1Species-eps-converted-to.pdf

Public Types

- enum Ensemble

    Statistical ensemble for number of molecules.

Public Member Functions

- Species ()

    Default constructor.
- double phi () const

    Get overall volume fraction for this species.
- double mu () const

    Get chemical potential for this species (units kT=1).
- double q () const

    Get molecular partition function for this species.
- Ensemble ensemble ()

    Get statistical ensemble for this species (open or closed).

Protected Attributes

- double phi_

    Volume fraction, set by either setPhi or compute function.
- double mu_

    Chemical potential, set by either setPhi or compute function.
- double q_

    Partition function, set by compute function.
- Ensemble ensemble_

    Statistical ensemble for this species (open or closed).
- bool isComputed_

    Set true by upon return by compute() and set false by clear().

12.56.1 Detailed Description

Base class for a molecular species (polymer or solvent).
Definition at line 20 of file Species.h.

12.56.2 Member Enumeration Documentation

12.56.2.1 Ensemble `enum Pscf::Species::Ensemble`

Statistical ensemble for number of molecules.
Definition at line 27 of file Species.h.

12.56.3 Constructor & Destructor Documentation

12.56.3.1 Species() `Pscf::Species::Species ( )`

Default constructor.
Definition at line 15 of file Species.cpp.

12.56.4 Member Function Documentation

12.56.4.1 phi() `double Pscf::Species::phi ( ) const [inline]`

Get overall volume fraction for this species.
Definition at line 86 of file Species.h.
References phi_.
Referenced by Pscf::Pspg::Continuous::System< D >::computeFreeEnergy().

12.56.4.2 mu() `double Pscf::Species::mu ( ) const [inline]`

Get chemical potential for this species (units kT=1).
Definition at line 92 of file Species.h.
References mu_.
Referenced by Pscf::Pspg::Continuous::System< D >::computeFreeEnergy().

12.56.4.3 q() `double Pscf::Species::q ( ) const`

Get molecular partition function for this species.

12.56.4.4 ensemble() `Species::Ensemble Pscf::Species::ensemble ( ) [inline]`

Get statistical ensemble for this species (open or closed).
Definition at line 98 of file Species.h.
References ensemble_.

12.56.5 Member Data Documentation

12.56.5.1 phi_ `double Pscf::Species::phi_ [protected]`

Volume fraction, set by either setPhi or compute function.
Definition at line 59 of file Species.h.
Referenced by phi().

12.56.5.2 mu_ `double Pscf::Species::mu_ [protected]`

Chemical potential, set by either setPhi or compute function.
Definition at line 64 of file Species.h.
Referenced by mu().

12.56.5.3 q_ `double Pscf::Species::q_ [protected]`

Partition function, set by compute function.
Definition at line 69 of file Species.h.

12.56.5.4 ensemble_ `Ensemble Pscf::Species::ensemble_ [protected]`

Statistical ensemble for this species (open or closed).
Definition at line 74 of file Species.h.
Referenced by ensemble().

12.56.5.5 isComputed_ `bool Pscf::Species::isComputed_ [protected]`

Set true by upon return by compute() and set false by clear().
Definition at line 79 of file Species.h.
The documentation for this class was generated from the following files:

- Species.h
- Species.cpp

12.57 Pscf::SymmetryGroup< Symmetry > Class Template Reference

Class template for a group of elements.
`#include <SymmetryGroup.h>`

Public Member Functions

- SymmetryGroup ()
  Default constructor.
- SymmetryGroup (const SymmetryGroup< Symmetry > &other)
  Copy constructor.
- ~SymmetryGroup ()
  Destructor.
- bool add (Symmetry &symmetry)
  Add a new element to the group.
- void makeCompleteGroup ()
  Generate a complete group from the current elements.
- void clear ()
  Remove all elements except the identity.
- const Symmetry * find (const Symmetry &symmetry) const
  Find a symmetry within a group.
- const Symmetry & identity () const
  Return a reference to the identity element.
- int size () const
  Return number of elements in group (i.e., the order of the group).
- SymmetryGroup< Symmetry > & operator= (const SymmetryGroup< Symmetry > &other)
  Assignment operator.
- Symmetry & operator[] (int i)
  Element access operator (by reference).
- const Symmetry & operator[] (int i) const
  Element access operator (by reference).
- bool isValid () const
  Return true if valid complete group, or throw an Exception.

12.57.1 Detailed Description

template<class Symmetry>
class Pscf::SymmetryGroup< Symmetry >

Class template for a group of elements.
This is written as a template to allow the creation of groups that use different types of objects to represent symmetry elements. The simplest distinction is between point groups and full space groups.
The algorithm requires only the template parameter class Symmetry satisfy the following requirements:

1) A Symmetry must be default constructible. 2) An operator $*$ is provided to represent element multiplication. 3) Operators == and != are provided to represent equality & inequality. 4) A method Symmetry::inverse() must return the inverse of a Symmetry. 5) A static method Symmetry::identity() must return the identity.
Definition at line 36 of file SymmetryGroup.h.

12.57.2 Constructor & Destructor Documentation

12.57.2.1 SymmetryGroup() [1/2] template<class Symmetry >

`Pscf::SymmetryGroup< Symmetry >::SymmetryGroup`

Default constructor.
After construction, the group contains only the identity element.
Definition at line 25 of file SymmetryGroup.tpp.

12.57.2.2 SymmetryGroup() [2/2] template<class Symmetry >

```
Pscf::SymmetryGroup< Symmetry >::SymmetryGroup (
        const SymmetryGroup< Symmetry > & other )
```

Copy constructor.
Definition at line 35 of file SymmetryGroup.tpp.

12.57.2.3 ∼SymmetryGroup() template<class Symmetry >

`Pscf::SymmetryGroup< Symmetry >::~SymmetryGroup`

Destructor.
Definition at line 48 of file SymmetryGroup.tpp.

12.57.3 Member Function Documentation

12.57.3.1 add() template<class Symmetry >

```
bool Pscf::SymmetryGroup< Symmetry >::add (
        Symmetry & symmetry )
```

Add a new element to the group.
Return false if the element was already present, true otherwise.

Parameters

symmetry	new symmetry element.

Returns

true if this is a new element, false if already present.

Definition at line 89 of file SymmetryGroup.tpp.

12.57.3.2 makeCompleteGroup() template<class Symmetry >

`void Pscf::SymmetryGroup< Symmetry >::makeCompleteGroup`

Generate a complete group from the current elements.
Definition at line 109 of file SymmetryGroup.tpp.

12.57.3.3 clear() template<class Symmetry >

void Pscf::SymmetryGroup< Symmetry >::clear

Remove all elements except the identity.

Return group to its state after default construction.

Definition at line 151 of file SymmetryGroup.tpp.

12.57.3.4 find() template<class Symmetry >

const Symmetry * Pscf::SymmetryGroup< Symmetry >::find (
    const Symmetry & *symmetry*) const

Find a symmetry within a group.

Return a pointer to a symmetry if it is in the group, or a null pointer if it is not.

Definition at line 74 of file SymmetryGroup.tpp.

Referenced by pscfpp.MakeMaker.MakeMaker::find().

12.57.3.5 identity() template<class Symmetry >

const Symmetry & Pscf::SymmetryGroup< Symmetry >::identity [inline]

Return a reference to the identity element.

Definition at line 143 of file SymmetryGroup.h.

Referenced by Pscf::SymmetryGroup< SpaceSymmetry< D > >::operator=(), and Pscf::SymmetryGroup< Space↩ Symmetry< D > >::SymmetryGroup().

12.57.3.6 size() template<class Symmetry >

int Pscf::SymmetryGroup< Symmetry >::size [inline]

Return number of elements in group (i.e., the order of the group).

Definition at line 135 of file SymmetryGroup.h.

Referenced by Pscf::SymmetryGroup< SpaceSymmetry< D > >::operator=(), and Pscf::SymmetryGroup< Space↩ Symmetry< D > >::SymmetryGroup().

12.57.3.7 operator=() template<class Symmetry >

SymmetryGroup< Symmetry > & Pscf::SymmetryGroup< Symmetry >::operator= (
    const SymmetryGroup< Symmetry > & *other*)

Assignment operator.

Definition at line 56 of file SymmetryGroup.tpp.

12.57.3.8 operator[]() [1/2] template<class Symmetry >

Symmetry & Pscf::SymmetryGroup< Symmetry >::operator[] (
    int *i*) [inline]

Element access operator (by reference).

Definition at line 151 of file SymmetryGroup.h.

12.57.3.9 operator[]() [2/2] template<class Symmetry >

const Symmetry & Pscf::SymmetryGroup< Symmetry >::operator[] (
    int *i*) const [inline]

Element access operator (by reference).

Definition at line 159 of file SymmetryGroup.h.

12.57.3.10 isValid() template<class Symmetry > bool Pscf::SymmetryGroup< Symmetry >::isValid

Return true if valid complete group, or throw an Exception.
Definition at line 162 of file SymmetryGroup.tpp.
The documentation for this class was generated from the following files:

- SymmetryGroup.h
- SymmetryGroup.tpp

12.58 Pscf::TridiagonalSolver Class Reference

Solver for Ax=b with tridiagonal matrix A.

```
#include <TridiagonalSolver.h>
```

Public Member Functions

- TridiagonalSolver ()

    Constructor.
- ∼TridiagonalSolver ()

    Destructor.
- void allocate (int n)

    Allocate memory.
- void computeLU (const DArray< double > &d, const DArray< double > &u)

    Compute LU decomposition of a symmetric tridiagonal matrix.
- void computeLU (const DArray< double > &d, const DArray< double > &u, const DArray< double > &l)

    Compute LU decomposition of a general tridiagonal matrix.
- void multiply (const DArray< double > &b, DArray< double > &x)

    Evaluate product Ab = x for known b to compute x.
- void solve (const DArray< double > &b, DArray< double > &x)

    Solve Ax = b for known b to compute x.

12.58.1 Detailed Description

Solver for Ax=b with tridiagonal matrix A.
Definition at line 27 of file TridiagonalSolver.h.

12.58.2 Constructor & Destructor Documentation

12.58.2.1 TridiagonalSolver() Pscf::TridiagonalSolver::TridiagonalSolver ( )

Constructor.
Definition at line 19 of file TridiagonalSolver.cpp.

12.58.2.2 ∼TridiagonalSolver() Pscf::TridiagonalSolver::∼TridiagonalSolver ( )

Destructor.
Definition at line 25 of file TridiagonalSolver.cpp.

12.58.3 Member Function Documentation

12.58.3.1 allocate()

```
void Pscf::TridiagonalSolver::allocate (
    int n )
```

Allocate memory.

Parameters

n	dimension of n x n square array.

Definition at line 31 of file TridiagonalSolver.cpp.
References Util::DArray< Data >::allocate().

12.58.3.2 computeLU() [1/2]

```
void Pscf::TridiagonalSolver::computeLU (
    const DArray< double > & d,
    const DArray< double > & u )
```

Compute LU decomposition of a symmetric tridiagonal matrix.

Parameters

d	diagonal elements of n x n matrix matrix (0,..,n-1)
u	upper off-diagonal elements (0,..,n-2)

Definition at line 43 of file TridiagonalSolver.cpp.

12.58.3.3 computeLU() [2/2]

```
void Pscf::TridiagonalSolver::computeLU (
    const DArray< double > & d,
    const DArray< double > & u,
    const DArray< double > & l )
```

Compute LU decomposition of a general tridiagonal matrix.

Parameters

d	diagonal elements of n x n matrix matrix (0,..,n-1)
u	upper off-diagonal elements (0,..,n-2)
l	lower off-diagonal elements (0,..,n-2)

Definition at line 59 of file TridiagonalSolver.cpp.

12.58.3.4 multiply()

```
void Pscf::TridiagonalSolver::multiply (
    const DArray< double > & b,
    DArray< double > & x )
```

Evaluate product Ab = x for known b to compute x.

Parameters

b	known vector to be multiplied (input)
x	result of multiplication Ab = x (output)

Definition at line 103 of file TridiagonalSolver.cpp.

12.58.3.5 solve()

```
void Pscf::TridiagonalSolver::solve (
    const DArray< double > & b,
    DArray< double > & x )
```

Solve Ax = b for known b to compute x.

Parameters

b	known vector on RHS (input)
x	unknown solution vector of Ax = b (output)

Definition at line 121 of file TridiagonalSolver.cpp.
The documentation for this class was generated from the following files:

- TridiagonalSolver.h
- TridiagonalSolver.cpp

12.59 Pscf::TWave< D > Struct Template Reference

Simple wave struct for use within Basis construction.
`#include <TWave.h>`

12.59.1 Detailed Description

template<int D>
struct Pscf::TWave< D >

Simple wave struct for use within Basis construction.
Definition at line 22 of file TWave.h.
The documentation for this struct was generated from the following file:

- TWave.h

12.60 Pscf::TWaveBzComp< D > Struct Template Reference

Comparator for TWave objects, based on TWave::indicesBz.
`#include <TWave.h>`

Public Member Functions

- bool operator() (const TWave< D > &a, const TWave< D > &b) const

  Function (a, b) returns true iff a.indicesBz > b.indicesBz.

12.60.1 Detailed Description

template<int D>
struct Pscf::TWaveBzComp< D >

Comparator for TWave objects, based on TWave::indicesBz.
Used to sort in descending order of Bz (Brillouin zone) indices.
Definition at line 74 of file TWave.h.

12.60.2 Member Function Documentation

12.60.2.1 operator()() template<int D>

```
bool Pscf::TWaveBzComp< D >::operator() (
            const TWave< D > & a,
            const TWave< D > & b ) const [inline]
```

Function (a, b) returns true iff a.indicesBz > b.indicesBz.
Definition at line 79 of file TWave.h.
The documentation for this struct was generated from the following file:

- TWave.h

12.61 Pscf::TWaveDftComp< D > Struct Template Reference

Comparator for TWave objects, based on TWave::indicesDft.
`#include <TWave.h>`

Public Member Functions

- bool operator() (const TWave< D > &a, const TWave< D > &b) const

    Function (a, b) returns true iff a.indicesDft < b.indicesDft.

12.61.1 Detailed Description

template<int D>
struct Pscf::TWaveDftComp< D >

Comparator for TWave objects, based on TWave::indicesDft.
Used to sort set of unique waves in ascending order of dft indices.
Definition at line 56 of file TWave.h.

12.61.2 Member Function Documentation

12.61.2.1 operator()() template<int D>

```
bool Pscf::TWaveDftComp< D >::operator() (
            const TWave< D > & a,
            const TWave< D > & b ) const [inline]
```

Function (a, b) returns true iff a.indicesDft < b.indicesDft.
Definition at line 61 of file TWave.h.
The documentation for this struct was generated from the following file:

- TWave.h

12.62 Pscf::TWaveNormComp< D > Struct Template Reference

Comparator for TWave objects, based on TWave::sqNorm.
`#include <TWave.h>`

Public Member Functions

- bool operator() (const TWave< D > &a, const TWave< D > &b) const

    Function (a, b) returns true iff a.sqNorm < b.sqNorm.

12.62.1 Detailed Description

template<int D>
struct Pscf::TWaveNormComp< D >

Comparator for TWave objects, based on TWave::sqNorm.
Used to sort in ascending order of wavevector norm.
Definition at line 38 of file TWave.h.

12.62.2 Member Function Documentation

12.62.2.1 operator()() `template<int D>`

```
bool Pscf::TWaveNormComp< D >::operator() (
            const TWave< D > & a,
            const TWave< D > & b ) const  [inline]
```

Function (a, b) returns true iff a.sqNorm < b.sqNorm.
Definition at line 43 of file TWave.h.
The documentation for this struct was generated from the following file:

- TWave.h

12.63 Pscf::UnitCell< D > Class Template Reference

Base template for UnitCell<D> classes, D=1, 2 or 3.
`#include <UnitCell.h>`
Inheritance diagram for Pscf::UnitCell< D >:

classPscf_1_1UnitCell-eps-converted-to.pdf

Additional Inherited Members

12.63.1 Detailed Description

template<int D>
class Pscf::UnitCell< D >

Base template for UnitCell<D> classes, D=1, 2 or 3.
Explicit specializations are provided for D=1, 2, and 3. In each case, class UnitCell<D> is derived from UnitCell↩Base<D>, and defines an enumeration named LatticeSystem of the types of Bravais lattice systems in D-dimensional space.
Definition at line 31 of file UnitCell.h.
The documentation for this class was generated from the following file:

- UnitCell.h

12.64 Pscf::UnitCell< 1 > Class Reference

1D crystal unit cell.
`#include <UnitCell.h>`
Inheritance diagram for Pscf::UnitCell< 1 >:

classPscf_1_1UnitCell_3_011_01_4-eps-converted-to.pdf

Public Types

- enum LatticeSystem

  Enumeration of 1D lattice system types.

Public Member Functions

- UnitCell ()

  Constructor.

Friends

- template<int D>
  std::istream & operator>> (std::istream &, UnitCell< D > &)

  istream input extractor for a UnitCell<D>.
- template<int D>
  std::ostream & operator<< (std::ostream &, UnitCell< D > const &)

  ostream output inserter for a UnitCell<D>.
- template<class Archive , int D>
  void serialize (Archive &, UnitCell< D > &, const unsigned int)

  Serialize to/from an archive.
- template<int D>
  void readUnitCellHeader (std::istream &, UnitCell< D > &)

  Read UnitCell<D> from a field file header (fortran pscf format).
- template<int D>
  void writeUnitCellHeader (std::ostream &, UnitCell< D > const &)

  Write UnitCell<D> to a field file header (fortran pscf format).

Additional Inherited Members

12.64.1 Detailed Description

1D crystal unit cell.
Definition at line 105 of file UnitCell.h.

12.64.2 Member Enumeration Documentation

12.64.2.1 LatticeSystem `enum Pscf::UnitCell< 1 >::LatticeSystem`

Enumeration of 1D lattice system types.
Definition at line 112 of file UnitCell.h.

12.64.3 Constructor & Destructor Documentation

12.64.3.1 UnitCell() `Pscf::UnitCell< 1 >::UnitCell ( )`

Constructor.

Definition at line 19 of file UnitCell1.cpp.

12.64.4 Friends And Related Function Documentation

12.64.4.1 operator>> `template<int D>`

```
std::istream& operator>> (
            std::istream & in,
            UnitCell< D > & cell )  [friend]
```

istream input extractor for a UnitCell<D>.

Parameters

in	input stream
cell	UnitCell<D> to be read

Returns

modified input stream

Definition at line 21 of file UnitCell.tpp.

12.64.4.2 operator<< `template<int D>`

```
std::ostream& operator<< (
            std::ostream & out,
            UnitCell< D > const & cell )  [friend]
```

ostream output inserter for a UnitCell<D>.

Parameters

out	output stream
cell	UnitCell<D> to be written

Returns

modified output stream

Definition at line 34 of file UnitCell.tpp.

12.64.4.3 serialize `template<class Archive , int D>`

```
void serialize (
            Archive & ar,
            UnitCell< D > & cell,
            const unsigned int version )  [friend]
```

Serialize to/from an archive.

Parameters

ar	input or output archive

Parameters

cell	UnitCell<D> object to be serialized
version	archive version id

Definition at line 48 of file UnitCell.tpp.

12.64.4.4 readUnitCellHeader

```
template<int D>
void readUnitCellHeader (
            std::istream & in,
            UnitCell< D > & cell )  [friend]
```

Read UnitCell<D> from a field file header (fortran pscf format).

If the unit cell has a non-null lattice system on entry, the value read from file must match this existing value, or this function throws an exception. If the lattice system is null on entry, the lattice system value is read from file. In either case, unit cell parameters (dimensions and angles) are updated using values read from file.

Parameters

in	input stream
cell	UnitCell<D> to be read

Definition at line 59 of file UnitCell.tpp.

12.64.4.5 writeUnitCellHeader

```
template<int D>
void writeUnitCellHeader (
            std::ostream & out,
            UnitCell< D > const & cell )  [friend]
```

Write UnitCell<D> to a field file header (fortran pscf format).

Parameters

out	output stream
cell	UnitCell<D> to be written

Definition at line 88 of file UnitCell.tpp.

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

- UnitCell.h
- UnitCell1.cpp

12.65 Pscf::UnitCell< 2 > Class Reference

2D crystal unit cell.

`#include <UnitCell.h>`

Inheritance diagram for Pscf::UnitCell< 2 >:

classPscf_1_1UnitCell_3_012_01_4-eps-converted-to.pdf

Public Types

- enum LatticeSystem

  Enumeration of 2D lattice system types.

Public Member Functions

- UnitCell ()

  Constructor.

Friends

- template<int D>
  std::istream & operator>> (std::istream &, UnitCell< D > &)

  istream input extractor for a UnitCell<D>.
- template<int D>
  std::ostream & operator<< (std::ostream &, UnitCell< D > const &)

  ostream output inserter for a UnitCell<D>.
- template<class Archive , int D>
  void serialize (Archive &, UnitCell< D > &, const unsigned int)

  Serialize to/from an archive.
- template<int D>
  void readUnitCellHeader (std::istream &, UnitCell< D > &)

  Read UnitCell<D> from a field file header (fortran pscf format).
- template<int D>
  void writeUnitCellHeader (std::ostream &, UnitCell< D > const &)

  Write UnitCell<D> to a field file header (fortran pscf format).

Additional Inherited Members

12.65.1 Detailed Description

2D crystal unit cell.
Definition at line 180 of file UnitCell.h.

12.65.2 Member Enumeration Documentation

12.65.2.1 LatticeSystem `enum Pscf::UnitCell< 2 >::LatticeSystem`

Enumeration of 2D lattice system types.
Definition at line 187 of file UnitCell.h.

12.65.3 Constructor & Destructor Documentation

12.65.3.1 UnitCell() `Pscf::UnitCell< 2 >::UnitCell ( )`

Constructor.

Definition at line 19 of file UnitCell2.cpp.

12.65.4 Friends And Related Function Documentation

12.65.4.1 operator>> `template<int D>`

```
std::istream& operator>> (
            std::istream & in,
            UnitCell< D > & cell )  [friend]
```

istream input extractor for a UnitCell<D>.

Parameters

in	input stream
cell	UnitCell<D> to be read

Returns

modified input stream

Definition at line 21 of file UnitCell.tpp.

12.65.4.2 operator<< `template<int D>`

```
std::ostream& operator<< (
            std::ostream & out,
            UnitCell< D > const & cell )  [friend]
```

ostream output inserter for a UnitCell<D>.

Parameters

out	output stream
cell	UnitCell<D> to be written

Returns

modified output stream

Definition at line 34 of file UnitCell.tpp.

12.65.4.3 serialize `template<class Archive , int D>`

```
void serialize (
            Archive & ar,
            UnitCell< D > & cell,
            const unsigned int version )  [friend]
```

Serialize to/from an archive.

Parameters

ar	input or output archive

Parameters

cell	UnitCell<D> object to be serialized
version	archive version id

Definition at line 48 of file UnitCell.tpp.

12.65.4.4 readUnitCellHeader `template<int D>`

```
void readUnitCellHeader (
            std::istream & in,
            UnitCell< D > & cell )  [friend]
```

Read UnitCell<D> from a field file header (fortran pscf format).

If the unit cell has a non-null lattice system on entry, the value read from file must match this existing value, or this function throws an exception. If the lattice system is null on entry, the lattice system value is read from file. In either case, unit cell parameters (dimensions and angles) are updated using values read from file.

Parameters

in	input stream
cell	UnitCell<D> to be read

Definition at line 59 of file UnitCell.tpp.

12.65.4.5 writeUnitCellHeader `template<int D>`

```
void writeUnitCellHeader (
            std::ostream & out,
            UnitCell< D > const & cell )  [friend]
```

Write UnitCell<D> to a field file header (fortran pscf format).

Parameters

out	output stream
cell	UnitCell<D> to be written

Definition at line 88 of file UnitCell.tpp.

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

- UnitCell.h
- UnitCell2.cpp

12.66 Pscf::UnitCell< 3 > Class Reference

3D crystal unit cell.

`#include <UnitCell.h>`

Inheritance diagram for Pscf::UnitCell< 3 >:

classPscf_1_1UnitCell_3_013_01_4-eps-converted-to.pdf

Public Types

- enum LatticeSystem

  Enumeration of the 7 possible 3D Bravais lattice systems.

Public Member Functions

- UnitCell ()

  Constructor.

Friends

- template<int D>
  std::istream & operator>> (std::istream &, UnitCell< D > &)

  istream input extractor for a UnitCell<D>.
- template<int D>
  std::ostream & operator<< (std::ostream &, UnitCell< D > const &)

  ostream output inserter for a UnitCell<D>.
- template<class Archive , int D>
  void serialize (Archive &, UnitCell< D > &, const unsigned int)

  Serialize to/from an archive.
- template<int D>
  void readUnitCellHeader (std::istream &, UnitCell< D > &)

  Read UnitCell<D> from a field file header (fortran pscf format).
- template<int D>
  void writeUnitCellHeader (std::ostream &, UnitCell< D > const &)

  Write UnitCell<D> to a field file header (fortran pscf format).

Additional Inherited Members

12.66.1 Detailed Description

3D crystal unit cell.
Definition at line 257 of file UnitCell.h.

12.66.2 Member Enumeration Documentation

12.66.2.1 LatticeSystem `enum Pscf::UnitCell< 3 >::LatticeSystem`

Enumeration of the 7 possible 3D Bravais lattice systems.
Allowed non-null values are: Cubic, Tetragonal, Orthorhombic, Monoclinic, Triclinic, Rhombohedral, and Hexagonal.
Definition at line 267 of file UnitCell.h.

12.66.3 Constructor & Destructor Documentation

12.66.3.1 UnitCell() `Pscf::UnitCell< 3 >::UnitCell ( )`

Constructor.

Definition at line 19 of file UnitCell3.cpp.

12.66.4 Friends And Related Function Documentation

12.66.4.1 operator>> `template<int D>`

```
std::istream& operator>> (
            std::istream & in,
            UnitCell< D > & cell )  [friend]
```

istream input extractor for a UnitCell<D>.

Parameters

in	input stream
cell	UnitCell<D> to be read

Returns

modified input stream

Definition at line 21 of file UnitCell.tpp.

12.66.4.2 operator<< `template<int D>`

```
std::ostream& operator<< (
            std::ostream & out,
            UnitCell< D > const & cell )  [friend]
```

ostream output inserter for a UnitCell<D>.

Parameters

out	output stream
cell	UnitCell<D> to be written

Returns

modified output stream

Definition at line 34 of file UnitCell.tpp.

12.66.4.3 serialize `template<class Archive , int D>`

```
void serialize (
            Archive & ar,
            UnitCell< D > & cell,
            const unsigned int version )  [friend]
```

Serialize to/from an archive.

Parameters

ar	input or output archive

Parameters

cell	UnitCell<D> object to be serialized
version	archive version id

Definition at line 48 of file UnitCell.tpp.

12.66.4.4 readUnitCellHeader

```
template<int D>
void readUnitCellHeader (
        std::istream & in,
        UnitCell< D > & cell ) [friend]
```

Read UnitCell<D> from a field file header (fortran pscf format).

If the unit cell has a non-null lattice system on entry, the value read from file must match this existing value, or this function throws an exception. If the lattice system is null on entry, the lattice system value is read from file. In either case, unit cell parameters (dimensions and angles) are updated using values read from file.

Parameters

in	input stream
cell	UnitCell<D> to be read

Definition at line 59 of file UnitCell.tpp.

12.66.4.5 writeUnitCellHeader

```
template<int D>
void writeUnitCellHeader (
        std::ostream & out,
        UnitCell< D > const & cell ) [friend]
```

Write UnitCell<D> to a field file header (fortran pscf format).

Parameters

out	output stream
cell	UnitCell<D> to be written

Definition at line 88 of file UnitCell.tpp.

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

- UnitCell.h
- UnitCell3.cpp

12.67 Pscf::UnitCellBase< D > Class Template Reference

Base class template for a crystallographic unit cell.

```
#include <UnitCellBase.h>
```

Inheritance diagram for Pscf::UnitCellBase< D >:

classPscf_1_1UnitCellBase-eps-converted-to.pdf

Public Member Functions

- UnitCellBase ()

  Constructor.
- ∼UnitCellBase ()

  Destructor.
- void setLattice ()

  Compute all private data, given latticeSystem and parameters.
- void setParameters (FSArray< double, 6 > const &parameters)

  Set all the parameters of unit cell (new version).
- virtual double ksq (IntVec< D > const &k) const

  Compute square magnitude of reciprocal lattice vector.
- virtual double dksq (IntVec< D > const &vec, int n) const

  Compute derivative of square wavevector w/ respect to cell parameter.
- int nParameter () const

  Get the number of parameters in the unit cell.
- FSArray< double, 6 > parameters () const

  Get the parameters of this unit cell.
- double parameter (int i) const

  Get a single parameter of the unit cell.
- const RealVec< D > & rBasis (int i) const

  Get Bravais basis vector i, denoted by a_i.
- const RealVec< D > & kBasis (int i) const

  Get reciprocal basis vector i, denoted by b_i.
- double drBasis (int k, int i, int j) const

  Get component j of derivative of rBasis vector a_i w/respect to k.
- double dkBasis (int k, int i, int j) const

  Get component j of derivative of kBasis vector bi w/respect to k.
- double drrBasis (int k, int i, int j) const

  Get the derivative of dot product ri.rj with respect to parameter k.
- double dkkBasis (int k, int i, int j) const

  Get the derivative of dot product bi.bj with respect to parameter k.

Protected Attributes

- FArray< RealVec< D >, D > rBasis_

  Array of Bravais lattice basis vectors.
- FArray< RealVec< D >, D > kBasis_

  Array of reciprocal lattice basis vectors.
- FArray< FMatrix< double, D, D >, 6 > drBasis_

  Array of derivatives of rBasis.
- FArray< FMatrix< double, D, D >, 6 > dkBasis_

Array of derivatives of kBasis.

- FArray< FMatrix< double, D, D >, 6 > drrBasis_

  Array of derivatives of a_i.a_j.
- FArray< FMatrix< double, D, D >, 6 > dkkBasis_

  Array of derivatives of b_i.b_j.
- FArray< double, 6 > parameters_

  Parameters used to describe the unit cell.
- int nParameter_

  Number of parameters required to specify unit cell.

12.67.1 Detailed Description

template<int D>
class Pscf::UnitCellBase< D >

Base class template for a crystallographic unit cell.
Definition at line 29 of file UnitCellBase.h.

12.67.2 Constructor & Destructor Documentation

12.67.2.1 UnitCellBase() `template<int D>`

`Pscf::UnitCellBase< D >::UnitCellBase`

Constructor.
Definition at line 305 of file UnitCellBase.h.

12.67.2.2 ∼UnitCellBase() `template<int D>`

`Pscf::UnitCellBase< D >::~UnitCellBase`

Destructor.
Definition at line 313 of file UnitCellBase.h.

12.67.3 Member Function Documentation

12.67.3.1 setLattice() `template<int D>`

`void Pscf::UnitCellBase< D >::setLattice`

Compute all private data, given latticeSystem and parameters.
Calls initializeToZero, setBasis, computeDerivatives internally.
Definition at line 441 of file UnitCellBase.h.
Referenced by Pscf::operator>>(), and Pscf::readUnitCellHeader().

12.67.3.2 setParameters() `template<int D>`

```
void Pscf::UnitCellBase< D >::setParameters (
            FSArray< double, 6 > const & parameters )
```

Set all the parameters of unit cell (new version).

Parameters

parameters	array of unit cell parameters

Definition at line 320 of file UnitCellBase.h.

12.67.3.3 ksq() template<int D>

```
double Pscf::UnitCellBase< D >::ksq (
            IntVec< D > const & k ) const [virtual]
```

Compute square magnitude of reciprocal lattice vector.

Definition at line 333 of file UnitCellBase.h.

Referenced by Pscf::Pspg::Continuous::Block< D >::setupUnitCell(), and Pscf::Pspg::Continuous::Mixture< D >::setupUnitCell().

12.67.3.4 dksq() template<int D>

```
double Pscf::UnitCellBase< D >::dksq (
            IntVec< D > const & vec,
            int n ) const [virtual]
```

Compute derivative of square wavevector w/ respect to cell parameter.

This function computes and returns a derivative with respect to unit cell parameter number n of the square of a reciprocal lattice vector with integer coefficients given by the elements of vec.

Parameters

vec	vector of components of a reciprocal lattice vector
n	index of a unit cell parameter

Definition at line 352 of file UnitCellBase.h.

Referenced by Pscf::Pspg::Continuous::Block< D >::setupUnitCell().

12.67.3.5 nParameter() template<int D>

```
int Pscf::UnitCellBase< D >::nParameter [inline]
```

Get the number of parameters in the unit cell.

Definition at line 228 of file UnitCellBase.h.

Referenced by Pscf::Pspg::Continuous::Mixture< D >::setMesh(), Pscf::Pspg::Continuous::Block< D >::setupUnitCell(), and Pscf::Pspg::Continuous::Mixture< D >::setupUnitCell().

12.67.3.6 parameters() template<int D>

```
FSArray< double, 6 > Pscf::UnitCellBase< D >::parameters [inline]
```

Get the parameters of this unit cell.

Definition at line 236 of file UnitCellBase.h.

12.67.3.7 parameter() template<int D>

```
double Pscf::UnitCellBase< D >::parameter (
            int i ) const
```

Get a single parameter of the unit cell.

Parameters

i	array index of the desired parameter

Definition at line 249 of file UnitCellBase.h.

12.67.3.8 rBasis() template<int D>

```
const RealVec< D > & Pscf::UnitCellBase< D >::rBasis (
              int i ) const
```

Get Bravais basis vector i, denoted by a_i.

Parameters

i	array index of the desired basis vector

Definition at line 256 of file UnitCellBase.h.

12.67.3.9 kBasis() template<int D>

```
const RealVec< D > & Pscf::UnitCellBase< D >::kBasis (
              int i ) const [inline]
```

Get reciprocal basis vector i, denoted by b_i.

Parameters

i	array index of the desired reciprocal lattice basis vector

Definition at line 264 of file UnitCellBase.h.

12.67.3.10 drBasis() template<int D>

```
double Pscf::UnitCellBase< D >::drBasis (
              int k,
              int i,
              int j ) const [inline]
```

Get component j of derivative of rBasis vector a_i w/respect to k.

Parameters

i	array index of the desired basis vector a_i
j	index of a Cartesian component of a_i
k	index of cell parameter

Definition at line 272 of file UnitCellBase.h.

12.67.3.11 dkBasis() template<int D>

```
double Pscf::UnitCellBase< D >::dkBasis (
              int k,
              int i,
              int j ) const [inline]
```

Get component j of derivative of kBasis vector bi w/respect to k.

Parameters

i	array index of the desired reciprocal basis vector b_i
j	index of a Cartesian component of b_i
k	index of cell parameter

Definition at line 280 of file UnitCellBase.h.

12.67.3.12 drrBasis() `template<int D>`

```
double Pscf::UnitCellBase< D >::drrBasis (
            int k,
            int i,
            int j ) const  [inline]
```

Get the derivative of dot product ri.rj with respect to parameter k.

Parameters

i	array index of 1st Bravais basis vector b_i
j	array index of 2nd Bravais basis vector b_i
k	index of cell parameter

Definition at line 296 of file UnitCellBase.h.

12.67.3.13 dkkBasis() `template<int D>`

```
double Pscf::UnitCellBase< D >::dkkBasis (
            int k,
            int i,
            int j ) const  [inline]
```

Get the derivative of dot product bi.bj with respect to parameter k.

Parameters

i	array index of 1st reciprocal basis vector b_i
j	array index of 2nd reciprocal basis vector b_i
k	index of cell parameter

Definition at line 288 of file UnitCellBase.h.

12.67.4 Member Data Documentation

12.67.4.1 rBasis_ `template<int D>`

```
FArray<RealVec<D>, D> Pscf::UnitCellBase< D >::rBasis_  [protected]
```

Array of Bravais lattice basis vectors.

Definition at line 146 of file UnitCellBase.h.

12.67.4.2 kBasis_ template<int D>

FArray<RealVec<D>, D> Pscf::UnitCellBase< D >::kBasis_ [protected]

Array of reciprocal lattice basis vectors.

Definition at line 151 of file UnitCellBase.h.

12.67.4.3 drBasis_ template<int D>

FArray<FMatrix<double, D, D>, 6> Pscf::UnitCellBase< D >::drBasis_ [protected]

Array of derivatives of rBasis.

Element drBasis_[k](i,j) is derivative with respect to parameter k of component j of Bravais basis vector i.

Definition at line 159 of file UnitCellBase.h.

12.67.4.4 dkBasis_ template<int D>

FArray<FMatrix<double, D, D>, 6> Pscf::UnitCellBase< D >::dkBasis_ [protected]

Array of derivatives of kBasis.

Element dkBasis_[k](i,j) is derivative with respect to parameter k of component j of reciprocal basis vector i.

Definition at line 167 of file UnitCellBase.h.

12.67.4.5 drrBasis_ template<int D>

FArray<FMatrix<double, D, D>, 6> Pscf::UnitCellBase< D >::drrBasis_ [protected]

Array of derivatives of a_i.a_j.

Element drrBasis_[k](i,j) is derivative with respect to parameter k of the dot product (a_i.a_j) of Bravais lattice basis vectors a_i and a_j.

Definition at line 176 of file UnitCellBase.h.

12.67.4.6 dkkBasis_ template<int D>

FArray<FMatrix<double, D, D>, 6> Pscf::UnitCellBase< D >::dkkBasis_ [protected]

Array of derivatives of b_i.b_j.

Element dkkBasis_[k](i,j) is derivative with respect to parameter k of the dot product (b_i.b_j) of reciprocal lattice basis vectors b_i and b_j.

Definition at line 185 of file UnitCellBase.h.

12.67.4.7 parameters_ template<int D>

FArray<double, 6> Pscf::UnitCellBase< D >::parameters_ [protected]

Parameters used to describe the unit cell.

Definition at line 190 of file UnitCellBase.h.

Referenced by Pscf::operator<<(), Pscf::operator>>(), Pscf::readUnitCellHeader(), Pscf::serialize(), and Pscf::write↩ UnitCellHeader().

12.67.4.8 nParameter_ template<int D>

int Pscf::UnitCellBase< D >::nParameter_ [protected]

Number of parameters required to specify unit cell.

Definition at line 195 of file UnitCellBase.h.

Referenced by Pscf::operator<<(), Pscf::operator>>(), Pscf::readUnitCellHeader(), Pscf::serialize(), and Pscf::write↩ UnitCellHeader().

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

- UnitCellBase.h

12.68 Pscf::Vec< D, T > Class Template Reference

A Vec<D, T><D,T> is a D-component vector with elements of type T.

`#include <Vec.h>`

Public Member Functions

Constructors

- Vec ()
  Default constructor.
- Vec (const Vec< D, T > &v)
  Copy constructor.
- Vec (T const *v)
  Constructor from a C-array.
- Vec (T s)
  Constructor, initialize all elements to a common scalar value.

Assignment and Initialization

- Vec< D, T > & operator= (const Vec< D, T > &v)
  Copy assignment.
- Vec< D, T > & operator= (T s)
  Assignment all elements to the same scalar T value.
- Vec< D, T > & setToZero ()
  Set all elements to zero.

Arithmetic Assignment

- void operator+= (const Vec< D, T > &dv)
  Add vector dv to this vector.
- void operator-= (const Vec< D, T > &dv)
  Subtract vector dv from this vector.
- void operator+= (T s)
  Add a common scalar to all components.
- void operator-= (T s)
  Subtract a common scalar from all components.
- void operator*= (T s)
  Multiply this vector by scalar s.

Array Subscript

- const T & operator[ ] (int i) const
  Return one Cartesian element by value.
- T & operator[ ] (int i)
  Return one element of the vector by references.

Vec<D, T> valued functions (assigned to invoking object)

- Vec< D, T > & add (const Vec< D, T > &v1, const Vec< D, T > &v2)
  Add vectors v1 and v2.
- Vec< D, T > & subtract (const Vec< D, T > &v1, const Vec< D, T > &v2)
  Subtract vector v2 from v1.
- Vec< D, T > & multiply (const Vec< D, T > &v, T s)
  Multiply a vector v by a scalar s.

- Vec< D, T > & negate (const Vec< D, T > &v)

  Return negative of vector v.
- Vec< D, T > & negate ()

  Negate all elements of this vector.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize to/from an archive.

12.68.1 Detailed Description

template<int D, typename T>
class Pscf::Vec< D, T >

A Vec<D, T><D,T> is a D-component vector with elements of type T.
The elements of a Vec<D, T> can be accessed using subscript operator, as for a built in array.
The arithmetic assignment operators +=, -=, and *= are overloaded to allow vector-vector addition and subtraction and vector-scalar multiplication.
All other unary and binary mathematical operations are implemented as methods or free functions. Operations that yield a Vec<D, T>, such as addition, are implemented by methods that assign the result to the invoking Vec object, and return this object by reference. For example,

```
Vec<3, double> a, b, c;
a[0] = 0.0
a[1] = 1.0
a[2] = 2.0
b[0] =  0.5
b[1] = -0.5
b[2] = -1.5
// Set a = a + b
a += b
// Set b = b*2
b *= 2.0;
// Set c = a + b
c.add(a, b);
```

This syntax for functions that yield a vector makes the allocation of a temporary Vec<D, T> object explicit, by requiring that the invoking function be a member of an object that will hold the result.
For efficiency, all member functions are declared inline.
Definition at line 63 of file Vec.h.

12.68.2 Constructor & Destructor Documentation

12.68.2.1 Vec() [1/4] template<int D, typename T >

`Pscf::Vec< D, T >::Vec [inline]`

Default constructor.
Definition at line 315 of file Vec.h.

12.68.2.2 Vec() [2/4] template<int D, typename T >

```
Pscf::Vec< D, T >::Vec (
              const Vec< D, T > & v ) [inline]
```

Copy constructor.

Parameters

v	Vec<D, T> to be copied

Definition at line 322 of file Vec.h.

12.68.2.3 Vec() [3/4] template<int D, typename T >

Pscf::Vec< D, T >::Vec (
T const * v) [inline], [explicit]

Constructor from a C-array.

Parameters

v	array to be copied

Definition at line 333 of file Vec.h.

12.68.2.4 Vec() [4/4] template<int D, typename T >

Pscf::Vec< D, T >::Vec (
T s) [inline], [explicit]

Constructor, initialize all elements to a common scalar value.

Parameters

s	initial value for all elements.

Definition at line 344 of file Vec.h.

12.68.3 Member Function Documentation

12.68.3.1 operator=() [1/2] template<int D, typename T >

Vec< D, T > & Pscf::Vec< D, T >::operator= (
const Vec< D, T > & v) [inline]

Copy assignment.

Parameters

v	Vec<D, T> to assign.

Returns

this object, after modification

Definition at line 355 of file Vec.h.

12.68.3.2 operator=() [2/2] template<int D, typename T >

Vec< D, T > & Pscf::Vec< D, T >::operator= (
T s) [inline]

Assignment all elements to the same scalar T value.

Parameters

s	scalar value

Returns

this object, after modification

Definition at line 367 of file Vec.h.

12.68.3.3 setToZero() `template<int D, typename T >`

`Vec< D, T > & Pscf::Vec< D, T >::setToZero [inline]`

Set all elements to zero.

Returns

this object, after modification

Definition at line 379 of file Vec.h.

12.68.3.4 operator+=() [1/2] `template<int D, typename T >`

```
void Pscf::Vec< D, T >::operator+= (
            const Vec< D, T > & dv ) [inline]
```

Add vector dv to this vector.
Upon return, ∗this = this + dv.

Parameters

dv	vector increment (input)

Definition at line 391 of file Vec.h.

12.68.3.5 operator-=() [1/2] `template<int D, typename T >`

```
void Pscf::Vec< D, T >::operator-= (
            const Vec< D, T > & dv ) [inline]
```

Subtract vector dv from this vector.
Upon return, ∗this = this + dv.

Parameters

dv	vector increment (input)

Definition at line 402 of file Vec.h.

12.68.3.6 operator+=() [2/2] `template<int D, typename T >`

```
void Pscf::Vec< D, T >::operator+= (
            T s ) [inline]
```

Add a common scalar to all components.

Parameters

s	scalar additive constant (input)

Definition at line 413 of file Vec.h.

12.68.3.7 operator-=() [2/2]

```
template<int D, typename T >
void Pscf::Vec< D, T >::operator-= (
            T s ) [inline]
```

Subtract a common scalar from all components.

Parameters

s	scalar subtractive constant (input)

Definition at line 424 of file Vec.h.

12.68.3.8 operator*=()

```
template<int D, typename T >
void Pscf::Vec< D, T >::operator*= (
            T s ) [inline]
```

Multiply this vector by scalar s.

Upon return, *this = (*this)*s.

Parameters

s	scalar multiplier

Definition at line 435 of file Vec.h.

12.68.3.9 operator[]() [1/2]

```
template<int D, typename T >
const T & Pscf::Vec< D, T >::operator[] (
            int i ) const [inline]
```

Return one Cartesian element by value.

Parameters

i	element index

Returns

element i of the vector

Definition at line 446 of file Vec.h.

12.68.3.10 operator[]() [2/2]

```
template<int D, typename T >
T & Pscf::Vec< D, T >::operator[] (
            int i ) [inline]
```

Return one element of the vector by references.

Parameters

i	element index

Returns

element i of this vector

Definition at line 457 of file Vec.h.

12.68.3.11 add() template<int D, typename T >

```
Vec< D, T > & Pscf::Vec< D, T >::add (
              const Vec< D, T > & v1,
              const Vec< D, T > & v2 )  [inline]
```

Add vectors v1 and v2.
Upon return, ∗this = v1 + v2.

Parameters

v1	vector (input)
v2	vector (input)

Returns

modified invoking vector

Definition at line 471 of file Vec.h.
Referenced by Pscf::operator+().

12.68.3.12 subtract() template<int D, typename T >

```
Vec< D, T > & Pscf::Vec< D, T >::subtract (
              const Vec< D, T > & v1,
              const Vec< D, T > & v2 )  [inline]
```

Subtract vector v2 from v1.
Upon return, ∗this = v1 - v2.

Parameters

v1	vector (input)
v2	vector (input)

Returns

modified invoking vector

Definition at line 488 of file Vec.h.

12.68.3.13 multiply() template<int D, typename T >

```
Vec< D, T > & Pscf::Vec< D, T >::multiply (
              const Vec< D, T > & v,
              T s )  [inline]
```

Multiply a vector v by a scalar s.
Upon return, ∗this = v∗s.

Parameters

v	vector input
s	scalar input

Returns

modified invoking vector

Definition at line 503 of file Vec.h.
Referenced by Pscf::UnitCellBase< 3 >::ksq().

12.68.3.14 negate() [1/2]

```
template<int D, typename T >
Vec< D, T > & Pscf::Vec< D, T >::negate (
            const Vec< D, T > & v )  [inline]
```

Return negative of vector v.
Upon return, ∗this = -v;

Parameters

v	vector input

Returns

modified invoking vector

Definition at line 518 of file Vec.h.
Referenced by Pscf::Basis< D >::isValid().

12.68.3.15 negate() [2/2]

```
template<int D, typename T >
Vec< D, T > & Pscf::Vec< D, T >::negate  [inline]
```

Negate all elements of this vector.
Upon return, all elements of this have been negated (reversed)

Returns

this object, after modification

Definition at line 533 of file Vec.h.

12.68.3.16 serialize()

```
template<int D, typename T >
template<class Archive >
void Pscf::Vec< D, T >::serialize (
            Archive & ar,
            const unsigned int version )  [inline]
```

Serialize to/from an archive.
Implementation uses syntax of Boost::serialize.

Parameters

ar	archive
version	archive version id

Definition at line 547 of file Vec.h.
The documentation for this class was generated from the following file:

- Vec.h

12.69 Pscf::Vertex Class Reference

A junction or chain end in a block polymer.
`#include <Vertex.h>`

Public Member Functions

- void setId (int id)

  Set the integer identifier of this vertex.
- void addBlock (const BlockDescriptor &block)

  Add block to the list of attached blocks.
- void addBond (const BondDescriptor &bond)

  Add bond to the list of attached bonds.
- int id () const

  Get the id of this vertex.
- int size () const

  Get the number of attached blocks or bonds.
- const Pair< int > & inPropagatorId (int i) const

  Get the block/bonds and direction of an incoming propagator.
- const Pair< int > & outPropagatorId (int i) const

  Get the block/bond and direction of an outgoing propagator.

12.69.1 Detailed Description

A junction or chain end in a block polymer.
Definition at line 26 of file Vertex.h.

12.69.2 Member Function Documentation

12.69.2.1 setId()

```
void Pscf::Vertex::setId (
    int id )
```

Set the integer identifier of this vertex.

Parameters

id	identifier

Definition at line 25 of file Vertex.cpp.

12.69.2.2 addBlock()

```
void Pscf::Vertex::addBlock (
    const BlockDescriptor & block )
```

Add block to the list of attached blocks.
Preconditions: The id for this vertex must have been set, vertex ids must have been set for the block, and the id of this vertex must match one of the ids for the two vertices attached to the block.

Parameters

block	attached BlockDescriptor object

Definition at line 28 of file Vertex.cpp.
References Pscf::BlockDescriptor::id(), UTIL_THROW, and Pscf::BlockDescriptor::vertexId().

12.69.2.3 addBond()

`void Pscf::Vertex::addBond ( const BondDescriptor & bond )`

Add bond to the list of attached bonds.
Preconditions: The id for this vertex must have been set, vertex ids must have been set for the bond, and the id of this vertex must match one of the ids for the two vertices attached to the bond.

Parameters

bond	attached BondDescriptor object

Definition at line 59 of file Vertex.cpp.
References Pscf::BondDescriptor::id(), UTIL_THROW, and Pscf::BondDescriptor::vertexId().

12.69.2.4 id()

`int Pscf::Vertex::id ( ) const [inline]`

Get the id of this vertex.
Definition at line 107 of file Vertex.h.

12.69.2.5 size()

`int Pscf::Vertex::size ( ) const [inline]`

Get the number of attached blocks or bonds.
Definition at line 110 of file Vertex.h.
Referenced by Pscf::DPolymerTmpl< Bond< D > >::readParameters(), and Pscf::PolymerTmpl< Block< D > >↩ ::readParameters().

12.69.2.6 inPropagatorId()

`const Pair< int > & Pscf::Vertex::inPropagatorId ( int i ) const [inline]`

Get the block/bonds and direction of an incoming propagator.
The first element of the integer pair is the block/bond id, and the second is a direction id which is 0 if this vertex is vertex 1 of the block/bond, and 1 if this vertex is vertex 0.

Parameters

i	index of incoming propagator

Returns

Pair<int> containing block/bond index, direction index

Definition at line 114 of file Vertex.h.
Referenced by Pscf::DPolymerTmpl< Bond< D > >::readParameters(), and Pscf::PolymerTmpl< Block< D > >↩ ::readParameters().

12.69.2.7 outPropagatorId()

`const Pair< int > & Pscf::Vertex::outPropagatorId ( int i ) const [inline]`

Get the block/bond and direction of an outgoing propagator.

The first element of the integer pair is the block/bond id, and the second is a direction id which is 0 if this vertex is vertex 0 of the block/bond, and 1 if this vertex is vertex 1.

Parameters

i	index of incoming propagator

Returns

Pair<int> containing block/bond index, direction index

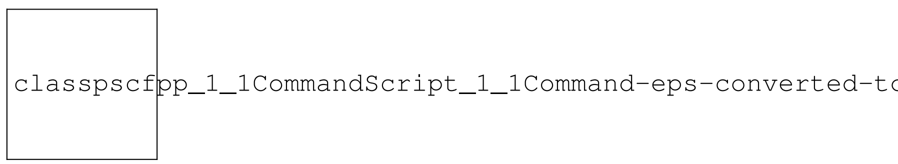
Definition at line 118 of file Vertex.h.

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

- Vertex.h
- Vertex.cpp

12.70 pscfpp.CommandScript.Command Class Reference

Inheritance diagram for pscfpp.CommandScript.Command:



12.70.1 Detailed Description

Definition at line 74 of file CommandScript.py.

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

- CommandScript.py

12.71 pscfpp.CommandScript.CommandScript Class Reference

12.71.1 Detailed Description

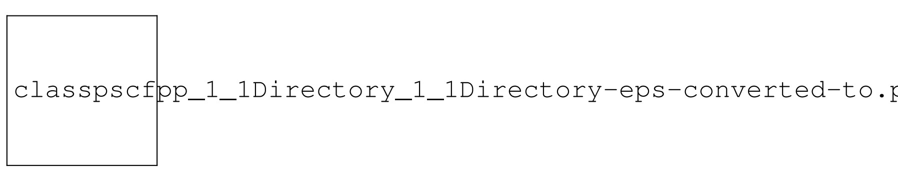
Definition at line 33 of file CommandScript.py.

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

- CommandScript.py

12.72 pscfpp.Directory.Directory Class Reference

Inheritance diagram for pscfpp.Directory.Directory:



12.72.1 Detailed Description

Definition at line 7 of file Directory.py.
The documentation for this class was generated from the following file:

- Directory.py

12.73 pscfpp.File.File Class Reference

Inheritance diagram for pscfpp.File.File:

classpscfpp_1_1File_1_1File-eps-converted-to.pdf

12.73.1 Detailed Description

Definition at line 5 of file File.py.
The documentation for this class was generated from the following file:

- File.py

12.74 pscfpp.FileEditor.FileEditor Class Reference

12.74.1 Detailed Description

Definition at line 8 of file FileEditor.py.
The documentation for this class was generated from the following file:

- FileEditor.py

12.75 pscfpp.Grep.Grep Class Reference

12.75.1 Detailed Description

Definition at line 6 of file Grep.py.
The documentation for this class was generated from the following file:

- Grep.py

12.76 pscfpp.MakeMaker.MakeMaker Class Reference

Public Member Functions

- def makeInclude (self, base)
- def srcSuffix (self)
- def find (self)

12.76.1 Detailed Description

Definition at line 11 of file MakeMaker.py.

12.76.2 Member Function Documentation

12.76.2.1 makeInclude() `def pscfpp.MakeMaker.MakeMaker.makeInclude ( self, base )`

```
Make include line suitable for a makefile
```

Definition at line 85 of file MakeMaker.py.
References pscfpp.MakeMaker.MakeMaker.pathFromSrc.
Referenced by pscfpp.MakeMaker.MakeMaker.find().

12.76.2.2 srcSuffix() `def pscfpp.MakeMaker.MakeMaker.srcSuffix ( self )`

```
Return suffix for source files: cpp or cc
```

Definition at line 92 of file MakeMaker.py.
References pscfpp.FileEditor.FileEditor.isTest, and pscfpp.MakeMaker.MakeMaker.isTest.
Referenced by pscfpp.MakeMaker.MakeMaker.find().

12.76.2.3 find() `def pscfpp.MakeMaker.MakeMaker.find ( self )`

```
Find all header and source files in this directory.
Note: Does not recursively descend into subdirectories
```

Definition at line 99 of file MakeMaker.py.
References pscfpp.MakeMaker.MakeMaker.dirname, pscfpp.Directory.Directory.dirs, pscfpp.MakeMaker.MakeMaker.dirs, Pscf::SymmetryGroup< Symmetry >.find(), pscfpp.MakeMaker.MakeMaker.find(), pscfpp.MakeMaker.MakeMaker.generation, pscfpp.MakeMaker.MakeMaker.globalInclude, pscfpp.MakeMaker.MakeMaker.hasHdrs, pscfpp.MakeMaker.MakeMaker.hasLib, pscfpp.MakeMaker.MakeMaker.hasSrcs, pscfpp.MakeMaker.MakeMaker.hdrs, pscfpp.FileEditor.FileEditor.isTest, pscfpp.MakeMaker.MakeMaker.isTest, pscfpp.MakeMaker.MakeMaker.libName, pscfpp.MakeMaker.MakeMaker.libObjs, pscfpp.MakeMaker.MakeMaker.linkObjs, pscfpp.MakeMaker.MakeMaker.makeInclude(), pscfpp.MakeMaker.MakeMaker.makePath(), pscfpp.MakeMaker.MakeMaker.makePathFromSrc(), pscfpp.File.File.path, pscfpp.Directory.Directory.path, pscfpp.MakeMaker.MakeMaker.path, pscfpp.MakeMaker.MakeMaker.pathFromSrc, pscfpp.MakeMaker.MakeMaker.pathToSrc, pscfpp.MakeMaker.MakeMaker.srcs, and pscfpp.MakeMaker.MakeMaker.srcSuffix().
Referenced by pscfpp.MakeMaker.MakeMaker.find().
The documentation for this class was generated from the following file:

- MakeMaker.py

12.77 pscfpp.ParamComposite.Blank Class Reference

12.77.1 Detailed Description

Definition at line 153 of file ParamComposite.py.
The documentation for this class was generated from the following file:

- ParamComposite.py

12.78 pscfpp.ParamComposite.ParamComposite Class Reference

12.78.1 Detailed Description

Definition at line 86 of file ParamComposite.py.
The documentation for this class was generated from the following file:

- ParamComposite.py

12.79 pscfpp.ParamComposite.Parameter Class Reference

Public Member Functions

- def read (self, records, i)
- def line (self, i)
- def value (self, i=0)
- def setValue (self, value, i=0)

12.79.1 Detailed Description

Definition at line 178 of file ParamComposite.py.

12.79.2 Member Function Documentation

12.79.2.1 read()

```
def pscfpp.ParamComposite.Parameter.read (
        self,
        records,
        i )
```

```
Read the line or lines associated with a parameter.
```

Definition at line 186 of file ParamComposite.py.
References pscfpp.ParamComposite.ParamComposite.indent_, Util::ParamComponent.indent_, pscfpp.ParamComposite.Parameter.indent_, pscfpp.CommandScript.CommandScript.label_, Util::End.label_, Util::XmlXmlTag.label_, Util::XmlAttribute.label_, Util::Begin.label_, Util::XmlEndTag.label_, pscfpp.CommandScript.Command.label_, pscfpp.ParamComposite.ParamComposite.label_, Util::XmlStartTag.label_, pscfpp.ParamComposite.Parameter.label_, Util::Parameter.label_, and pscfpp.ParamComposite.Parameter.records_.

12.79.2.2 line()

```
def pscfpp.ParamComposite.Parameter.line (
        self,
        i )
```

```
Get line i from a multi-line data format.
```

Definition at line 214 of file ParamComposite.py.
References pscfpp.ParamComposite.Parameter.records_.

12.79.2.3 value()

```
def pscfpp.ParamComposite.Parameter.value (
        self,
        i = 0 )
```

```
Get value i in line 0.
```

Definition at line 218 of file ParamComposite.py.
References pscfpp.ParamComposite.Parameter.records_.

12.79.2.4 setValue()

```
def pscfpp.ParamComposite.Parameter.setValue (
    self,
    value,
    i = 0 )
```

`Set value i in line 0.`

Definition at line 222 of file ParamComposite.py.
References pscfpp.ParamComposite.ParamComposite.indent_, Util::ParamComponent.indent_, pscfpp.ParamComposite.Parameter.indent_, pscfpp.CommandScript.CommandScript.label_, Util::End.label_, Util::XmlXmlTag.label_, Util::XmlAttribute.label_, Util::Begin.label_, Util::XmlEndTag.label_, pscfpp.CommandScript.Command.label_, pscfpp.ParamComposite.ParamComposite.label_, Util::XmlStartTag.label_, pscfpp.ParamComposite.Parameter.label_, Util::Parameter.label_, and pscfpp.ParamComposite.Parameter.records_.
The documentation for this class was generated from the following file:

- ParamComposite.py

12.80 pscfpp.ParamComposite.ParamRecord Class Reference

Inheritance diagram for pscfpp.ParamComposite.ParamRecord:

classpscfpp_1_1ParamComposite_1_1ParamRecord-eps-convert

12.80.1 Detailed Description

Definition at line 163 of file ParamComposite.py.
The documentation for this class was generated from the following file:

- ParamComposite.py

12.81 pscfpp.Record.Record Class Reference

Inheritance diagram for pscfpp.Record.Record:

classpscfpp_1_1Record_1_1Record-eps-converted-to.pdf

12.81.1 Detailed Description

Definition at line 13 of file Record.py.
The documentation for this class was generated from the following file:

- Record.py

12.82 pscfpp.RecordEditor.RecordEditor Class Reference

12.82.1 Detailed Description

Definition at line 5 of file RecordEditor.py.
The documentation for this class was generated from the following file:

- RecordEditor.py

12.83 pscfpp.TextWrapper.TextWrapper Class Reference

12.83.1 Detailed Description

Definition at line 1 of file TextWrapper.py.
The documentation for this class was generated from the following file:

- TextWrapper.py

12.84 Ridder< System, T > Class Template Reference

Public Member Functions

- Ridder ()=default

  Construct a new Ridder object.
- Ridder (int max_itr, T err)

  Construct a new Ridder object.
- ∼Ridder ()=default

  Destroy the Ridder object.

12.84.1 Detailed Description

template<class System, typename T = double>
class Ridder< System, T >

Definition at line 9 of file Ridder.h.

12.84.2 Constructor & Destructor Documentation

12.84.2.1 Ridder() [1/2] template<class System , typename T = double>

```
Ridder< System, T >::Ridder ( )  [default]
```

Construct a new Ridder object.

12.84.2.2 Ridder() [2/2] template<class System , typename T >

```
Ridder< System, T >::Ridder (
            int max_itr,
            T err )
```

Construct a new Ridder object.

Parameters

max←_itr	
err	

Definition at line 46 of file Ridder.h.

12.84.2.3 ∼Ridder() template<class System , typename T = double>

`Ridder< System, T >::~Ridder ( ) [default]`

Destroy the Ridder object.

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

- Ridder.h

12.85 TestA Class Reference

This example shows how to construct and run a single UnitTest class.

Inheritance diagram for TestA:

classTestA-eps-converted-to.pdf

Additional Inherited Members

12.85.1 Detailed Description

This example shows how to construct and run a single UnitTest class.

Trivial example of UnitTest use for parallel MPI job.

This file demonstrates the usage of a composite test runner.

We create a subclass of UnitTest named TestA, which has 3 test methods. We then use a set of preprocessor macros to define an associated subclass of UnitTestRunner. The name of the UnitTestRunner subclass is given by the macro TEST_RUNNER(TestA), which expands to TestA_Runner.

In the main program, we create an instance of TEST_RUNNER(TestA) and call its run method, which runs all 3 test methods in sequence. Trivial subclass of UnitTest, for illustration.

A CompositeTestRunner is a TestRunner that runs and accumulates statistics of the tests associated with several child TestRunner objects. The child runners are usually instances of UnitTestRunner.

To demonstrate the usage, we define two trivial unit tests, TestA and TestB, and use macros to define associated UnitTestRunner subclasses, TEST_RUNNER(TestA) and TEST_RUNNER(TestB). The preprocessor macros in the main program then define a class CompositeExample that is derived from CompositeTestRunner, which contains instances of the TEST_RUNNER(TestA) and TEST_RUNNER(TestB). Calling the run() method of the CompositeExample then runs the all of the tests defined in TestA and TestB. Trivial UnitTest A.

Trivial subclass of UnitTest for an MPI job.

Definition at line 20 of file example1.cpp.

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

- example1.cpp
- example2.cpp
- example3.cpp

12.86 TestB Class Reference

Trivial UnitTest B.

Inheritance diagram for TestB:

classTestB-eps-converted-to.pdf

Additional Inherited Members

12.86.1 Detailed Description

Trivial UnitTest B.
Definition at line 60 of file example2.cpp.
The documentation for this class was generated from the following file:

- example2.cpp

12.87 TestException Class Reference

An exception thrown by a failed unit test.
`#include <TestException.h>`

Public Member Functions

- TestException ()

  Default constructor.
- TestException (const char ∗function, const char ∗message, const char ∗file, int line)

  Constructor for throwing.
- TestException (const char ∗message, const char ∗file, int line)

  Constructor without function name parameter.
- ∼TestException ()

  Destructor.
- void write (std::ostream &out)

  Write error message to output stream.
- const std::string & message ()

  Return the error message.

Protected Attributes

- std::string message_

  Error message.

12.87.1 Detailed Description

An exception thrown by a failed unit test.
Definition at line 19 of file TestException.h.

12.87.2 Constructor & Destructor Documentation

12.87.2.1 TestException() `[1/3]` `TestException::TestException ( )`

Default constructor.
Function Definitions --—.
Definition at line 87 of file TestException.h.

12.87.2.2 TestException() [2/3]

```
TestException::TestException (
    const char * function,
    const char * message,
    const char * file,
    int line )
```

Constructor for throwing.

Constructs error message that includes file and line number. Values of the file and line parameters should be given by the built-in macros **FILE** and **LINE**, respectively, in the calling function. A typical call of the constructor is thus of the form:

```
throw TestException("MyClass::myFunction", "A terrible thing happened!",
                    __FILE__, __LINE__ );
```

Parameters

function	name of the function from which the TestException was thrown
message	message describing the nature of the error
file	name of the file from which the TestException was thrown
line	line number in file

Definition at line 94 of file TestException.h.

References message(), and message_.

12.87.2.3 TestException() [3/3]

```
TestException::TestException (
    const char * message,
    const char * file,
    int line )
```

Constructor without function name parameter.

Parameters

message	message describing the nature of the error
file	name of the file from which the TestException was thrown
line	line number in file

Definition at line 116 of file TestException.h.

References message(), and message_.

12.87.2.4 ~TestException()

```
TestException::~TestException ( )
```

Destructor.

Definition at line 134 of file TestException.h.

12.87.3 Member Function Documentation

12.87.3.1 write()

```
void TestException::write (
    std::ostream & out ) [inline]
```

Write error message to output stream.

Parameters

out	output stream

Definition at line 142 of file TestException.h.
References message_.

12.87.3.2 message() `const std::string & TestException::message ( ) [inline]`

Return the error message.
Definition at line 148 of file TestException.h.
References message_.
Referenced by UnitTestRunner< UnitTestClass >::method(), and TestException().

12.87.4 Member Data Documentation

12.87.4.1 message_ `std::string TestException::message_ [protected]`

Error message.
Definition at line 78 of file TestException.h.
Referenced by message(), TestException(), and write().
The documentation for this class was generated from the following file:

- TestException.h

12.88 TestRunner Class Reference

Abstract base class for classes that run tests.
`#include <TestRunner.h>`
Inheritance diagram for TestRunner:

classTestRunner-eps-converted-to.pdf

Public Member Functions

- TestRunner ()

  Constructor.
- virtual ~TestRunner ()

  Destructor.
- virtual int run ()=0

  Run all tests.
- void recordFailure ()

  Increment counter for failed tests, and that of parent (if any).
- void recordSuccess ()

  Increment counter for successful tests, and that of parent (if any).
- void setParent (TestRunner &parent)

  Set another TestRunner as the parent.
- TestRunner & parent ()

  Return the parent object, if any.
- bool hasParent () const

Does this object have a parent?

- int nSuccess () const

  Return number of successful tests run.

- int nFailure () const

  Return number of failed tests run.

- void report () const

  If this object has no parent, report success and failure counters.

- bool isIoProcessor () const

  Is this the IO processor of an MPI communicator?

- virtual void addFilePrefix (const std::string &prefix)

  Prepend argument prefix to existing filePrefix.

- const std::string & filePrefix () const

  Return file prefix by const reference.

Protected Attributes

- std::string filePrefix_

  Prefix added to file names.

12.88.1 Detailed Description

Abstract base class for classes that run tests.
TestRunner is an abstract base class with two types of subclass: The UnitTestRunner class template defines a TestRunner that runs the tests for an associated UnitTest. A CompositeTestRunner runs the tests for a sequence of other TestRunner objects, each of which can be a UnitTestRunner or another CompositeTestRunner.
An implementation of the pure virtual run() method of must run all of the associated test methods, and records the number nSuccess() of tests that succeed and the number nFailure() that fail. A test fails if it throws a TestException. Test methods use the TEST_ASSERT(expr) macro to assert the truth of a logical exprression expr, which throws a TestException if expr is false. The implementation of run() for a UnitTestRunner runs each unit test method of the associated UnitTest in a try-catch block and catches any thrown TestExceptions. The implementation of run for a Test←Composite calls the run() method for each of its children.
Each TestRunner may optionally have a parent TestRunner. The parent if any, is always a TestComposite. A Test←Composite can have any number of children.

The recordFailure() and recordSuccess() methods of a TestRunner, which can be called by the run method, increment the nSuccess or nFailure counters. If the TestRunner has a parent, each function also calls the corresponding function of the parent, thus incrementing the corresponding counter of the parent. The nSuccess and nFailure counters for a TestComposite thereby keep track of the total number of successful and failed unit test methods run by all descendants.
Each TestRunner has a filePrefix string. The filePrefix is initialized to an empty string, and may be modified by the virtual addFilePrefix() function. The default implementation of this function prepends a string argument to the existing filePrefix.
The UnitTestRunner class template supplies the filePrefix to instances of the associated UnitTest class when a UnitTest is created. (See the notes for UnitTestRunner for details of how). The filePrefix string of the UnitTest is then prepended to the names of any files opened by the functions openInputFile(), openOutputFile(), and openFile of the UnitTest subclass.
The implementation of addFilePrefix() by the TestComposite subclass calls the addFilePrefix method of each of its children, and thus allows a common prefix to be added to the file paths used by all of its children.
Definition at line 73 of file TestRunner.h.

12.88.2 Constructor & Destructor Documentation

12.88.2.1 TestRunner() `TestRunner::TestRunner ( )`

Constructor.
Definition at line 252 of file TestRunner.h.

12.88.2.2 ∼TestRunner() `TestRunner::∼TestRunner ( ) [virtual]`

Destructor.
Definition at line 278 of file TestRunner.h.

12.88.3 Member Function Documentation

12.88.3.1 run() `virtual int TestRunner::run ( ) [pure virtual]`

Run all tests.

Returns

number of failures.

Implemented in UnitTestRunner< UnitTestClass >, and CompositeTestRunner.

12.88.3.2 recordFailure() `void TestRunner::recordFailure ( )`

Increment counter for failed tests, and that of parent (if any).
Definition at line 284 of file TestRunner.h.
References hasParent(), isIoProcessor(), parent(), and recordFailure().
Referenced by recordFailure().

12.88.3.3 recordSuccess() `void TestRunner::recordSuccess ( )`

Increment counter for successful tests, and that of parent (if any).
Definition at line 297 of file TestRunner.h.
References hasParent(), isIoProcessor(), parent(), and recordSuccess().
Referenced by recordSuccess().

12.88.3.4 setParent() `void TestRunner::setParent ( TestRunner & parent ) [inline]`

Set another TestRunner as the parent.

Parameters

parent	parent CompositeTestRunner object

Definition at line 199 of file TestRunner.h.
References parent().
Referenced by CompositeTestRunner::addChild().

12.88.3.5 parent() `TestRunner & TestRunner::parent ( ) [inline]`

Return the parent object, if any.
Definition at line 205 of file TestRunner.h.
Referenced by recordFailure(), recordSuccess(), and setParent().

12.88.3.6 hasParent() `bool TestRunner::hasParent ( ) const [inline]`

Does this object have a parent?

Definition at line 211 of file TestRunner.h.

Referenced by recordFailure(), recordSuccess(), and report().

12.88.3.7 nSuccess() `int TestRunner::nSuccess ( ) const [inline]`

Return number of successful tests run.

Definition at line 217 of file TestRunner.h.

12.88.3.8 nFailure() `int TestRunner::nFailure ( ) const [inline]`

Return number of failed tests run.

Definition at line 223 of file TestRunner.h.

Referenced by CompositeTestRunner::run().

12.88.3.9 report() `void TestRunner::report ( ) const`

If this object has no parent, report success and failure counters.

Definition at line 310 of file TestRunner.h.

References hasParent(), and isIoProcessor().

Referenced by CompositeTestRunner::run().

12.88.3.10 isIoProcessor() `bool TestRunner::isIoProcessor ( ) const [inline]`

Is this the IO processor of an MPI communicator?

Definition at line 236 of file TestRunner.h.

Referenced by recordFailure(), recordSuccess(), report(), and UnitTestRunner< UnitTestClass >::UnitTestRunner().

12.88.3.11 addFilePrefix() `void TestRunner::addFilePrefix ( const std::string & prefix ) [virtual]`

Prepend argument prefix to existing filePrefix.

Reimplemented in CompositeTestRunner.

Definition at line 323 of file TestRunner.h.

References filePrefix_.

Referenced by CompositeTestRunner::addChild(), and CompositeTestRunner::addFilePrefix().

12.88.3.12 filePrefix() `const std::string & TestRunner::filePrefix ( ) const [inline]`

Return file prefix by const reference.

Definition at line 230 of file TestRunner.h.

References filePrefix_.

12.88.4 Member Data Documentation

12.88.4.1 filePrefix_ `std::string TestRunner::filePrefix_ [protected]`

Prefix added to file names.

Definition at line 167 of file TestRunner.h.

Referenced by addFilePrefix(), and filePrefix().

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

- TestRunner.h

12.89 UnitTest Class Reference

UnitTest is a base class for classes that define unit tests.

#include <UnitTest.h>

Inheritance diagram for UnitTest:

classUnitTest-eps-converted-to.pdf

Public Member Functions

- UnitTest ()

  Constructor.
- virtual ~UnitTest ()

  Destructor.
- virtual void setUp ()

  Set up before each test method (empty default implementation).
- virtual void tearDown ()

  Tear down after each test method (empty default implementation).
- void setVerbose (int verbose)

  Set verbosity level.
- void setFilePrefix (const std::string &prefix)

  Set file prefix.
- const std::string & filePrefix ()

  Get file prefix string.
- bool isIoProcessor () const

  Should this processor read and write to file?

Protected Member Functions

- void printMethod (const char *methodName)

  Write name of a class method, iff ioProcessor.
- void printEndl ()

  Write carriage return, iff isIoProcessor.
- virtual void endMarker ()

  Print a line of hashes, iff isIoProcessor.
- void openInputFile (const std::string &name, std::ifstream &in) const

  Open C++ input file ifstream.
- void openOutputFile (const std::string &name, std::ofstream &out) const

Open C++ output file ofstream.

- FILE ∗ openFile (const std::string &name, const char ∗mode) const

  Open C file handle with specified mode.

- int verbose () const

  Return integer verbosity level (0 == silent).

Static Protected Member Functions

- static bool eq (int s1, int s2)

  Return true if two integers are equal.

- static bool eq (double s1, double s2)

  Return true if two double precision floats are equal.

12.89.1 Detailed Description

UnitTest is a base class for classes that define unit tests.
Each subclass of UnitTest should define one or more test methods. Each test method must be a zero parameter function that returns void. Test methods may be given arbitrary names. Individual test methods should use the preprocessor macro TEST_ASSERT(expression) defined in TextException.h to assert the truth of logical expressions.
The test methods defined by a UnitTest are run by an associated subclass of TestRunner. Each test method of a UnitTest must be added to the associated TestRunner. The run() method of a TestRunner calls all of the associated test methods in the order in which they were added, and counts the number of successful and failed tests.
The TestRunner associated with a single UnitTest is defined by a class template UnitTestRunner, which takes a UnitTest subclass as a template argument. For example, the TestRunner associated with a UnitTest subclass named TestA is a template instantiation UnitTestRunner<TestA>.
Preprocessor macros defined in the file UnitTestRunner.h should be used to create the boiler-plate code necessary to define a unit test runner and to add test methods to it.
Definition at line 50 of file UnitTest.h.

12.89.2 Constructor & Destructor Documentation

12.89.2.1 UnitTest() `UnitTest::UnitTest ( )`

Constructor.
Definition at line 219 of file UnitTest.h.

12.89.2.2 ∼UnitTest() `UnitTest::∼UnitTest ( ) [virtual]`

Destructor.
Definition at line 236 of file UnitTest.h.

12.89.3 Member Function Documentation

12.89.3.1 setUp() `void UnitTest::setUp ( ) [virtual]`

Set up before each test method (empty default implementation).
Definition at line 242 of file UnitTest.h.

12.89.3.2 tearDown() `void UnitTest::tearDown ( ) [virtual]`

Tear down after each test method (empty default implementation).
Reimplemented in ParamFileTest.
Definition at line 248 of file UnitTest.h.

12.89.3.3 setVerbose() `void UnitTest::setVerbose ( int verbose )`

Set verbosity level.

Parameters

verbose	verbosity level (0 = silent).

Definition at line 256 of file UnitTest.h.
References verbose().

12.89.3.4 setFilePrefix() `void UnitTest::setFilePrefix ( const std::string & prefix )`

Set file prefix.
This function is called by the UnitTestRunner::method(int i) function to set the filePrefix of the unit test equal to that of the runner after construction but before running the relevant test method.

Parameters

prefix	string to be prepended to input and output file names.

Definition at line 262 of file UnitTest.h.

12.89.3.5 filePrefix() `const std::string & UnitTest::filePrefix ( )`

Get file prefix string.
Definition at line 268 of file UnitTest.h.

12.89.3.6 isIoProcessor() `bool UnitTest::isIoProcessor ( ) const`

Should this processor read and write to file?
Definition at line 274 of file UnitTest.h.
Referenced by endMarker(), ParamFileTest::openFile(), printEndl(), and printMethod().

12.89.3.7 printMethod() `void UnitTest::printMethod ( const char * methodName ) [protected]`

Write name of a class method, iff ioProcessor.

Parameters

methodName	name of class test method

Definition at line 314 of file UnitTest.h.
References isIoProcessor().

12.89.3.8 printEndl() `void UnitTest::printEndl ( ) [protected]`

Write carriage return, iff isIoProcessor.
Definition at line 324 of file UnitTest.h.
References isIoProcessor().

12.89.3.9 endMarker() `void UnitTest::endMarker ( ) [protected], [virtual]`

Print a line of hashes, iff isIoProcessor.
Definition at line 330 of file UnitTest.h.
References isIoProcessor().

12.89.3.10 openInputFile() `void UnitTest::openInputFile (`

```
const std::string & name,
std::ifstream & in ) const [protected]
```

Open C++ input file ifstream.
This function adds the filePrefix before the name parameter. It does not check if this node isIoProcessor.

Parameters

name	base file name (added to filePrefix).
in	input file (opened on return).

Definition at line 343 of file UnitTest.h.
Referenced by ParamFileTest::openFile().

12.89.3.11 openOutputFile() `void UnitTest::openOutputFile (`

```
const std::string & name,
std::ofstream & out ) const [protected]
```

Open C++ output file ofstream.
This function adds the filePrefix before the name parameter. It does not check if this node isIoProcessor.

Parameters

name	base file name (added to filePrefix)
out	output file (opened on return)

Definition at line 361 of file UnitTest.h.

12.89.3.12 openFile() `FILE * UnitTest::openFile (`

```
const std::string & name,
const char * mode ) const [protected]
```

Open C file handle with specified mode.
This function adds the filePrefix before the name parameter. It does not check if this node isIoProcessor.

Parameters

name	base file name (added to filePrefix)

Parameters

mode	string that specified read or write mode

Returns

C file handle, opened for reading or writing

Definition at line 379 of file UnitTest.h.

12.89.3.13 verbose() `int UnitTest::verbose ( ) const [inline], [protected]`

Return integer verbosity level (0 == silent).
Definition at line 397 of file UnitTest.h.
Referenced by setVerbose().

12.89.3.14 eq() `[1/2]` `bool UnitTest::eq (`

```
int s1,
int s2 ) [inline], [static], [protected]
```

Return true if two integers are equal.
Definition at line 403 of file UnitTest.h.

12.89.3.15 eq() `[2/2]` `bool UnitTest::eq (`

```
double s1,
double s2 ) [static], [protected]
```

Return true if two double precision floats are equal.
Definition at line 409 of file UnitTest.h.
The documentation for this class was generated from the following file:

- UnitTest.h

12.90 UnitTestRunner< UnitTestClass > Class Template Reference

Template for a TestRunner that runs test methods of an associated UnitTest.
`#include <UnitTestRunner.h>`
Inheritance diagram for UnitTestRunner< UnitTestClass >:

classUnitTestRunner-eps-converted-to.pdf

Public Types

- typedef void(UnitTestClass::∗ MethodPtr) ()

  Pointer to a test method of the associated UnitTest class.

Public Member Functions

- UnitTestRunner ()

  Constructor.
- ∼UnitTestRunner ()

  Destructor.
- void addTestMethod (MethodPtr methodPtr)

  Register a test method of the associated unit test class.
- int nTestMethod ()

  Return the number of registered test methods.
- void method (unsigned int i)

  Run test method number i.
- virtual int run ()

  Run all registered test methods in the order added.
- int nFailure () const

  Return number of failed tests run.
- bool isIoProcessor () const

  Is this the IO processor of an MPI communicator?

Additional Inherited Members

12.90.1 Detailed Description

template<class UnitTestClass>
class UnitTestRunner< UnitTestClass >

Template for a TestRunner that runs test methods of an associated UnitTest.
A instance of UnitTestRunner<MyTest> holds an array of pointers to all of the test methods of a class MyTest that is a subclass of UnitTest. Each such test method must return void and take zero parameters. The addTestMethod() method is used to register a test method with the UnitTestRunner instantiation, by adding a pointer to a test method to this array. The run() method runs all of the registered test methods in sequence.
To run a set of unit tests one must:

- Define a subclass of UnitTest,
- Define an associated subclass of UnitTestRunner,
- Construct a UnitTestRunner object and call run().

The boilerplate code required to define a UnitTestRunner class may be simplified by using set preprocessor macros that are defined at the end of this file.
Here is an example of the code to to define a subclass of UnitTestRunner<MyTest>, associated with a subclass MyTest of UnitTest, and then run all of its test methods, written without using any preprocessor macros:

```
// Define a UnitTest class
class MyTest : public UnitTest {
public:
   test1()
   { .... }
   test2
   { ...  }
};
// Define a UnitTestRunner associated with MyTest
class MyTest_Runner : public UnitTestRunner<MyTest> {
public:
   MyTest_Runner(){
      addTestMethod(&MyTest::test1);
      addTestMethod(&MyTest::test2);
   }
}
// Run the tests.
```

```
MyTest_Runner runner;
runner.run();
```

Note that, by convention:

- We defined a subclass of UnitTestRunner<MyTest>, called MyTest_Runner.
- All test methods of MyTest are registered in the MyTest_Runner constructor.

Calling the run() method of MyTest_Runner will then run all of the tests.
The following series of preprocessor macros may be used to generate the definition of the MyTest_Runner class in the above example, and to create an instance of this class:

```
TEST_BEGIN(MyTest)
TEST_ADD(MyTest, test1)
TEST_ADD(MyTest, test2)
TEST_END(MyTest)
TEST_RUNNER(MyTest) runner;
runner.run();
```

The macro TEST_BEGIN(TestClass) generates the beginning of the class definition for subclass MyTest_Runner of UnitTestRunner<TestClass>. The TEST_ADD(TestClass, Method) adds a specified method of the associated class TestClass to the constructor of the new UnitTestRunner class. The TEST_END macro closes both the constructor definition and the class definition. After expansion, the resulting code is completely equivalent to that given in the previous example, after the definition of MyTest.
The name of the UnitTestRunner class created by these preprocessor macros is created by appending the standard suffix "_Runner" to the name of the unit test class. Thus, in the above example, the TestRunner subclass is named My↩Test_Runner. This TestRunner subclass name may be referred directly, using this name, or by using the preprocessor macro TEST_RUNNER(TestClass), which expands to the name of the test runner class, e.g., to TestClass_Runner. In the above example, this macro is used as a class name to instantiate an instance of the of required test runner.
Definition at line 110 of file UnitTestRunner.h.

12.90.2 Member Typedef Documentation

12.90.2.1 MethodPtr `template<class UnitTestClass >`

`typedef void(UnitTestClass::* UnitTestRunner< UnitTestClass >::MethodPtr) ()`

Pointer to a test method of the associated UnitTest class.
Definition at line 121 of file UnitTestRunner.h.

12.90.3 Constructor & Destructor Documentation

12.90.3.1 UnitTestRunner() `template<class UnitTestClass >`

`UnitTestRunner< UnitTestClass >::UnitTestRunner`

Constructor.
Definition at line 172 of file UnitTestRunner.h.
References TestRunner::isIoProcessor().

12.90.3.2 ∼UnitTestRunner() `template<class UnitTestClass >`

`UnitTestRunner< UnitTestClass >::∼UnitTestRunner`

Destructor.
Definition at line 189 of file UnitTestRunner.h.

12.90.4 Member Function Documentation

12.90.4.1 addTestMethod() `template<class UnitTestClass >`

```
void UnitTestRunner< UnitTestClass >::addTestMethod (
            MethodPtr methodPtr )
```

Register a test method of the associated unit test class.
Definition at line 196 of file UnitTestRunner.h.

12.90.4.2 nTestMethod() `template<class UnitTestClass >`

```
int UnitTestRunner< UnitTestClass >::nTestMethod
```

Return the number of registered test methods.
Definition at line 203 of file UnitTestRunner.h.

12.90.4.3 method() `template<class UnitTestClass >`

```
void UnitTestRunner< UnitTestClass >::method (
            unsigned int i )
```

Run test method number i.

Parameters

i	index of test method

Definition at line 212 of file UnitTestRunner.h.
References TestException::message().

12.90.4.4 run() `template<class UnitTestClass >`

```
int UnitTestRunner< UnitTestClass >::run  [virtual]
```

Run all registered test methods in the order added.
Implements TestRunner.
Definition at line 306 of file UnitTestRunner.h.

12.90.4.5 nFailure() `template<class UnitTestClass >`

```
int TestRunner::nFailure  [inline]
```

Return number of failed tests run.
Definition at line 223 of file TestRunner.h.

12.90.4.6 isIoProcessor() `template<class UnitTestClass >`

```
bool TestRunner::isIoProcessor  [inline]
```

Is this the IO processor of an MPI communicator?
Definition at line 236 of file TestRunner.h.
The documentation for this class was generated from the following file:

- UnitTestRunner.h

12.91 Util::Ar1Process Class Reference

Generator for a discrete AR(1) Markov process.

```
#include <Ar1Process.h>
```

Public Member Functions

- Ar1Process ()

  Constructor.
- Ar1Process (Random &random)

  Constructor.
- void setRNG (Random &random)

  Associate a random number generator.
- void init (double tau)

  Initialize process.
- double operator() ()

  Generate and return a new value.

12.91.1 Detailed Description

Generator for a discrete AR(1) Markov process.

An auto-regressive AR(1) process is a discrete stationary Markov process x(n) with an autocorrelation function <x(n)∗x(n+m)> = exp(-m/tau), where tau is a decay time. It is a discrete version of the Ornstein-Uhlenbeck continuous Markov process.

Definition at line 27 of file Ar1Process.h.

12.91.2 Constructor & Destructor Documentation

12.91.2.1 Ar1Process() [1/2] `Util::Ar1Process::Ar1Process ( )`

Constructor.

Definition at line 16 of file Ar1Process.cpp.

12.91.2.2 Ar1Process() [2/2] `Util::Ar1Process::Ar1Process ( Random & random )`

Constructor.

Parameters

random	associated random number generator.

Definition at line 27 of file Ar1Process.cpp.

12.91.3 Member Function Documentation

12.91.3.1 setRNG() `void Util::Ar1Process::setRNG ( Random & random )`

Associate a random number generator.

Parameters

random	associated random number generator.

Definition at line 38 of file Ar1Process.cpp.

12.91.3.2 init() `void Util::Ar1Process::init ( double tau )`

Initialize process.

Parameters

tau	decay time (in discrete steps)

Definition at line 46 of file Ar1Process.cpp.
References Util::Random::gaussian(), and UTIL_THROW.

12.91.3.3 operator()() `double Util::Ar1Process::operator() ( ) [inline]`

Generate and return a new value.
Definition at line 77 of file Ar1Process.h.
References Util::Random::gaussian().
The documentation for this class was generated from the following files:

- Ar1Process.h
- Ar1Process.cpp

12.92 Util::Array< Data > Class Template Reference

Array container class template.
`#include <Array.h>`
Inheritance diagram for Util::Array< Data >:

classUtil_1_1Array-eps-converted-to.pdf

Public Member Functions

- virtual ~Array ()

    Destructor.
- int capacity () const

    Return allocated size.
- void begin (ArrayIterator< Data > &iterator)

    Set an iterator to begin this Array.
- void begin (ConstArrayIterator< Data > &iterator) const

    Set a const iterator to begin this Array.
- Data & operator[ ] (int i)

    Get an element by non-const reference.
- const Data & operator[ ] (int i) const

    Get an element by const reference.
- Data * cArray ()

Return pointer to underlying C array.

- const Data ∗ cArray () const

  Return pointer to const to underlying C array.

Protected Member Functions

- Array ()

  Default constructor.

Protected Attributes

- Data ∗ data_

  Pointer to an array of Data elements.

- int capacity_

  Allocated size of the data_ array.

12.92.1 Detailed Description

template<typename Data>
class Util::Array< Data >

Array container class template.
An Array is a sequence that supports random access via an overloaded operator [], and that wraps a dynamically allocated C array.
Array is a base class for DArray, which is dynamically allocated, and RArray, which acts as a reference to another DArray or FSArray.
When compiled in debug mode (i.e., when NDEBUG is not defined) the subscript operator [] checks the validity of the element index.
Definition at line 28 of file AutoCorrArray.h.

12.92.2 Constructor & Destructor Documentation

12.92.2.1 ∼Array() `template<typename Data >`

`Util::Array< Data >::∼Array  [virtual]`

Destructor.
Definition at line 146 of file Array.h.

12.92.2.2 Array() `template<typename Data >`

`Util::Array< Data >::Array  [protected]`

Default constructor.
Protected to prevent direct instantiation.
Definition at line 137 of file Array.h.

12.92.3 Member Function Documentation

12.92.3.1 capacity()

`template<typename Data >`
`int Util::Array< Data >::capacity [inline]`

Return allocated size.

Returns

Number of elements allocated in array.

Definition at line 153 of file Array.h.

Referenced by Util::DRaggedMatrix< Data >::allocate(), Util::ArraySet< Data >::allocate(), Util::RArray< Data >::associate(), Util::bcast(), Pscf::Homogeneous::Mixture::computePhi(), Util::Distribution::Distribution(), Util::IntDistribution::IntDistribution(), Util::RadialDistribution::loadParameters(), Util::IntDistribution::loadParameters(), Util::Distribution::loadParameters(), Util::IntDistribution::operator=(), Util::Distribution::operator=(), Util::DArray< Pscf::Monomer >::operator=(), Util::Polynomial< double >::Polynomial(), Util::recv(), Util::send(), Util::Distribution::serialize(), and Pscf::LuSolver::solve().

12.92.3.2 begin() [1/2]

```
template<typename Data >
void Util::Array< Data >::begin (
            ArrayIterator< Data > & iterator ) [inline]
```

Set an iterator to begin this Array.

Parameters

iterator	ArrayIterator, initialized on output.

Definition at line 160 of file Array.h.

12.92.3.3 begin() [2/2]

```
template<typename Data >
void Util::Array< Data >::begin (
            ConstArrayIterator< Data > & iterator ) const [inline]
```

Set a const iterator to begin this Array.

Parameters

iterator	ConstArrayIterator, initialized on output.

Definition at line 172 of file Array.h.

12.92.3.4 operator[]() [1/2]

```
template<typename Data >
Data & Util::Array< Data >::operator[] (
            int i ) [inline]
```

Get an element by non-const reference.
Mimic C-array subscripting.

Parameters

i	array index

Returns

non-const reference to element i

Definition at line 184 of file Array.h.

12.92.3.5 operator[]() [2/2] template<typename Data >

```
const Data & Util::Array< Data >::operator[] (
              int i ) const  [inline]
```

Get an element by const reference.
Mimics C-array subscripting.

Parameters

i	array index

Returns

const reference to element i

Definition at line 196 of file Array.h.

12.92.3.6 cArray() [1/2] template<typename Data >

```
const Data * Util::Array< Data >::cArray  [inline]
```

Return pointer to underlying C array.
Definition at line 208 of file Array.h.
Referenced by Util::Distribution::reduce(), and Pscf::LuSolver::solve().

12.92.3.7 cArray() [2/2] template<typename Data >

```
const Data* Util::Array< Data >::cArray ( ) const
```

Return pointer to const to underlying C array.

12.92.4 Member Data Documentation

12.92.4.1 data_ template<typename Data >

```
Data* Util::Array< Data >::data_  [protected]
```

Pointer to an array of Data elements.
Definition at line 103 of file Array.h.
Referenced by Util::RArray< Data >::associate(), Util::DArray< Pscf::Monomer >::DArray(), and Util::RArray< Data >::RArray().

12.92.4.2 capacity_ template<typename Data >

```
int Util::Array< Data >::capacity_  [protected]
```

Allocated size of the data_ array.
Definition at line 106 of file Array.h.
Referenced by Util::RArray< Data >::associate(), Util::DArray< Pscf::Monomer >::DArray(), Util::DArray< Pscf::↩ Monomer >::operator=(), and Util::RArray< Data >::RArray().
The documentation for this class was generated from the following files:

- AutoCorrArray.h
- Array.h

12.93 Util::ArrayIterator< Data > Class Template Reference

Forward iterator for an Array or a C array.

`#include <ArrayIterator.h>`

Public Member Functions

- ArrayIterator ()
  Default constructor.
- void setCurrent (Data ∗ptr)
  Set the current pointer value.
- void setEnd (Data ∗ptr)
  Set the value of the end pointer.
- bool isEnd () const
  Has the end of the array been reached?
- bool notEnd () const
  Is the current pointer not at the end of the array?
- Data ∗ get () const
  Return a pointer to the current data.

Operators

- Data & operator∗ () const
  Get a reference to the current Data.
- Data ∗ operator-> () const
  Provide a pointer to the current Data object.
- ArrayIterator< Data > & operator++ ()
  Increment the current pointer.

12.93.1 Detailed Description

template<typename Data>
class Util::ArrayIterator< Data >

Forward iterator for an Array or a C array.
An ArrayIterator is an abstraction of a pointer, similar to an STL forward iterator. The ∗ operator returns a reference to an associated Data object, the -> operator returns a pointer to that object. The ++ operator increments the current pointer by one array element.
Unlike an STL forward iterator, an ArrayIterator contains the address of the end of the array. The isEnd() method can be used to test for termination of a for or while loop. When isEnd() is true, the current pointer is one past the end of the array, and thus the iterator has no current value, and cannot be incremented further.
An ArrayIterator behave like a pointer to non-const data, and provides read-write access to the objects to which it points.
A ConstArrayIterator behaves like a pointer to const, and provides read-only access
Definition at line 39 of file ArrayIterator.h.

12.93.2 Constructor & Destructor Documentation

12.93.2.1 ArrayIterator() `template<typename Data >`

`Util::ArrayIterator< Data >::ArrayIterator ( ) [inline]`

Default constructor.

Constructs an uninitialized iterator.

Definition at line 49 of file ArrayIterator.h.

12.93.3 Member Function Documentation

12.93.3.1 setCurrent() `template<typename Data >`

```
void Util::ArrayIterator< Data >::setCurrent (
            Data * ptr ) [inline]
```

Set the current pointer value.

Parameters

ptr	Pointer to current element of the array.

Definition at line 59 of file ArrayIterator.h.

Referenced by Util::Array< Pscf::Monomer >::begin(), Util::FSArray< double, 6 >::begin(), Util::FArray< DPropagator, 2 >::begin(), Util::GArray< Rational >::begin(), and Util::DSArray< Data >::begin().

12.93.3.2 setEnd() `template<typename Data >`

```
void Util::ArrayIterator< Data >::setEnd (
            Data * ptr ) [inline]
```

Set the value of the end pointer.

Parameters

ptr	Pointer to one element past end of array.

Definition at line 67 of file ArrayIterator.h.

Referenced by Util::Array< Pscf::Monomer >::begin(), Util::FSArray< double, 6 >::begin(), Util::FArray< DPropagator, 2 >::begin(), Util::GArray< Rational >::begin(), and Util::DSArray< Data >::begin().

12.93.3.3 isEnd() `template<typename Data >`

`bool Util::ArrayIterator< Data >::isEnd ( ) const [inline]`

Has the end of the array been reached?

Returns

true if at end, false otherwise.

Definition at line 75 of file ArrayIterator.h.

12.93.3.4 notEnd() `template<typename Data >`

`bool Util::ArrayIterator< Data >::notEnd ( ) const [inline]`

Is the current pointer not at the end of the array?

Returns

true if not at end, false otherwise.

Definition at line 83 of file ArrayIterator.h.

12.93.3.5 get() `template<typename Data >`

`Data* Util::ArrayIterator< Data >::get ( ) const [inline]`

Return a pointer to the current data.

Returns

true if at end, false otherwise.

Definition at line 91 of file ArrayIterator.h.

12.93.3.6 operator*() `template<typename Data >`

`Data& Util::ArrayIterator< Data >::operator* ( ) const [inline]`

Get a reference to the current Data.

Returns

reference to associated Data object

Definition at line 102 of file ArrayIterator.h.

12.93.3.7 operator->() `template<typename Data >`

`Data* Util::ArrayIterator< Data >::operator-> ( ) const [inline]`

Provide a pointer to the current Data object.

Returns

const pointer to the Data object

Definition at line 110 of file ArrayIterator.h.

12.93.3.8 operator++() `template<typename Data >`

`ArrayIterator<Data>& Util::ArrayIterator< Data >::operator++ ( ) [inline]`

Increment the current pointer.

Returns

this ArrayIterator, after modification.

Definition at line 118 of file ArrayIterator.h.
The documentation for this class was generated from the following file:

- ArrayIterator.h

12.94 Util::ArraySet< Data > Class Template Reference

A container for pointers to a subset of elements of an associated array.

`#include <ArraySet.h>`

Inheritance diagram for Util::ArraySet< Data >:

classUtil_1_1ArraySet-eps-converted-to.pdf

Public Member Functions

- ArraySet ()

  Constructor.
- virtual ∼ArraySet ()

  Destructor.
- void allocate (const Data ∗array, int capacity)

  Associate with a C array and allocate required memory.
- void allocate (const Array< Data > &array)

  Associate with an Array container and allocate required memory.

Mutators

- void append (Data &data)

  Append an element to the set.
- void remove (const Data &data)

  Remove an element from the set.
- Data & pop ()

  Pop the topmost from the set.
- void clear ()

  Reset to empty state.

Accessors

- int index (const Data &data) const

  Return the current index of an element within the set, if any.
- bool isAllocated () const

  Return true if the ArraySet is initialized, false otherwise.
- bool isValid () const

  Return true if the ArraySet is valid, or throw an exception.
- void dump () const

  Write the internal state of the ArraySet to std::cout.

Additional Inherited Members

12.94.1 Detailed Description

template< typename Data >
class Util::ArraySet< Data >

A container for pointers to a subset of elements of an associated array.
An ArraySet is a PArray that stores pointers to a subset of the elements of an associated Array container or bare C array. Pointers to the elements of this set are stored in a contiguous sequence, with indices in the range 0, ..., size() - 1. The order in which these pointers are stored is mutable, and generally changes whenever an element is removed.
The append() method appends a pointer to a new element to the end of the sequence and increments the size. The remove() method removes a specified element, then moves the pointer of the last element to the space vacated by the

removed element (unless the removed element was the last in the sequence), and decrements the size. The order in which the remaining elements of an ArraySet are stored thus can change whenever an element is removed.
An ArraySet provides O(N) sequential access to all elements of a set, O(1) insertion and deletion, and O(1) access to a randomly chosen element.
Definition at line 46 of file ArraySet.h.

12.94.2 Constructor & Destructor Documentation

12.94.2.1 ArraySet() `template<typename Data >`

`Util::ArraySet< Data >::ArraySet`

Constructor.
Definition at line 187 of file ArraySet.h.

12.94.2.2 ∼ArraySet() `template<typename Data >`

`Util::ArraySet< Data >::~ArraySet [virtual]`

Destructor.
Definition at line 196 of file ArraySet.h.

12.94.3 Member Function Documentation

12.94.3.1 allocate() [1/2] `template<typename Data >`

```
void Util::ArraySet< Data >::allocate (
            const Data * array,
            int capacity )
```

Associate with a C array and allocate required memory.
This method associates an ArraySet with a bare C array, and allocates all memory required by the ArraySet.
An ArraySet may only be allocated once. This method throws an Exception if it is called more than once.

Parameters

array	associated C array of Data objects
capacity	number of elements in the array

Definition at line 210 of file ArraySet.h.
References UTIL_THROW.

12.94.3.2 allocate() [2/2] `template<typename Data >`

```
void Util::ArraySet< Data >::allocate (
            const Array< Data > & array )
```

Associate with an Array container and allocate required memory.
Invokes allocate(&array[0], array.capacity()) internally.

Parameters

array	associated Array<Data> container

Definition at line 237 of file ArraySet.h.

References Util::Array< Data >::capacity().

12.94.3.3 append() template<typename Data >

void Util::ArraySet< Data >::append (
Data & *data*)

Append an element to the set.

This appends a new element to the end of the sequence. This does not change the order of other elements.

Parameters

data	array element to be added.

Definition at line 244 of file ArraySet.h.

References UTIL_THROW.

12.94.3.4 remove() template<typename Data >

void Util::ArraySet< Data >::remove (
const Data & *data*)

Remove an element from the set.

Removal of an element generally changes the order of the remaining elements.

Throws an Exception if data is not in this ArraySet.

Parameters

data	array element to be added.

Definition at line 268 of file ArraySet.h.

References UTIL_THROW.

12.94.3.5 pop() template<typename Data >

Data & Util::ArraySet< Data >::pop

Pop the topmost from the set.

Popping the top element does not change the order of the remaining elements.

Definition at line 301 of file ArraySet.h.

References UTIL_THROW.

12.94.3.6 clear() template<typename Data >

void Util::ArraySet< Data >::clear

Reset to empty state.

Definition at line 319 of file ArraySet.h.

12.94.3.7 index() template<typename Data >

int Util::ArraySet< Data >::index (
const Data & *data*) const

Return the current index of an element within the set, if any.

Return the current index of an element within the set, or return a negative value -1 if the element is not in the set.

This method returns the current index of the pointer to object data within this ArraySet, in the range 0 < index < size() -1. The method returns -1 if data is an element of the associated array but is not in the ArraySet.

Throws an exception if data is not in the associated array.

Parameters

data	array element of interest.

Returns

current index of pointer to element within this ArraySet.

Definition at line 335 of file ArraySet.h.
References UTIL_THROW.

12.94.3.8 isAllocated() `template<typename Data >`

`bool Util::ArraySet< Data >::isAllocated [inline]`
Return true if the ArraySet is initialized, false otherwise.
Definition at line 348 of file ArraySet.h.

12.94.3.9 isValid() `template<typename Data >`

`bool Util::ArraySet< Data >::isValid`
Return true if the ArraySet is valid, or throw an exception.
Definition at line 355 of file ArraySet.h.
References UTIL_THROW.

12.94.3.10 dump() `template<typename Data >`

`void Util::ArraySet< Data >::dump ( ) const`
Write the internal state of the ArraySet to std::cout.
The documentation for this class was generated from the following file:

- ArraySet.h

12.95 Util::ArrayStack< Data > Class Template Reference

A stack of fixed capacity.
`#include <ArrayStack.h>`

Public Member Functions

- ArrayStack ()
  Default constructor.
- virtual ∼ArrayStack ()
  Destructor.
- void allocate (int capacity)
  Initialize and allocate required memory.

Mutators

- void push (Data &data)
  Push an element onto the Stack.
- Data & pop ()
  Pop an element off the stack.

Accessors

- int capacity () const

  Return capacity of the underlying array.
- int size () const

  Get the number of elements in the stack.
- Data & peek ()

  Return a reference to the top element (don't pop).
- const Data & peek () const

  Return a const ref to the top element (don't pop).
- bool isValid () const

  Return true if the ArrayStack is valid, or throw an exception.
- bool isAllocated () const

  Return true only if the ArrayStack has been allocated.

12.95.1 Detailed Description

template<typename Data>
class Util::ArrayStack< Data >

A stack of fixed capacity.
Pointers to elements are stored in an allocatable, non-resizable array.
Definition at line 25 of file ArrayStack.h.

12.95.2 Constructor & Destructor Documentation

12.95.2.1 ArrayStack() `template<typename Data >`

`Util::ArrayStack< Data >::ArrayStack`
Default constructor.
Definition at line 126 of file ArrayStack.h.

12.95.2.2 ∼ArrayStack() `template<typename Data >`

`Util::ArrayStack< Data >::~ArrayStack [virtual]`
Destructor.
Definition at line 136 of file ArrayStack.h.

12.95.3 Member Function Documentation

12.95.3.1 allocate() `template<typename Data >`

```
void Util::ArrayStack< Data >::allocate (
            int capacity )
```
Initialize and allocate required memory.

Parameters

capacity	maximum size of stack.

Definition at line 147 of file ArrayStack.h.

References UTIL_THROW.

12.95.3.2 push() template<typename Data >

```
void Util::ArrayStack< Data >::push (
            Data & data )
```

Push an element onto the Stack.

Parameters

data	element to be added to stack.

Definition at line 183 of file ArrayStack.h.
References UTIL_THROW.

12.95.3.3 pop() template<typename Data >

```
Data & Util::ArrayStack< Data >::pop
```

Pop an element off the stack.

Returns

the top element (which is popped off stack).

Definition at line 196 of file ArrayStack.h.
References UTIL_THROW.

12.95.3.4 capacity() template<typename Data >

```
int Util::ArrayStack< Data >::capacity
```

Return capacity of the underlying array.

Returns

Number of elements allocated in array.

Definition at line 169 of file ArrayStack.h.

12.95.3.5 size() template<typename Data >

```
int Util::ArrayStack< Data >::size  [inline]
```

Get the number of elements in the stack.
Definition at line 176 of file ArrayStack.h.

12.95.3.6 peek() [1/2] template<typename Data >

```
const Data & Util::ArrayStack< Data >::peek  [inline]
```

Return a reference to the top element (don't pop).
Definition at line 211 of file ArrayStack.h.

12.95.3.7 peek() [2/2] template<typename Data >

```
const Data& Util::ArrayStack< Data >::peek ( ) const
```

Return a const ref to the top element (don't pop).

12.95.3.8 isValid() template<typename Data >

bool Util::ArrayStack< Data >::isValid

Return true if the ArrayStack is valid, or throw an exception.

Definition at line 225 of file ArrayStack.h.

References UTIL_THROW.

12.95.3.9 isAllocated() template<typename Data >

bool Util::ArrayStack< Data >::isAllocated [inline]

Return true only if the ArrayStack has been allocated.

Definition at line 261 of file ArrayStack.h.

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

- ArrayStack.h

12.96 Util::AutoCorr< Data, Product > Class Template Reference

Auto-correlation function for one sequence of Data values.

#include <AutoCorr.h>

Inheritance diagram for Util::AutoCorr< Data, Product >:

classUtil_1_1AutoCorr-eps-converted-to.pdf

Public Member Functions

- AutoCorr ()

  Constructor.
- ~AutoCorr ()

  Destructor.
- void clear ()

  Reset to empty state.
- void readParameters (std::istream &in)

  Read buffer capacity, allocate memory and initialize.
- void setParam (int bufferCapacity)

  Set buffer capacity, allocate memory and initialize.
- virtual void loadParameters (Serializable::IArchive &ar)

  Load state from an archive.
- virtual void save (Serializable::OArchive &ar)

  Save state to an archive.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize to/from an archive.

- void sample (Data value)

    Sample a value.
- void output (std::ostream &out)

    Output the autocorrelation function.
- int bufferCapacity () const

    Return capacity of history buffer.
- int nSample () const

    Return number of values sampled thus far.
- Data average () const

    Return average of all sampled values.
- double corrTime () const

    Numerical integration of autocorrelation function.
- Product autoCorrelation (int t) const

    Return autocorrelation at a given lag time.

Additional Inherited Members

12.96.1 Detailed Description

template<typename Data, typename Product>
class Util::AutoCorr< Data, Product >

Auto-correlation function for one sequence of Data values.
This class calculates an autocorrelation function for a sequence x(i) of values of a variable or object of type Data. The resulting autocorrelation function is and array of values of type Product, where C(j) = <x(i-j), x(i)>. Here <A,B> denotes an inner product of type Product for objects A and B of type Data.
The meaning of the inner product is defined for various data types b the overloaded function Product product(Data, Data) that is defined for double, complex and Vector data in the product.h file.
The zero value for variables of type Data is returned by the overloaded function void setToZero(Data) method defined in the setToData.h file.
Definition at line 49 of file AutoCorr.h.

12.96.2 Constructor & Destructor Documentation

12.96.2.1 AutoCorr() `template<typename Data , typename Product >`
`Util::AutoCorr< Data, Product >::AutoCorr`
Constructor.
Definition at line 183 of file AutoCorr.h.
References Util::ParamComposite::setClassName(), and Util::setToZero().

12.96.2.2 ∼AutoCorr() `template<typename Data , typename Product >`
`Util::AutoCorr< Data, Product >::∼AutoCorr`
Destructor.
Definition at line 198 of file AutoCorr.h.

12.96.3 Member Function Documentation

12.96.3.1 clear() template<typename Data , typename Product >

void Util::AutoCorr< Data, Product >::clear

Reset to empty state.

Definition at line 247 of file AutoCorr.h.

References Util::setToZero().

12.96.3.2 readParameters() template<typename Data , typename Product >

void Util::AutoCorr< Data, Product >::readParameters (std::istream & *in*) [virtual]

Read buffer capacity, allocate memory and initialize.

Parameters

in	input parameter stream.

Reimplemented from Util::ParamComposite.

Definition at line 205 of file AutoCorr.h.

12.96.3.3 setParam() template<typename Data , typename Product >

void Util::AutoCorr< Data, Product >::setParam (int *bufferCapacity*)

Set buffer capacity, allocate memory and initialize.

Parameters

bufferCapacity	maximum number of values in history buffer.

Definition at line 215 of file AutoCorr.h.

12.96.3.4 loadParameters() template<typename Data , typename Product >

void Util::AutoCorr< Data, Product >::loadParameters (Serializable::IArchive & *ar*) [virtual]

Load state from an archive.

Parameters

ar	binary loading (input) archive.

Reimplemented from Util::ParamComposite.

Definition at line 225 of file AutoCorr.h.

12.96.3.5 save() template<typename Data , typename Product >

void Util::AutoCorr< Data, Product >::save (Serializable::OArchive & *ar*) [virtual]

Save state to an archive.

Parameters

ar	binary saving (output) archive.

Reimplemented from Util::ParamComposite.
Definition at line 240 of file AutoCorr.h.

12.96.3.6 serialize() `template<typename Data , typename Product >`

```
template<class Archive >
void Util::AutoCorr< Data, Product >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize to/from an archive.

Parameters

ar	archive
version	archive version id

Definition at line 419 of file AutoCorr.h.

12.96.3.7 sample() `template<typename Data , typename Product >`

```
void Util::AutoCorr< Data, Product >::sample (
            Data value )
```

Sample a value.

Parameters

value	current value

Definition at line 295 of file AutoCorr.h.
References Util::product().

12.96.3.8 output() `template<typename Data , typename Product >`

```
void Util::AutoCorr< Data, Product >::output (
            std::ostream & out )
```

Output the autocorrelation function.

Parameters

out	output stream.

Definition at line 335 of file AutoCorr.h.
References Util::product().

12.96.3.9 bufferCapacity() `template<typename Data , typename Product >`

```
int Util::AutoCorr< Data, Product >::bufferCapacity
```

Return capacity of history buffer.
Definition at line 310 of file AutoCorr.h.

12.96.3.10 nSample() `template<typename Data , typename Product >`

int Util::AutoCorr< Data, Product >::nSample
Return number of values sampled thus far.
Definition at line 317 of file AutoCorr.h.

12.96.3.11 average() template<typename Data , typename Product >

Data Util::AutoCorr< Data, Product >::average
Return average of all sampled values.
Definition at line 324 of file AutoCorr.h.

12.96.3.12 corrTime() template<typename Data , typename Product >

double Util::AutoCorr< Data, Product >::corrTime
Numerical integration of autocorrelation function.
Definition at line 361 of file AutoCorr.h.
References Util::product(), and Util::setToZero().

12.96.3.13 autoCorrelation() template<typename Data , typename Product >

Product Util::AutoCorr< Data, Product >::autoCorrelation (
int t) const
Return autocorrelation at a given lag time.

Parameters

t	the lag time

Definition at line 395 of file AutoCorr.h.
References Util::product().
The documentation for this class was generated from the following file:

- AutoCorr.h

12.97 Util::AutoCorrArray< Data, Product > Class Template Reference

Auto-correlation function for an ensemble of sequences.
#include <AutoCorrArray.h>
Inheritance diagram for Util::AutoCorrArray< Data, Product >:

classUtil_1_1AutoCorrArray-eps-converted-to.pdf

Public Member Functions

- AutoCorrArray ()

Default constructor.

- ∼AutoCorrArray ()

  Default destructor.

- virtual void readParameters (std::istream &in)

  Read parameters, allocate memory and clear history.

- void setParam (int ensembleCapacity, int bufferCapacity)

  Allocate memory, and clear history.

- virtual void loadParameters (Serializable::IArchive &ar)

  Load internal state from an archive.

- virtual void save (Serializable::OArchive &ar)

  Save internal state to an archive.

- void setNEnsemble (int nEnsemble)

  Set actual number of sequences in ensemble.

- void clear ()

  Reset to empty state.

- void sample (const Array< Data > &values)

  Sample an array of current values.

- void output (std::ostream &out)

  Output the autocorrelation function.

- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize this AutoCorrArray to/from an archive.

- int bufferCapacity () const

  Return maximum number of samples in history for each sequence.

- int nEnsemble () const

  Return nEnsemble.

- int nSample () const

  Return the total number of samples per sequence thus far.

- Data average () const

  Return average of sampled values.

- double corrTime () const

  Numerical integral of autocorrelation function.

Additional Inherited Members

12.97.1 Detailed Description

template<typename Data, typename Product>
class Util::AutoCorrArray< Data, Product >

Auto-correlation function for an ensemble of sequences.

This class calculates an autocorrelation function for a ensemble of statistically equivalent sequences x(i) of values of a variable of type Data. The resulting autocorrelation function is an array of values of type Product, where C(j) = <x(i-j), x(i)>. Here <A,B> denotes an inner product of type Product for objects A and B of type Data.

The meaning of <A,B> for two Data values is defined for various data types by the overloaded functions product(Data, Data) defined in file "product.h" . These functions define a product as an arithmetic product for floating point numbers, and use the following definitions for complex numbers and Vector objects:

```
double  product(double, double)   = A*B
complex product(complex, complex) = conjug(A)*B
double  product(Vector,  Vector)  = A.dot(B)
```

The meaning of setting a variable to zero is defined for various types of data by the overloaded functions setToZero(←
Data&) that are defined in file setToZero.h.

Definition at line 57 of file AutoCorrArray.h.

12.97.2 Constructor & Destructor Documentation

12.97.2.1 AutoCorrArray() `template<typename Data , typename Product >`

`Util::AutoCorrArray< Data, Product >::AutoCorrArray`

Default constructor.

Definition at line 205 of file AutoCorrArray.h.

References Util::ParamComposite::setClassName(), and Util::setToZero().

12.97.2.2 ∼AutoCorrArray() `template<typename Data , typename Product >`

`Util::AutoCorrArray< Data, Product >::∼AutoCorrArray`

Default destructor.

Definition at line 222 of file AutoCorrArray.h.

12.97.3 Member Function Documentation

12.97.3.1 readParameters() `template<typename Data , typename Product >`

```
void Util::AutoCorrArray< Data, Product >::readParameters (
            std::istream & in ) [virtual]
```

Read parameters, allocate memory and clear history.

Reads parameters ensembleCapacity and bufferCapacity, allocates memory, sets nEnsemble=ensembleCapacity, and calls clear().

Parameters

in	input parameter stream

Reimplemented from Util::ParamComposite.

Definition at line 229 of file AutoCorrArray.h.

12.97.3.2 setParam() `template<typename Data , typename Product >`

```
void Util::AutoCorrArray< Data, Product >::setParam (
            int ensembleCapacity,
            int bufferCapacity )
```

Allocate memory, and clear history.

Sets parameters ensembleCapacity and bufferCapacity, allocates memory, sets nEnsemble=ensembleCapacity, and calls clear().

Parameters

ensembleCapacity	maximum number of sequences in ensemble
bufferCapacity	maximum number of values in each history

Definition at line 241 of file AutoCorrArray.h.

12.97.3.3 loadParameters() `template<typename Data , typename Product >`

```
void Util::AutoCorrArray< Data, Product >::loadParameters (
```

```
            Serializable::IArchive & ar )  [virtual]
```

Load internal state from an archive.

Parameters

ar	input/loading archive

Reimplemented from Util::ParamComposite.
Definition at line 253 of file AutoCorrArray.h.

12.97.3.4 save() `template<typename Data , typename Product >`

```
void Util::AutoCorrArray< Data, Product >::save (
            Serializable::OArchive & ar )  [virtual]
```

Save internal state to an archive.

Parameters

ar	output/saving archive

Reimplemented from Util::ParamComposite.
Definition at line 269 of file AutoCorrArray.h.

12.97.3.5 setNEnsemble() `template<typename Data , typename Product >`

```
void Util::AutoCorrArray< Data, Product >::setNEnsemble (
            int nEnsemble )
```

Set actual number of sequences in ensemble.

Precondition

readParam() or setParam() must have been called previously

nEnsemble <= ensembleCapacity

Parameters

nEnsemble	actual number of sequences in ensemble

Definition at line 276 of file AutoCorrArray.h.
References UTIL_THROW.

12.97.3.6 clear() `template<typename Data , typename Product >`

```
void Util::AutoCorrArray< Data, Product >::clear
```

Reset to empty state.
Definition at line 291 of file AutoCorrArray.h.
References Util::setToZero().

12.97.3.7 sample() `template<typename Data , typename Product >`

```
void Util::AutoCorrArray< Data, Product >::sample (
            const Array< Data > & values )
```

Sample an array of current values.

Parameters

values	Array of current values

Definition at line 331 of file AutoCorrArray.h.
References Util::product().

12.97.3.8 output()

```
template<typename Data , typename Product >
void Util::AutoCorrArray< Data, Product >::output (
    std::ostream & out )
```

Output the autocorrelation function.
Definition at line 386 of file AutoCorrArray.h.

12.97.3.9 serialize()

```
template<typename Data , typename Product >
template<class Archive >
void Util::AutoCorrArray< Data, Product >::serialize (
    Archive & ar,
    const unsigned int version )
```

Serialize this AutoCorrArray to/from an archive.

Parameters

ar	input or output archive
version	file version id

Definition at line 436 of file AutoCorrArray.h.

12.97.3.10 bufferCapacity()

```
template<typename Data , typename Product >
int Util::AutoCorrArray< Data, Product >::bufferCapacity
```

Return maximum number of samples in history for each sequence.
Definition at line 354 of file AutoCorrArray.h.

12.97.3.11 nEnsemble()

```
template<typename Data , typename Product >
int Util::AutoCorrArray< Data, Product >::nEnsemble
```

Return nEnsemble.
Definition at line 361 of file AutoCorrArray.h.

12.97.3.12 nSample()

```
template<typename Data , typename Product >
int Util::AutoCorrArray< Data, Product >::nSample
```

Return the total number of samples per sequence thus far.
Definition at line 368 of file AutoCorrArray.h.

12.97.3.13 average()

```
template<typename Data , typename Product >
Data Util::AutoCorrArray< Data, Product >::average
```

Return average of sampled values.
Definition at line 375 of file AutoCorrArray.h.

12.97.3.14 corrTime() `template<typename Data , typename Product >`

`double Util::AutoCorrArray< Data, Product >::corrTime`

Numerical integral of autocorrelation function.
Definition at line 407 of file AutoCorrArray.h.
References Util::product(), and Util::setToZero().
The documentation for this class was generated from the following file:

- AutoCorrArray.h

12.98 Util::AutoCorrelation< Data, Product > Class Template Reference

Auto-correlation function, using hierarchical algorithm.

`#include <AutoCorrelation.h>`

Inheritance diagram for Util::AutoCorrelation< Data, Product >:

classUtil_1_1AutoCorrelation-eps-converted-to.pdf

Public Member Functions

- AutoCorrelation ()
  Constructor.
- virtual void readParameters (std::istream &in)
  Read parameters from file and initialize.
- virtual void load (Serializable::IArchive &ar)
  Load internal state from an archive.
- virtual void save (Serializable::OArchive &ar)
  Save internal state to an archive.
- int maxDelay () const
  Return maximum delay, in primary samples.

Protected Member Functions

- virtual void registerDescendant (AutoCorrStage< Data, Product > *ptr)
  Register a descendant stage.

Additional Inherited Members

12.98.1 Detailed Description

template<typename Data, typename Product>
class Util::AutoCorrelation< Data, Product >

Auto-correlation function, using hierarchical algorithm.

This class represents the primary stage of a linked list of AutoCorrStage objects that implement a hierarchical blocking algorithm for an auto-correlation function.
Definition at line 29 of file AutoCorrelation.h.

12.98.2 Constructor & Destructor Documentation

12.98.2.1 AutoCorrelation() `template<typename Data , typename Product >`

`Util::AutoCorrelation< Data, Product >::AutoCorrelation`

Constructor.
Definition at line 23 of file AutoCorrelation.tpp.

12.98.3 Member Function Documentation

12.98.3.1 readParameters() `template<typename Data , typename Product >`

```
void Util::AutoCorrelation< Data, Product >::readParameters (
            std::istream & in ) [virtual]
```

Read parameters from file and initialize.

Parameters

in	input parameter file

Reimplemented from Util::ParamComposite.
Definition at line 33 of file AutoCorrelation.tpp.

12.98.3.2 load() `template<typename Data , typename Product >`

```
void Util::AutoCorrelation< Data, Product >::load (
            Serializable::IArchive & ar ) [virtual]
```

Load internal state from an archive.

Parameters

ar	input/loading archive

Reimplemented from Util::ParamComposite.
Definition at line 45 of file AutoCorrelation.tpp.

12.98.3.3 save() `template<typename Data , typename Product >`

```
void Util::AutoCorrelation< Data, Product >::save (
            Serializable::OArchive & ar ) [virtual]
```

Save internal state to an archive.

Parameters

ar	output/saving archive

Reimplemented from Util::ParamComposite.

Definition at line 57 of file AutoCorrelation.tpp.

12.98.3.4 maxDelay() `template<typename Data , typename Product >`

`int Util::AutoCorrelation< Data, Product >::maxDelay`

Return maximum delay, in primary samples.
Definition at line 64 of file AutoCorrelation.tpp.
References Util::AutoCorrStage< Data, Product >::bufferSize(), and Util::AutoCorrStage< Data, Product >::stage↵ Interval().

12.98.3.5 registerDescendant() `template<typename Data , typename Product >`

`void Util::AutoCorrelation< Data, Product >::registerDescendant ( AutoCorrStage< Data, Product > * ptr ) [protected], [virtual]`

Register a descendant stage.
This should be called only by a root stage.

Parameters

ptr	pointer to a descendant AutoCorrelation.

Reimplemented from Util::AutoCorrStage< Data, Product >.
Definition at line 79 of file AutoCorrelation.tpp.
The documentation for this class was generated from the following files:

- AutoCorrelation.h
- AutoCorrelation.tpp

12.99 Util::AutoCorrStage< Data, Product > Class Template Reference

Hierarchical auto-correlation function algorithm.
`#include <AutoCorrStage.h>`
Inheritance diagram for Util::AutoCorrStage< Data, Product >:

classUtil_1_1AutoCorrStage-eps-converted-to.pdf

Public Member Functions

- AutoCorrStage ()

  Constructor.
- virtual ~AutoCorrStage ()

  Destructor.
- void setParam (int bufferCapacity=64, int maxStageId=0, int blockFactor=2)

  Set all parameters and allocate to initialize state.
- virtual void sample (Data value)

  Sample a value.
- void clear ()

  Clear accumulators and destroy descendants.

- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize to/from an archive.

Accessors

- int **bufferCapacity_**
- int maxStageId_

  Maximum allowed stage index (controls maximum degree of blocking).
- int blockFactor_

  Number of values per block (ratio of intervals for successive stages).
- void output (std::ostream &out)

  Output the autocorrelation function, assuming zero mean.
- void output (std::ostream &out, Product aveSq)

  Output the autocorrelation function.
- int bufferCapacity () const

  Return capacity of history buffer.
- int bufferSize () const

  Return current size of history buffer.
- long nSample () const

  Return the number of sampled values.
- long stageInterval () const

  Return the number of primary values per block at this stage.
- Product autoCorrelation (int t) const

  Return autocorrelation at a given time, assuming zero average.
- Product autoCorrelation (int t, Product aveSq) const

  Return autocorrelation at a given lag time.
- double corrTime () const

  Estimate of autocorrelation time, in samples.
- double corrTime (Product aveSq) const

  Numerical integration of autocorrelation function.
- void allocate ()

  Allocate memory and initialize to empty state.
- bool hasChild () const

  Does this have a child AutoCorrStage?
- AutoCorrStage & child ()

  Return the child AutoCorrStage by reference.
- virtual void registerDescendant (AutoCorrStage< Data, Product > *ptr)

  Register the creation of a descendant stage.
- template<class Archive >
  void serializePrivate (Archive &ar, const unsigned int version)

  Serialize private data members, and descendants.

12.99.1 Detailed Description

template<typename Data, typename Product>
class Util::AutoCorrStage< Data, Product >

Hierarchical auto-correlation function algorithm.

This class calculates an autocorrelation function for a sequence x(i) of values of a variable or object of type Data. The resulting autocorrelation function is and array of values of type Product, where C(j) = <x(i-j), x(i)>. Here <A,B> denotes an inner product of type Product for objects A and B of type Data.

The meaning of the inner product is defined for various data types b the overloaded function Product product(Data, Data) that is defined for double, complex and Vector data in the product.h file.

The zero value for variables of type Data is returned by the overloaded function void setToZero(Data) method defined in the setToData.h file.

This class implements a hierarchical algorithm to calculate C(j). The algorithm is implemented by a linked list of AutoCorrStage objects. Each object in this list is assigned an integer chainId.

The "primary" AutoCorrStage object in this list, with chainId=0, calculates the autocorrelation for a primary sequence of primary Data values that are passed to the sample method of this object. For each n > 0, the object with chainId = n calculates the autocorrelation function for a sequence of values in which each value is an average of a block of block↩
Factor∗∗n consecutive values of the primary sequence or, equivalently, an average of blockFactor consecutive values of the sequence maintained by the parent object with chainId = n-1. Additional stages are added to this list dynamically as needed.

Definition at line 53 of file AutoCorrStage.h.

12.99.2 Constructor & Destructor Documentation

12.99.2.1 AutoCorrStage() `template<typename Data , typename Product >`

`Util::AutoCorrStage< Data, Product >::AutoCorrStage`

Constructor.

This constructor creates a primary AutoCorrStage object with stageId = 0 and stageInterval = 1. A private constructor is used to recursively create descendant stages as needed.

Definition at line 30 of file AutoCorrStage.tpp.

References Util::setToZero().

12.99.2.2 ∼AutoCorrStage() `template<typename Data , typename Product >`

`Util::AutoCorrStage< Data, Product >::~AutoCorrStage [virtual]`

Destructor.

Recursively destroy all descendant stages.

Definition at line 79 of file AutoCorrStage.tpp.

12.99.3 Member Function Documentation

12.99.3.1 setParam() `template<typename Data , typename Product >`

```
void Util::AutoCorrStage< Data, Product >::setParam (
            int bufferCapacity = 64,
            int maxStageId = 0,
            int blockFactor = 2 )
```

Set all parameters and allocate to initialize state.

Parameters

bufferCapacity	max. number of values stored in buffer
maxStageId	maximum stage index (0=primary)
blockFactor	ratio of block sizes of subsequent stages

Definition at line 90 of file AutoCorrStage.tpp.

12.99.3.2 sample() `template<typename Data , typename Product >`

```
void Util::AutoCorrStage< Data, Product >::sample (
            Data value ) [virtual]
```

Sample a value.

Parameters

value	current Data value

Definition at line 121 of file AutoCorrStage.tpp.
References Util::product(), and Util::setToZero().

12.99.3.3 clear() `template<typename Data , typename Product >`

```
void Util::AutoCorrStage< Data, Product >::clear
```

Clear accumulators and destroy descendants.
Definition at line 103 of file AutoCorrStage.tpp.
References Util::setToZero().
Referenced by Util::AutoCorrStage< Data, Product >::allocate().

12.99.3.4 serialize() `template<typename Data , typename Product >`

```
template<class Archive >
void Util::AutoCorrStage< Data, Product >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize to/from an archive.

Parameters

ar	archive
version	archive version id

Definition at line 168 of file AutoCorrStage.tpp.

12.99.3.5 output() `[1/2]` `template<typename Data , typename Product >`

```
void Util::AutoCorrStage< Data, Product >::output (
            std::ostream & out )
```

Output the autocorrelation function, assuming zero mean.
This calls output(std::ostream out, Product aveSq) with a zero value for aveSq.

Parameters

out	output stream.

Definition at line 259 of file AutoCorrStage.tpp.
References Util::setToZero().

12.99.3.6 output() [2/2] `template<typename Data , typename Product >`

```
void Util::AutoCorrStage< Data, Product >::output (
            std::ostream & out,
            Product aveSq )
```

Output the autocorrelation function.
The parameter avSq = ave(x)^2 is subtracted from the correlation function ave(x(t)x(0)).

Parameters

out	output stream
aveSq	square of ave(x)

Definition at line 270 of file AutoCorrStage.tpp.

12.99.3.7 bufferCapacity() `template<typename Data , typename Product >`

```
int Util::AutoCorrStage< Data, Product >::bufferCapacity
```

Return capacity of history buffer.
Definition at line 231 of file AutoCorrStage.tpp.

12.99.3.8 bufferSize() `template<typename Data , typename Product >`

```
int Util::AutoCorrStage< Data, Product >::bufferSize
```

Return current size of history buffer.
Definition at line 238 of file AutoCorrStage.tpp.
Referenced by Util::AutoCorrelation< Data, Product >::maxDelay().

12.99.3.9 nSample() `template<typename Data , typename Product >`

```
long Util::AutoCorrStage< Data, Product >::nSample
```

Return the number of sampled values.
Definition at line 245 of file AutoCorrStage.tpp.

12.99.3.10 stageInterval() `template<typename Data , typename Product >`

```
long Util::AutoCorrStage< Data, Product >::stageInterval
```

Return the number of primary values per block at this stage.
Definition at line 252 of file AutoCorrStage.tpp.
Referenced by Util::AutoCorrelation< Data, Product >::maxDelay().

12.99.3.11 autoCorrelation() [1/2] `template<typename Data , typename Product >`

```
Product Util::AutoCorrStage< Data, Product >::autoCorrelation (
            int t ) const
```

Return autocorrelation at a given time, assuming zero average.
This calls autoCorrelations(t, aveSq) with a zero value for for aveSq.

Parameters

t	the lag time, in Data samples

Definition at line 296 of file AutoCorrStage.tpp.
References Util::setToZero().

12.99.3.12 autoCorrelation() [2/2]

```
template<typename Data , typename Product >
Product Util::AutoCorrStage< Data, Product >::autoCorrelation (
            int t,
            Product aveSq ) const
```

Return autocorrelation at a given lag time.
The parameter aveSq is subtracted from ave(x(t)x(0)) in output.

Parameters

t	the lag time, in Data samples
aveSq	square ave(x(t))

Definition at line 307 of file AutoCorrStage.tpp.

12.99.3.13 corrTime() [1/2]

```
template<typename Data , typename Product >
double Util::AutoCorrStage< Data, Product >::corrTime
```

Estimate of autocorrelation time, in samples.
This variant assumes a zero average.
Definition at line 319 of file AutoCorrStage.tpp.
References Util::setToZero().

12.99.3.14 corrTime() [2/2]

```
template<typename Data , typename Product >
double Util::AutoCorrStage< Data, Product >::corrTime (
            Product aveSq ) const
```

Numerical integration of autocorrelation function.
This function returns the time integral of the autocorrelation function. The parameter aveSq is subtracted from ave(x(t)x(0)) in the integrand.

Parameters

aveSq	square ave(x(t))

Definition at line 330 of file AutoCorrStage.tpp.
References Util::setToZero().

12.99.3.15 allocate()

```
template<typename Data , typename Product >
void Util::AutoCorrStage< Data, Product >::allocate [protected]
```

Allocate memory and initialize to empty state.

Definition at line 357 of file AutoCorrStage.tpp.
References Util::AutoCorrStage< Data, Product >::clear().

12.99.3.16 hasChild() `template<typename Data , typename Product >`

`bool Util::AutoCorrStage< Data, Product >::hasChild [inline], [protected]`

Does this have a child AutoCorrStage?
Definition at line 305 of file AutoCorrStage.h.

12.99.3.17 child() `template<typename Data , typename Product >`

`AutoCorrStage< Data, Product > & Util::AutoCorrStage< Data, Product >::child [inline], [protected]`

Return the child AutoCorrStage by reference.
Definition at line 312 of file AutoCorrStage.h.

12.99.3.18 registerDescendant() `template<typename Data , typename Product >`

```
void Util::AutoCorrStage< Data, Product >::registerDescendant (
             AutoCorrStage< Data, Product > * ptr ) [protected], [virtual]
```

Register the creation of a descendant stage.
This should be called only by a root stage.

Parameters

ptr	pointer to a descendant AutoCorrStage.

Reimplemented in Util::AutoCorrelation< Data, Product >.
Definition at line 385 of file AutoCorrStage.tpp.

12.99.3.19 serializePrivate() `template<typename Data , typename Product >`

```
template<class Archive >
void Util::AutoCorrStage< Data, Product >::serializePrivate (
             Archive & ar,
             const unsigned int version ) [protected]
```

Serialize private data members, and descendants.

Parameters

ar	archive
version	archive version id

Definition at line 186 of file AutoCorrStage.tpp.

12.99.4 Member Data Documentation

12.99.4.1 maxStageId_ `template<typename Data , typename Product >`

`int Util::AutoCorrStage< Data, Product >::maxStageId_ [protected]`

Maximum allowed stage index (controls maximum degree of blocking).
Definition at line 196 of file AutoCorrStage.h.

12.99.4.2 blockFactor_ template<typename Data , typename Product > int Util::AutoCorrStage< Data, Product >::blockFactor_ [protected]

Number of values per block (ratio of intervals for successive stages).
Definition at line 199 of file AutoCorrStage.h.
The documentation for this class was generated from the following files:

- AutoCorrStage.h
- AutoCorrStage.tpp

12.100 Util::Average Class Reference

Calculates the average and variance of a sampled property.
`#include <Average.h>`
Inheritance diagram for Util::Average:

classUtil_1_1Average-eps-converted-to.pdf

Public Member Functions

- Average (int blockFactor=2)

  Constructor.
- virtual ∼Average ()

  Destructor.
- void readParameters (std::istream &in)

  Read parameter nSamplePerBlock from file and initialize.
- void setNSamplePerBlock (int nSamplePerBlock)

  Set nSamplePerBlock.
- virtual void loadParameters (Serializable::IArchive &ar)

  Load internal state from an archive.
- virtual void save (Serializable::OArchive &ar)

  Save internal state to an archive.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize this Average to or from an archive.
- void clear ()

  Clear all accumulators, set to empty initial state.
- void sample (double value)

  Add a sampled value to the ensemble.
- void sample (double value, std::ostream &out)

  Add a sampled value to ensemble, and output block averages.
- void output (std::ostream &out) const

  Output final statistical properties to file.

- double blockingError () const

  Return estimated error on average from blocking analysis.
- int nSamplePerBlock () const

  Get number of samples per block average.
- int iBlock () const

  Get number of samples in current block average.
- bool isBlockComplete () const

  Is the current block average complete?

Additional Inherited Members

12.100.1 Detailed Description

Calculates the average and variance of a sampled property.
Average calculates block and global averages of a sampled value and its square, from which it obtains a global average and variance for a sequence. A hierarchical blocking algorithm is used to estimate the error on the average. No error estimate is provided for the variance.
The sample function of also optionally calculates block averages, which can be useful for reducing how frequently values are logged to a file. The parameter nSamplePerBlock is the number of samples per block average. This is initialized to zero. A zero value disables calculation of block averages. An overloaded method of the sample function that takes an std::ostream file as an argument outputs block averages to file as blocks are completed.
The hierarchical blocking algorithm is implemented using a linked list of Util::AverageStage objects. See documentation of that class for further details, and a literature reference.
Definition at line 43 of file Average.h.

12.100.2 Constructor & Destructor Documentation

12.100.2.1 Average() `Util::Average::Average ( int blockFactor = 2 )`

Constructor.

Parameters

blockFactor	ratio of block sizes for subsequent stages.

Definition at line 20 of file Average.cpp.
References Util::ParamComposite::setClassName().

12.100.2.2 ∼Average() `Util::Average::∼Average ( ) [virtual]`

Destructor.
Definition at line 36 of file Average.cpp.

12.100.3 Member Function Documentation

12.100.3.1 readParameters() `void Util::Average::readParameters ( std::istream & in ) [virtual]`

Read parameter nSamplePerBlock from file and initialize.
See setNSamplePerBlock() for discussion of value.

Parameters

in	input stream

Reimplemented from Util::ParamComposite.
Definition at line 52 of file Average.cpp.
References UTIL_THROW.

12.100.3.2 setNSamplePerBlock() `void Util::Average::setNSamplePerBlock ( int nSamplePerBlock )`

Set nSamplePerBlock.
If nSamplePerBlock > 0, the sample function will increment block averages, and reset the average every nSamplePer←Block samples.
If nSamplePerBlock == 0, block averaging is disabled. This is the default (i.e., the initial value set in the constructor).

Parameters

nSamplePerBlock	number of samples per block average output

Definition at line 63 of file Average.cpp.
References nSamplePerBlock(), and UTIL_THROW.

12.100.3.3 loadParameters() `void Util::Average::loadParameters ( Serializable::IArchive & ar ) [virtual]`

Load internal state from an archive.

Parameters

ar	input/loading archive

Reimplemented from Util::ParamComposite.
Definition at line 74 of file Average.cpp.
References Util::AverageStage::serialize(), and UTIL_THROW.

12.100.3.4 save() `void Util::Average::save ( Serializable::OArchive & ar ) [virtual]`

Save internal state to an archive.

Parameters

ar	output/saving archive

Reimplemented from Util::ParamComposite.
Definition at line 88 of file Average.cpp.

12.100.3.5 serialize() `template<class Archive >`

```
void Util::Average::serialize (
            Archive & ar,
```

`const unsigned int version )`

Serialize this Average to or from an archive.

Parameters

ar	input or output archive
version	file version id

Definition at line 252 of file Average.h.
References Util::AverageStage::serialize().

12.100.3.6 clear() `void Util::Average::clear ( ) [virtual]`

Clear all accumulators, set to empty initial state.
Reimplemented from Util::AverageStage.
Definition at line 42 of file Average.cpp.
References Util::AverageStage::clear().

12.100.3.7 sample() [1/2] `void Util::Average::sample ( double value ) [virtual]`

Add a sampled value to the ensemble.

Parameters

value	sampled value

Reimplemented from Util::AverageStage.
Definition at line 94 of file Average.cpp.
References Util::AverageStage::sample().

12.100.3.8 sample() [2/2] `void Util::Average::sample ( double value, std::ostream & out )`

Add a sampled value to ensemble, and output block averages.

Parameters

value	sampled value
out	output stream to which to write block averages

Definition at line 112 of file Average.cpp.
References Util::AverageStage::sample().

12.100.3.9 output() `void Util::Average::output ( std::ostream & out ) const`

Output final statistical properties to file.
This function outputs the average value, an estimate of the error on the average, the variance. It also outputs a sequence of naive values for the error on the average obtained from sequences of block averages, with different levels of blocking. The naive estimate obtained from each stage is calculated as if subsequent values were uncorrelated. This gives

sqrt(variance/nSample), where variance is the variance of the sequence of block averages processed by that stage, and nSample is the number of such block averages thus far. The final estimate of the error on the average is obtained by trying to identify several stages of block averaging that yield statistically indistinguishable naive estimates.

Parameters

out	output stream

Definition at line 178 of file Average.cpp.
References Util::AverageStage::average(), blockingError(), Util::AverageStage::error(), Util::AverageStage::nSample(), Util::AverageStage::stageInterval(), Util::AverageStage::stdDeviation(), and Util::AverageStage::variance().

12.100.3.10 blockingError() `double Util::Average::blockingError ( ) const`

Return estimated error on average from blocking analysis.
Definition at line 133 of file Average.cpp.
References Util::AverageStage::error(), and Util::AverageStage::nSample().
Referenced by output().

12.100.3.11 nSamplePerBlock() `int Util::Average::nSamplePerBlock ( ) const [inline]`

Get number of samples per block average.
A zero value indicates that block averaging is disabled.
Definition at line 220 of file Average.h.
Referenced by setNSamplePerBlock().

12.100.3.12 iBlock() `int Util::Average::iBlock ( ) const [inline]`

Get number of samples in current block average.
Return 0 if block averaging disabled, if !nSamplePerBlock.
Definition at line 226 of file Average.h.

12.100.3.13 isBlockComplete() `bool Util::Average::isBlockComplete ( ) const [inline]`

Is the current block average complete?

Returns

(iBlock $>$ 0) && (iBlock == nSamplePerBlock)

Definition at line 232 of file Average.h.
The documentation for this class was generated from the following files:

- Average.h
- Average.cpp

12.101 Util::AverageStage Class Reference

Evaluate average with hierarchical blocking error analysis.
`#include <AverageStage.h>`
Inheritance diagram for Util::AverageStage:

classUtil_1_1AverageStage-eps-converted-to.pdf

Public Member Functions

- AverageStage (int blockFactor=2)

  Constructor.
- virtual ∼AverageStage ()

  Destructor.
- void setBlockFactor (int blockFactor)

  Reset the value of blockFactor.
- virtual void clear ()

  Initialize all accumulators and recursively destroy all children.
- virtual void sample (double value)

  Add a sampled value to the ensemble.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Add a sampled value to the ensemble.

Accessors

- double average () const

  Return the average of all sampled values.
- double variance () const

  Return the variance of all sampled values.
- double stdDeviation () const

  Return the standard deviation of all sampled values.
- double error () const

  Return a naive estimate for the std deviation of the average.
- long nSample () const

  Return the number of sampled values in this sequence.
- long stageInterval () const

  Return the number of sampled values per block at this stage.
- bool hasChild () const

  Does this object have a child AverageStage for block averages?
- AverageStage & child ()

  Return the child AverageStage by reference.

12.101.1 Detailed Description

Evaluate average with hierarchical blocking error analysis.

This class implements an algorithm to evaluate the average of a sequence, using a hierarchical blocking algorithm to estimate the error on the average. The algorithm is based on the calculation of variances for sequences of block averages for multiple levels of block sizes, as described in the following reference:

``Error estimates on averages of correlated data'', H. Flyvbjerg and H.G. Petersen, J. Chem. Phys. 91, pgs. 461-466 (1989).

The blocking algorithm is implemented here by a creating a linked list of AverageStage objects, each of which is responsible for computing the variance on block averages using a different level of blocking. Each object in this list is assigned an integer chainId. The first AverageStage object in the list, with chainId=0, calculates the average and variance for a "primary" sequence of measured values that are passed as parameters to its sample method. This first object is normally an instance of the Average class, which is a subclass of AverageStage that implements features that are only required by the primary stage. This object has a pointer to a child AverageStage with chainId=1 that calculates the variance of a secondary sequence in which each value is the average of blockFactor consecutive values in the primary sequence. The object with chainId=1 in turn has has a pointer to a child object with chainId=2 that calculates the variance of a sequence in which each value is the average of a block of blockFactor∗∗2 consecutive values of the primary sequence, and so on. In general, the object with chainId=n, calculates the variance of a sequence in which each value is an average of blockFactor∗∗n values of the primary sequence. Each value in the sequence analyzed by the object with chainId=n+1 is calculated by the parent object with chainId=n, by calculating an average of a block of blockFactor consecutive values of its own sequence and passing this block average as a parameter the sample() function of the object with chainId=n+1. New stages in this linked list are instantiated and to the list as needed as the length of the primary sequence grows: When an object with chainId=n has been passed a sequence of exactly blockFactor values, it creates a child AverageStage object with chainId=n+1 and passes the average of these first blockFactor values to the sample function of the child object as the first value in its sequence.
A value of the integer parameter blockFactor is passed to the constructor of the primary AverageStage object. This parameter is set to blockFactor=2 by default. Its value may be reset using the setBlockFactor() function before any data is sampled, but may not be changed thereafter.
Definition at line 66 of file AverageStage.h.

12.101.2 Constructor & Destructor Documentation

12.101.2.1 AverageStage() `Util::AverageStage::AverageStage ( int blockFactor = 2 )`

Constructor.
This constructor creates a primary AverageStage object with stageId = 0 and stageInterval = 1. A private constructor is used to recursively create children of this object.

Parameters

blockFactor	ratio of block sizes of subsequent stages

Definition at line 20 of file AverageStage.cpp.
Referenced by sample(), and serialize().

12.101.2.2 ∼AverageStage() `Util::AverageStage::∼AverageStage ( ) [virtual]`

Destructor.
Recursively destroy all children.
Definition at line 53 of file AverageStage.cpp.

12.101.3 Member Function Documentation

12.101.3.1 setBlockFactor() `void Util::AverageStage::setBlockFactor ( int blockFactor )`

Reset the value of blockFactor.

Exceptions

Exception	if called when nSample > 0.

Definition at line 63 of file AverageStage.cpp.
References UTIL_THROW.

12.101.3.2 clear() `void Util::AverageStage::clear ( ) [virtual]`

Initialize all accumulators and recursively destroy all children.
Reimplemented in Util::Average.
Definition at line 80 of file AverageStage.cpp.
Referenced by Util::Average::clear().

12.101.3.3 sample() `void Util::AverageStage::sample ( double value ) [virtual]`

Add a sampled value to the ensemble.

Parameters

value	sampled value

Reimplemented in Util::Average.
Definition at line 95 of file AverageStage.cpp.
References AverageStage(), and sample().
Referenced by sample(), and Util::Average::sample().

12.101.3.4 serialize() `template<class Archive >`

```
void Util::AverageStage::serialize (
            Archive & ar,
            const unsigned int version )
```

Add a sampled value to the ensemble.

Parameters

ar	input or output archive
version	file version id

Definition at line 252 of file AverageStage.h.
References AverageStage(), and hasChild().
Referenced by Util::Average::loadParameters(), and Util::Average::serialize().

12.101.3.5 average() `double Util::AverageStage::average ( ) const`

Return the average of all sampled values.
Definition at line 131 of file AverageStage.cpp.
Referenced by Util::Average::output().

12.101.3.6 variance() `double Util::AverageStage::variance ( ) const`

Return the variance of all sampled values.
Definition at line 137 of file AverageStage.cpp.
Referenced by error(), Util::Average::output(), and stdDeviation().

12.101.3.7 stdDeviation() `double Util::AverageStage::stdDeviation ( ) const`

Return the standard deviation of all sampled values.

Returns

sqrt(variance())

Definition at line 148 of file AverageStage.cpp.
References variance().
Referenced by Util::Average::output().

12.101.3.8 error() `double Util::AverageStage::error ( ) const`

Return a naive estimate for the std deviation of the average.

Returns

sqrt(variance()/nSample())

Definition at line 166 of file AverageStage.cpp.
References variance().
Referenced by Util::Average::blockingError(), and Util::Average::output().

12.101.3.9 nSample() `long Util::AverageStage::nSample ( ) const`

Return the number of sampled values in this sequence.
Definition at line 154 of file AverageStage.cpp.
Referenced by Util::Average::blockingError(), and Util::Average::output().

12.101.3.10 stageInterval() `long Util::AverageStage::stageInterval ( ) const`

Return the number of sampled values per block at this stage.
Definition at line 160 of file AverageStage.cpp.
Referenced by Util::Average::output().

12.101.3.11 hasChild() `bool Util::AverageStage::hasChild ( ) const [inline], [protected]`

Does this object have a child AverageStage for block averages?
Definition at line 237 of file AverageStage.h.
Referenced by serialize().

12.101.3.12 child() `AverageStage & Util::AverageStage::child ( ) [inline], [protected]`

Return the child AverageStage by reference.
Definition at line 243 of file AverageStage.h.
The documentation for this class was generated from the following files:

- AverageStage.h
- AverageStage.cpp

12.102 Util::Begin Class Reference

Beginning line of a composite parameter block.
#include <Begin.h>
Inheritance diagram for Util::Begin:

classUtil_1_1Begin-eps-converted-to.pdf

Public Member Functions

- Begin (const char ∗label, bool isRequired=true)

  Constructor.
- virtual void readParam (std::istream &in)

  Read the opening line.
- virtual void writeParam (std::ostream &out)

  Write the opening line.
- bool isRequired () const

  Is this the beginning line for a required element?
- bool isActive () const

  Is this an active element (has it been read from file)?
- virtual void resetParam ()

  Do-nothing implementation of virtual resetParam function.

Additional Inherited Members

12.102.1 Detailed Description

Beginning line of a composite parameter block.
Definition at line 24 of file Begin.h.

12.102.2 Constructor & Destructor Documentation

12.102.2.1 Begin()

```
Util::Begin::Begin (
    const char * label,
    bool isRequired = true )
```

Constructor.
Definition at line 21 of file Begin.cpp.
References Util::Label::setString().

12.102.3 Member Function Documentation

12.102.3.1 readParam() `void Util::Begin::readParam ( std::istream & in ) [virtual]`

Read the opening line.

Parameters

in	input stream

Implements Util::ParamComponent.

Definition at line 33 of file Begin.cpp.

References Util::bcast< bool >(), Util::ParamComponent::echo(), Util::Log::file(), Util::MpiFileIo::hasIoCommunicator(), Util::ParamComponent::indent(), Util::MpiFileIo::ioCommunicator(), Util::Label::isClear(), Util::MpiFileIo::isIo←Processor(), isRequired(), UTIL_THROW, and writeParam().

Referenced by Util::ParamComposite::readBegin().

12.102.3.2 writeParam() `void Util::Begin::writeParam ( std::ostream & out ) [virtual]`

Write the opening line.

Parameters

out	output stream

Implements Util::ParamComponent.

Definition at line 80 of file Begin.cpp.

References Util::ParamComponent::indent(), and Util::Label::string().

Referenced by Util::ParamComposite::load(), Util::Factory< Data >::readObject(), and readParam().

12.102.3.3 isRequired() `bool Util::Begin::isRequired ( ) const [inline]`

Is this the beginning line for a required element?

Definition at line 78 of file Begin.h.

References Util::Label::isRequired().

Referenced by readParam().

12.102.3.4 isActive() `bool Util::Begin::isActive ( ) const [inline]`

Is this an active element (has it been read from file)?

Definition at line 84 of file Begin.h.

Referenced by Util::Manager< Data >::beginReadManager(), Util::ParamComposite::readBegin(), and Util::Param←Composite::readParamOptional().

12.102.3.5 resetParam() `void Util::Begin::resetParam ( ) [virtual]`

Do-nothing implementation of virtual resetParam function.

Reimplemented from Util::ParamComponent.

Definition at line 86 of file Begin.cpp.

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

- Begin.h
- Begin.cpp

12.103 Util::BinaryFileIArchive Class Reference

Saving archive for binary istream.
`#include <BinaryFileIArchive.h>`

Public Member Functions

- BinaryFileIArchive ()

  Constructor.
- BinaryFileIArchive (std::string filename)

  Constructor.
- BinaryFileIArchive (std::ifstream &file)

  Constructor.
- virtual ~BinaryFileIArchive ()

  Destructor.
- std::ifstream & file ()

  Get the underlying ifstream by reference.
- template<typename T >
  BinaryFileIArchive & operator& (T &data)

  Read one object.
- template<typename T >
  BinaryFileIArchive & operator>> (T &data)

  Read one object.
- template<typename T >
  void unpack (T &data)

  Unpack a single T object.
- template<typename T >
  void unpack (T ∗array, int n)

  Unpack a C array.
- template<typename T >
  void unpack (T ∗array, int m, int n, int np)

  Unpack a 2D C array.

Static Public Member Functions

- static bool is_saving ()

  Returns true;.
- static bool is_loading ()

  Returns false;.

12.103.1 Detailed Description

Saving archive for binary istream.
Definition at line 30 of file BinaryFileIArchive.h.

12.103.2 Constructor & Destructor Documentation

12.103.2.1 BinaryFileIArchive() [1/3] `Util::BinaryFileIArchive::BinaryFileIArchive ( )`

Constructor.

Definition at line 18 of file BinaryFileIArchive.cpp.

12.103.2.2 BinaryFileIArchive() [2/3] `Util::BinaryFileIArchive::BinaryFileIArchive ( std::string filename )`

Constructor.

Parameters

filename	name of file to open for reading.

Definition at line 27 of file BinaryFileIArchive.cpp.

12.103.2.3 BinaryFileIArchive() [3/3] `Util::BinaryFileIArchive::BinaryFileIArchive ( std::ifstream & file )`

Constructor.

Parameters

file	output file

Definition at line 36 of file BinaryFileIArchive.cpp.

12.103.2.4 ∼BinaryFileIArchive() `Util::BinaryFileIArchive::∼BinaryFileIArchive ( ) [virtual]`

Destructor.

Definition at line 45 of file BinaryFileIArchive.cpp.

12.103.3 Member Function Documentation

12.103.3.1 is_saving() `bool Util::BinaryFileIArchive::is_saving ( ) [inline], [static]`

Returns true;.

Definition at line 128 of file BinaryFileIArchive.h.

12.103.3.2 is_loading() `bool Util::BinaryFileIArchive::is_loading ( ) [inline], [static]`

Returns false;.

Definition at line 131 of file BinaryFileIArchive.h.

12.103.3.3 file() `std::ifstream & Util::BinaryFileIArchive::file ( )`

Get the underlying ifstream by reference.

Definition at line 55 of file BinaryFileIArchive.cpp.

12.103.3.4 operator&() template<typename T >

```
BinaryFileIArchive & Util::BinaryFileIArchive::operator& (
            T & data ) [inline]
```

Read one object.

Definition at line 140 of file BinaryFileIArchive.h.

12.103.3.5 operator>>() template<typename T >

```
BinaryFileIArchive & Util::BinaryFileIArchive::operator>> (
            T & data ) [inline]
```

Read one object.

Definition at line 150 of file BinaryFileIArchive.h.

12.103.3.6 unpack() [1/3] template<typename T >

```
void Util::BinaryFileIArchive::unpack (
            T & data ) [inline]
```

Unpack a single T object.

Definition at line 162 of file BinaryFileIArchive.h.

Referenced by Util::CArrayParam< Type >::loadValue(), and Util::CArray2DParam< Type >::loadValue().

12.103.3.7 unpack() [2/3] template<typename T >

```
void Util::BinaryFileIArchive::unpack (
            T * array,
            int n ) [inline]
```

Unpack a C array.

Parameters

array	pointer to array (or first element)
n	number of elements

Definition at line 169 of file BinaryFileIArchive.h.

12.103.3.8 unpack() [3/3] template<typename T >

```
void Util::BinaryFileIArchive::unpack (
            T * array,
            int m,
            int n,
            int np ) [inline]
```

Unpack a 2D C array.

This unpacks the elements of an m x n logical array into a physical 2D C array of type array[][np], where np is the physical length of a row, i.e., the amount of memory allocated per row.

Parameters

array	pointer to first row
m	number of rows
n	logical number of columns
np	physical number of columns

Definition at line 180 of file BinaryFileIArchive.h.
The documentation for this class was generated from the following files:

- BinaryFileIArchive.h
- BinaryFileIArchive.cpp

12.104 Util::BinaryFileOArchive Class Reference

Saving / output archive for binary ostream.
`#include <BinaryFileOArchive.h>`

Public Member Functions

- BinaryFileOArchive ()
  Constructor.
- BinaryFileOArchive (std::string filename)
  Constructor.
- BinaryFileOArchive (std::ofstream &file)
  Constructor.
- virtual ∼BinaryFileOArchive ()
  Destructor.
- std::ofstream & file ()
  Get the underlying ifstream by reference.
- template<typename T >
  BinaryFileOArchive & operator& (T &data)
  Save one object.
- template<typename T >
  BinaryFileOArchive & operator<< (T &data)
  Save one object.
- template<typename T >
  void pack (const T &data)
  Pack one object of type T.
- template<typename T >
  void pack (const T ∗array, int n)
  Pack a C array.
- template<typename T >
  void pack (const T ∗array, int m, int n, int np)
  Pack a 2D C array.

Static Public Member Functions

- static bool is_saving ()
  Returns true;.
- static bool is_loading ()
  Returns false;.

12.104.1 Detailed Description

Saving / output archive for binary ostream.
Definition at line 30 of file BinaryFileOArchive.h.

12.104.2 Constructor & Destructor Documentation

12.104.2.1 BinaryFileOArchive() [1/3] `Util::BinaryFileOArchive::BinaryFileOArchive ( )`

Constructor.

Definition at line 16 of file BinaryFileOArchive.cpp.

12.104.2.2 BinaryFileOArchive() [2/3] `Util::BinaryFileOArchive::BinaryFileOArchive ( std::string filename )`

Constructor.

Parameters

filename	name of file to open for reading.

Definition at line 25 of file BinaryFileOArchive.cpp.

12.104.2.3 BinaryFileOArchive() [3/3] `Util::BinaryFileOArchive::BinaryFileOArchive ( std::ofstream & file )`

Constructor.

Parameters

file	output file

Definition at line 34 of file BinaryFileOArchive.cpp.

12.104.2.4 ∼BinaryFileOArchive() `Util::BinaryFileOArchive::∼BinaryFileOArchive ( ) [virtual]`

Destructor.

Definition at line 43 of file BinaryFileOArchive.cpp.

12.104.3 Member Function Documentation

12.104.3.1 is_saving() `bool Util::BinaryFileOArchive::is_saving ( ) [inline], [static]`

Returns true;.

Definition at line 126 of file BinaryFileOArchive.h.

12.104.3.2 is_loading() `bool Util::BinaryFileOArchive::is_loading ( ) [inline], [static]`

Returns false;.

Definition at line 129 of file BinaryFileOArchive.h.

12.104.3.3 file() `std::ofstream & Util::BinaryFileOArchive::file ( )`

Get the underlying ifstream by reference.

Definition at line 53 of file BinaryFileOArchive.cpp.

12.104.3.4 operator&() template<typename T >

BinaryFileOArchive & Util::BinaryFileOArchive::operator& (
    T & *data*) [inline]

Save one object.

Definition at line 138 of file BinaryFileOArchive.h.

12.104.3.5 operator<<() template<typename T >

BinaryFileOArchive & Util::BinaryFileOArchive::operator<< (
    T & *data*) [inline]

Save one object.

Definition at line 148 of file BinaryFileOArchive.h.

12.104.3.6 pack() [1/3] template<typename T >

void Util::BinaryFileOArchive::pack (
    const T & *data*) [inline]

Pack one object of type T.

Definition at line 160 of file BinaryFileOArchive.h.

Referenced by Util::Parameter::saveOptionalCArray(), Util::Parameter::saveOptionalCArray2D(), Util::CArrayParam< Type >::saveValue(), and Util::CArray2DParam< Type >::saveValue().

12.104.3.7 pack() [2/3] template<typename T >

void Util::BinaryFileOArchive::pack (
    const T * *array*,
    int *n*) [inline]

Pack a C array.

Parameters

array	address of first element
n	number of elements

Definition at line 167 of file BinaryFileOArchive.h.

12.104.3.8 pack() [3/3] template<typename T >

void Util::BinaryFileOArchive::pack (
    const T * *array*,
    int *m*,
    int *n*,
    int *np*) [inline]

Pack a 2D C array.

This packs m rows of length n within a 2D C array allocated as array[][np], where np is the physical length of one row.

Parameters

array	pointer to [0][0] element in 2D array
m	number of rows
n	logical number of columns
np	physical number of columns

Definition at line 178 of file BinaryFileOArchive.h.
The documentation for this class was generated from the following files:

- BinaryFileOArchive.h
- BinaryFileOArchive.cpp

12.105 Util::Binomial Class Reference

Class for binomial coefficients (all static members)
`#include <Binomial.h>`

Static Public Member Functions

- static void setup (int nMax)

  Precompute all combinations C(n, m) up to n = nMax.
- static void clear ()

  Release all static memory.
- static int coeff (int n, int m)

  Return coefficient "n choose m", or C(n, m) = n!/(m!(n-m)!).

12.105.1 Detailed Description

Class for binomial coefficients (all static members)
Definition at line 27 of file Binomial.h.

12.105.2 Member Function Documentation

12.105.2.1 setup()

```
void Util::Binomial::setup (
        int nMax )  [static]
```

Precompute all combinations C(n, m) up to n = nMax.
Algorithm: Construct rows [0,...,nMax] of Pascal's triangle.

Parameters

nMax	maximum value of n to precompute.

Definition at line 17 of file Binomial.cpp.
References Util::GArray< Data >::resize(), and UTIL_CHECK.
Referenced by coeff(), and Util::Polynomial< double >::shift().

12.105.2.2 clear()

```
void Util::Binomial::clear ( )  [static]
```

Release all static memory.
Definition at line 47 of file Binomial.cpp.
References Util::GArray< Data >::capacity(), and Util::GArray< Data >::deallocate().

12.105.2.3 coeff()

```
int Util::Binomial::coeff (
        int n,
        int m )  [static]
```

Return coefficient "n choose m", or C(n, m) = n!/(m!(n-m)!).

Algorithm: Returns precomputed value C(n,m) if already known. Otherwise, calls setup(n) to computes and stores values of C(n', m) for all n' <= n, then returns desired value.

Parameters

n	larger integer (overall power in binomial)
m	parameter in range [0,n]

Definition at line 55 of file Binomial.cpp.
References setup(), and UTIL_CHECK.
Referenced by Util::Polynomial< double >::shift().
The documentation for this class was generated from the following files:

- Binomial.h
- Binomial.cpp

12.106 Util::Bit Class Reference

Represents a specific bit location within an unsigned int.
`#include <Bit.h>`

Public Member Functions

- Bit ()

  Default constructor.
- Bit (unsigned int shift)

  Constructor.
- void setMask (unsigned int shift)

  Set or reset the bit mask.
- void set (unsigned int &flags) const

  Set this bit in the flags parameter.
- void clear (unsigned int &flags) const

  Clear this bit in the flags parameter.
- bool isSet (unsigned int flags) const

  Is this bit set in the flags integer?
- unsigned int mask () const

  Return integer with only this bit set.

12.106.1 Detailed Description

Represents a specific bit location within an unsigned int.
Provides methods to query, set or clear a particular bit.
Definition at line 21 of file Bit.h.

12.106.2 Constructor & Destructor Documentation

12.106.2.1 Bit() [1/2] `Util::Bit::Bit ( )`

Default constructor.
Definition at line 18 of file Bit.cpp.

12.106.2.2 Bit() [2/2] `Util::Bit::Bit ( unsigned int shift )`

Constructor.

Parameters

shift	location of the bit, 0 < shift <= 32.

Definition at line 25 of file Bit.cpp.
References setMask().

12.106.3 Member Function Documentation

12.106.3.1 setMask() `void Util::Bit::setMask ( unsigned int shift )`

Set or reset the bit mask.

Parameters

shift	location of the bit, 0 < shift <= 32.

Definition at line 31 of file Bit.cpp.
References UTIL_THROW.
Referenced by Bit().

12.106.3.2 set() `void Util::Bit::set ( unsigned int & flags ) const [inline]`

Set this bit in the flags parameter.

Parameters

flags	unsigned int to be modified

Definition at line 80 of file Bit.h.

12.106.3.3 clear() `void Util::Bit::clear ( unsigned int & flags ) const [inline]`

Clear this bit in the flags parameter.

Parameters

flags	unsigned int to be modified

Definition at line 86 of file Bit.h.

12.106.3.4 isSet() `bool Util::Bit::isSet ( unsigned int flags ) const [inline]`

Is this bit set in the flags integer?

Parameters

flags	unsigned int to be queried

Definition at line 92 of file Bit.h.

12.106.3.5 mask() `unsigned int Util::Bit::mask ( ) const [inline]`

Return integer with only this bit set.
Definition at line 98 of file Bit.h.
The documentation for this class was generated from the following files:

- Bit.h
- Bit.cpp

12.107 Util::Blank Class Reference

An empty line within a parameter file.
`#include <Blank.h>`
Inheritance diagram for Util::Blank:

classUtil_1_1Blank-eps-converted-to.pdf

Public Member Functions

- Blank ()

  Constructor.
- virtual ∼Blank ()

  Virtual Destructor.
- virtual void readParam (std::istream &in)

  Read a blank line.
- virtual void writeParam (std::ostream &out)

  Write a blank line.

Additional Inherited Members

12.107.1 Detailed Description

An empty line within a parameter file.
A Param represents an empty line within a file format that is represented as a ParamComposite.
Definition at line 24 of file Blank.h.

12.107.2 Constructor & Destructor Documentation

12.107.2.1 Blank() `Util::Blank::Blank ( )`

Constructor.
Definition at line 14 of file Blank.cpp.

12.107.2.2 ∼Blank() `Util::Blank::∼Blank ( ) [virtual]`

Virtual Destructor.
Definition at line 19 of file Blank.cpp.

12.107.3 Member Function Documentation

12.107.3.1 readParam() `void Util::Blank::readParam ( std::istream & in ) [virtual]`

Read a blank line.

Parameters

in	input stream

Implements Util::ParamComponent.
Definition at line 27 of file Blank.cpp.
References Util::ParamComponent::echo(), Util::Log::file(), Util::MpiFileIo::isIoProcessor(), and writeParam().
Referenced by Util::ParamComposite::readBlank().

12.107.3.2 writeParam() `void Util::Blank::writeParam ( std::ostream & out ) [virtual]`

Write a blank line.

Parameters

out	output stream

Implements Util::ParamComponent.
Definition at line 43 of file Blank.cpp.
Referenced by readParam().
The documentation for this class was generated from the following files:

- Blank.h
- Blank.cpp

12.108 Util::Bool Class Reference

Wrapper for an bool value, for formatted ostream output.
`#include <Bool.h>`

Public Member Functions

Constructors

- Bool ()
  Default constructor.

- Bool (bool value)
  Constructor, value only.
- Bool (bool value, int width)
  Constructor, value and width.

Setters

- void **setValue** (bool value)
- void **setWidth** (int width)

Accessors

- bool **value** ()
- int **width** ()
- std::istream & operator>> (std::istream &in, Bool &object)
  Input stream extractor for an Bool object.
- std::ostream & operator<< (std::ostream &out, const Bool &object)
  Output stream inserter for an Bool object.

12.108.1 Detailed Description

Wrapper for an bool value, for formatted ostream output.
An Bool object has a bool value, and a minimum output field width. The << operator for a Bool uses the specified width. The value and width may both be specified as parameters to a constructor. If the width is not specified as a constructor parameter, it is set within the constructor to the default Format::defaultWidth().
An Bool object may be passed to an ostream as a temporary object. For example, the expression:

```
std::cout « Bool(true) « Bool(false, 10) « std::endl;
```

outputs the value true using the default width, followed by the false value in a field of minimum width 10.
Definition at line 35 of file Bool.h.

12.108.2 Constructor & Destructor Documentation

12.108.2.1 Bool() [1/3] `Util::Bool::Bool ( )`

Default constructor.
Definition at line 15 of file Bool.cpp.

12.108.2.2 Bool() [2/3] `Util::Bool::Bool (`

```
bool value ) [explicit]
```

Constructor, value only.
Definition at line 21 of file Bool.cpp.

12.108.2.3 Bool() [3/3] `Util::Bool::Bool (`

```
bool value,
int width )
```

Constructor, value and width.
Definition at line 27 of file Bool.cpp.

12.108.3 Friends And Related Function Documentation

12.108.3.1 operator>> std::istream& operator>> (std::istream & *in*, Bool & *object*) [friend]

Input stream extractor for an Bool object.

Parameters

in	input stream
object	Bool object to be read from stream

Returns

modified input stream

Definition at line 47 of file Bool.cpp.

12.108.3.2 operator<< std::ostream& operator<< (std::ostream & *out*, const Bool & *object*) [friend]

Output stream inserter for an Bool object.

Parameters

out	output stream
object	Bool to be written to stream

Returns

modified output stream

Definition at line 56 of file Bool.cpp.

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

- Bool.h
- Bool.cpp

12.109 Util::CardinalBSpline Class Reference

A cardinal B-spline basis function.

`#include <CardinalBSpline.h>`

Constructor and Destructor

- CardinalBSpline (int degree, bool verbose=false)

    Construct a spline basis of specified degree.
- ∼CardinalBSpline ()

    Destructor.
- const Polynomial< double > & operator[ ] (int i) const

    Get Polynomial<double> object for domain [i,i+1].
- double operator() (double x) const

    Compute the value of the spline basis function.
- int degree () const

    Return degree of basis function (i.e., degree of polynomials).

12.109.1 Detailed Description

A cardinal B-spline basis function.

A cardinal B-Spline of order m or degree k = m - 1 is a piecewise continuous polynomial of degree k defined over the domain [0, k+1]. Such a function is defined by k different polynomials, each of which has a domain [i, i+1] for an integer i with 0 <= i <=k. For k > 0, the function and k-1 derivatives are continuous.

A CardinalBSpline object of degree k has k Polynomial<double> objects, indexed by an integer 0 <= i <= k, each of which defines the polynomial with a domain [i,i+1].

If object b is a CardinalBSpline of degree k, then:

- Operator b[i] returns the Polynomial<double> for domain [i,i+1]
- Operator b(x) returns the value of basis function b for real x.

Definition at line 42 of file CardinalBSpline.h.

12.109.2 Constructor & Destructor Documentation

12.109.2.1 CardinalBSpline() `Util::CardinalBSpline::CardinalBSpline ( int degree, bool verbose = false )`

Construct a spline basis of specified degree.

Parameters

degree	degree of the function (i.e., degree of polynomials)
verbose	if true, write verbose report to std::cout

Definition at line 17 of file CardinalBSpline.cpp.

References Util::DArray< Data >::allocate(), degree(), and UTIL_CHECK.

12.109.2.2 ∼CardinalBSpline() `Util::CardinalBSpline::∼CardinalBSpline ( )`

Destructor.

Definition at line 125 of file CardinalBSpline.cpp.

12.109.3 Member Function Documentation

12.109.3.1 operator[]() `const Polynomial< double > & Util::CardinalBSpline::operator[] ( int i ) const [inline]`

Get Polynomial<double> object for domain [i,i+1].

If b is a CardinalBSpline, b[i] returns the Polynomial<double> object (the polynomial with double precision floating point coefficients) associated with the domain [i,i+1].

Parameters

i	integer index in range 0 <= i <= degree.

Returns

polynomial associated with domain [i, i+1]

Definition at line 119 of file CardinalBSpline.h.

12.109.3.2 operator()() `double Util::CardinalBSpline::operator() ( double x ) const [inline]`

Compute the value of the spline basis function.

If b is a CardinalBSpline, b(x) returns the value of the spline function for specified floating point argument x, giving a nonzero value only for 0 < x < degree + 1.

Parameters

x	argument of spline basis function

Definition at line 126 of file CardinalBSpline.h.

References UTIL_ASSERT.

12.109.3.3 degree() `int Util::CardinalBSpline::degree ( ) const [inline]`

Return degree of basis function (i.e., degree of polynomials).

Definition at line 139 of file CardinalBSpline.h.

Referenced by CardinalBSpline().

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

- CardinalBSpline.h
- CardinalBSpline.cpp

12.110 Util::CArray2DParam< Type > Class Template Reference

A Parameter associated with a 2D built-in C array.

`#include <CArray2DParam.h>`

Inheritance diagram for Util::CArray2DParam< Type >:

classUtil_1_1CArray2DParam-eps-converted-to.pdf

Public Member Functions

- CArray2DParam (const char ∗label, Type ∗ptr, int m, int n, int np, bool isRequired=true)

  Constructor.
- void writeParam (std::ostream &out)

  Write 2D C array to file.

Protected Member Functions

- virtual void readValue (std::istream &in)

  Read 2D array parameter from an input stream.
- virtual void loadValue (Serializable::IArchive &ar)

  Load 2D array from an archive.
- virtual void saveValue (Serializable::OArchive &ar)

  Save 2D array to an archive.
- virtual void bcastValue ()

  Broadcast 2D array within the ioCommunicator.

Additional Inherited Members

12.110.1 Detailed Description

template< class Type >
class Util::CArray2DParam< Type >

A Parameter associated with a 2D built-in C array.
Definition at line 29 of file CArray2DParam.h.

12.110.2 Constructor & Destructor Documentation

12.110.2.1 CArray2DParam() `template<class Type >`

```
Util::CArray2DParam< Type >::CArray2DParam (
        const char * label,
        Type * ptr,
        int m,
        int n,
        int np,
        bool isRequired = true )
```

Constructor.
Example: A 2 X 2 matrix stored in an oversized 3 x 3 C array.

```
double                 matrix[3][3];
CArray2DParam<double> param("matrix", matrix[0], 2, 2, 3);
```

Parameters

label	parameter label (usually a literal C-string)
ptr	pointer to first element of first row of 2D array
m	logical number of rows
n	logical number of columns
np	physical number of columns (allocated elements per row).
isRequired	Is this a required parameter?

Definition at line 109 of file CArray2DParam.h.

12.110.3 Member Function Documentation

12.110.3.1 writeParam() template<class Type >

```
void Util::CArray2DParam< Type >::writeParam (
            std::ostream & out )  [virtual]
```

Write 2D C array to file.

Implements Util::ParamComponent.

Definition at line 158 of file CArray2DParam.h.

References Util::Parameter::Precision, and Util::Parameter::Width.

12.110.3.2 readValue() template<class Type >

```
void Util::CArray2DParam< Type >::readValue (
            std::istream & in )  [protected], [virtual]
```

Read 2D array parameter from an input stream.

Parameters

in	input stream from which to read

Reimplemented from Util::Parameter.

Definition at line 121 of file CArray2DParam.h.

12.110.3.3 loadValue() template<class Type >

```
void Util::CArray2DParam< Type >::loadValue (
            Serializable::IArchive & ar )  [protected], [virtual]
```

Load 2D array from an archive.

Parameters

ar	input archive from which to load

Reimplemented from Util::Parameter.

Definition at line 135 of file CArray2DParam.h.

References Util::BinaryFileIArchive::unpack().

12.110.3.4 saveValue() template<class Type >

```
void Util::CArray2DParam< Type >::saveValue (
            Serializable::OArchive & ar )  [protected], [virtual]
```

Save 2D array to an archive.

Parameters

ar	output archive to which to save

Reimplemented from Util::Parameter.

Definition at line 142 of file CArray2DParam.h.

References Util::BinaryFileOArchive::pack().

12.110.3.5 bcastValue() template<class Type >

```
void Util::CArray2DParam< Type >::bcastValue  [protected], [virtual]
```

Broadcast 2D array within the ioCommunicator.
Reimplemented from Util::Parameter.
Definition at line 150 of file CArray2DParam.h.
The documentation for this class was generated from the following file:

- CArray2DParam.h

12.111 Util::CArrayParam< Type > Class Template Reference

A Parameter associated with a 1D C array.
`#include <CArrayParam.h>`
Inheritance diagram for Util::CArrayParam< Type >:

classUtil_1_1CArrayParam-eps-converted-to.pdf

Public Member Functions

- CArrayParam (const char ∗label, Type ∗value, int n, bool isRequired=true)

  Constructor.
- void writeParam (std::ostream &out)

  Write parameter to stream.

Protected Member Functions

- virtual void readValue (std::istream &in)

  Read parameter value from an input stream.
- virtual void loadValue (Serializable::IArchive &ar)

  Load bare parameter value from an archive.
- virtual void saveValue (Serializable::OArchive &ar)

  Save parameter value to an archive.
- virtual void bcastValue ()

  Broadcast parameter value within the ioCommunicator.

Additional Inherited Members

12.111.1 Detailed Description

template<class Type>
class Util::CArrayParam< Type >

A Parameter associated with a 1D C array.
Definition at line 22 of file CArrayParam.h.

12.111.2 Constructor & Destructor Documentation

12.111.2.1 CArrayParam() template<class Type >

```
Util::CArrayParam< Type >::CArrayParam (
            const char * label,
            Type * value,
            int n,
            bool isRequired = true )
```

Constructor.

Definition at line 83 of file CArrayParam.h.

12.111.3 Member Function Documentation

12.111.3.1 writeParam() template<class Type >

```
void Util::CArrayParam< Type >::writeParam (
            std::ostream & out ) [virtual]
```

Write parameter to stream.

Parameters

out	output stream

Implements Util::ParamComponent.

Definition at line 127 of file CArrayParam.h.

References Util::Parameter::Precision, and Util::Parameter::Width.

12.111.3.2 readValue() template<class Type >

```
void Util::CArrayParam< Type >::readValue (
            std::istream & in ) [protected], [virtual]
```

Read parameter value from an input stream.

Parameters

in	input stream from which to read

Reimplemented from Util::Parameter.

Definition at line 93 of file CArrayParam.h.

12.111.3.3 loadValue() template<class Type >

```
void Util::CArrayParam< Type >::loadValue (
            Serializable::IArchive & ar ) [protected], [virtual]
```

Load bare parameter value from an archive.

Parameters

ar	input archive from which to load

Reimplemented from Util::Parameter.
Definition at line 104 of file CArrayParam.h.
References Util::BinaryFileIArchive::unpack().

12.111.3.4 saveValue() `template<class Type >`

`void Util::CArrayParam< Type >::saveValue ( Serializable::OArchive & ar ) [protected], [virtual]`

Save parameter value to an archive.

Parameters

ar	output archive to which to save

Reimplemented from Util::Parameter.
Definition at line 111 of file CArrayParam.h.
References Util::BinaryFileOArchive::pack().

12.111.3.5 bcastValue() `template<class Type >`

`void Util::CArrayParam< Type >::bcastValue [protected], [virtual]`

Broadcast parameter value within the ioCommunicator.
Reimplemented from Util::Parameter.
Definition at line 119 of file CArrayParam.h.
The documentation for this class was generated from the following file:

- CArrayParam.h

12.112 Util::Constants Class Reference

Mathematical constants.
`#include <Constants.h>`

Static Public Member Functions

- static void initStatic ()

  Initialize static constants.

Static Public Attributes

- static const double Pi = 2.0∗acos(0.0)

  Trigonometric constant Pi.
- static const std::complex< double > Im = std::complex<double>(0.0, 1.0)

  Square root of -1.

12.112.1 Detailed Description

Mathematical constants.
Definition at line 22 of file Constants.h.

12.112.2 Member Function Documentation

12.112.2.1 initStatic() `void Util::Constants::initStatic ( ) [static]`

Initialize static constants.

Definition at line 22 of file Constants.cpp.

Referenced by Util::initStatic().

12.112.3 Member Data Documentation

12.112.3.1 Pi `const double Util::Constants::Pi = 2.0*acos(0.0) [static]`

Trigonometric constant Pi.

Definition at line 35 of file Constants.h.

12.112.3.2 Im `const std::complex< double > Util::Constants::Im = std::complex<double>(0.0, 1.0) [static]`

Square root of -1.

Definition at line 40 of file Constants.h.

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

- Constants.h
- Constants.cpp

12.113 Util::ConstArrayIterator< Data > Class Template Reference

Forward const iterator for an Array or a C array.

`#include <ConstArrayIterator.h>`

Public Member Functions

- ConstArrayIterator ()

  Default constructor.
- void setCurrent (Data ∗ptr)

  Set the current pointer value.
- void setEnd (Data ∗ptr)

  Set the value of the end pointer.
- bool isEnd () const

  Has the end of the array been reached?
- bool notEnd () const

  Is this not the end of the array?
- const Data ∗ get () const

  Return a pointer to the current data.

Operators

- const Data & operator∗ () const

  Get a reference to the current Data.
- const Data ∗ operator-> () const

  Provide a pointer to the current Data object.
- ConstArrayIterator< Data > & operator++ ()

  Increment the current pointer.

12.113.1 Detailed Description

template<typename Data>
class Util::ConstArrayIterator< Data >

Forward const iterator for an Array or a C array.
A ConstArrayIterator is a forward iterator for an array of const data. It is an abstraction of a pointer to const, similar to an STL const forward iterator. The * operator returns a const reference to an associated Data object the -> operator returns a const pointer to that object. The ++ operator increments the current pointer by one array element.
Unlike an STL const forward iterator, an ConstArrayIterator contains the address of the end of the array. The isEnd() method can be used to test for termination of a for or while loop. When isEnd() is true, the current pointer is one past the end of the array, and thus the iterator has no current value.
A ConstArrayIterator behave like a pointer to constant data, and provides read-only access to the object to which it points. Use an ArrayIterator if you need read-write access.
Definition at line 37 of file ConstArrayIterator.h.

12.113.2 Constructor & Destructor Documentation

12.113.2.1 ConstArrayIterator()

```
template<typename Data >
Util::ConstArrayIterator< Data >::ConstArrayIterator ( )  [inline]
```

Default constructor.

Constructs an uninitialized iterator.

Definition at line 47 of file ConstArrayIterator.h.

12.113.3 Member Function Documentation

12.113.3.1 setCurrent()

```
template<typename Data >
void Util::ConstArrayIterator< Data >::setCurrent (
             Data * ptr )  [inline]
```

Set the current pointer value.

Parameters

ptr	Pointer to current element of the array.

Definition at line 57 of file ConstArrayIterator.h.
Referenced by Util::Array< Pscf::Monomer >::begin(), Util::FSArray< double, 6 >::begin(), Util::FArray< DPropagator, 2 >::begin(), Util::GArray< Rational >::begin(), and Util::DSArray< Data >::begin().

12.113.3.2 setEnd()

```
template<typename Data >
void Util::ConstArrayIterator< Data >::setEnd (
             Data * ptr )  [inline]
```

Set the value of the end pointer.

Parameters

ptr	Pointer to one element past end of array.

Definition at line 65 of file ConstArrayIterator.h.
Referenced by Util::Array< Pscf::Monomer >::begin(), Util::FSArray< double, 6 >::begin(), Util::FArray< DPropagator, 2 >::begin(), Util::GArray< Rational >::begin(), and Util::DSArray< Data >::begin().

12.113.3.3 isEnd() `template<typename Data >`

`bool Util::ConstArrayIterator< Data >::isEnd ( ) const [inline]`

Has the end of the array been reached?

Returns

true if at end, false otherwise.

Definition at line 73 of file ConstArrayIterator.h.

12.113.3.4 notEnd() `template<typename Data >`

`bool Util::ConstArrayIterator< Data >::notEnd ( ) const [inline]`

Is this not the end of the array?

Returns

true if not at end, false otherwise.

Definition at line 81 of file ConstArrayIterator.h.

12.113.3.5 get() `template<typename Data >`

`const Data* Util::ConstArrayIterator< Data >::get ( ) const [inline]`

Return a pointer to the current data.

Returns

true if at end, false otherwise.

Definition at line 89 of file ConstArrayIterator.h.

12.113.3.6 operator*() `template<typename Data >`

`const Data& Util::ConstArrayIterator< Data >::operator* ( ) const [inline]`

Get a reference to the current Data.

Returns

reference to associated Data object

Definition at line 100 of file ConstArrayIterator.h.

12.113.3.7 operator->() `template<typename Data >`

`const Data* Util::ConstArrayIterator< Data >::operator-> ( ) const [inline]`

Provide a pointer to the current Data object.

Returns

const pointer to the Data object

Definition at line 108 of file ConstArrayIterator.h.

12.113.3.8 operator++() `template<typename Data >`

`ConstArrayIterator<Data>& Util::ConstArrayIterator< Data >::operator++ ( ) [inline]`

Increment the current pointer.

Returns

this ConstArrayIterator, after modification.

Definition at line 116 of file ConstArrayIterator.h.

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

- ConstArrayIterator.h

12.114 Util::ConstPArrayIterator< Data > Class Template Reference

Forward iterator for a PArray.

`#include <ConstPArrayIterator.h>`

Public Member Functions

- ConstPArrayIterator ()
  Default constructor.
- void setCurrent (Data **ptr)
  Set the current pointer value.
- void setEnd (Data **ptr)
  Set the value of the end pointer.
- void setNull ()
  Nullify the iterator.
- bool isEnd () const
  Is the current pointer at the end of the array.
- bool notEnd () const
  Is the current pointer not at the end of the array?
- const Data * get () const
  Return a pointer to const current data.

Operators

- const Data & operator* () const
  Return a const refererence to the current Data.
- const Data * operator-> () const
  Provide a pointer to the current Data object.
- ConstPArrayIterator< Data > & operator++ ()
  Increment the current pointer.

12.114.1 Detailed Description

template<typename Data>
class Util::ConstPArrayIterator< Data >

Forward iterator for a PArray.

An ConstPArrayIterator is an abstraction of a pointer, similar to an STL forward iterator. The * operator returns a reference to an associated Data object, the -> operator returns a pointer to that object. The ++ operator increments the current pointer by one array element.

Unlike an STL forward iterator, an ConstPArrayIterator contains the address of the end of the array. The isEnd() method can be used to test for termination of a for or while loop. When isEnd() is true, the iterator has no current value, and cannot be incremented further. The isEnd() method returns true either if the iterator: i) has already been incremented one past the end of an associated PArray, or ii) is in a null state that is produced by the constructor and the clear() method.

Definition at line 34 of file ConstPArrayIterator.h.

12.114.2 Constructor & Destructor Documentation

12.114.2.1 ConstPArrayIterator() `template<typename Data >`

`Util::ConstPArrayIterator< Data >::ConstPArrayIterator ( ) [inline]`

Default constructor.

Constructs a null iterator.

Definition at line 44 of file ConstPArrayIterator.h.

12.114.3 Member Function Documentation

12.114.3.1 setCurrent() `template<typename Data >`

```
void Util::ConstPArrayIterator< Data >::setCurrent (
             Data ** ptr ) [inline]
```

Set the current pointer value.

Parameters

ptr	Pointer to current element of array of Data∗ pointers.

Definition at line 55 of file ConstPArrayIterator.h.

Referenced by Util::PArray< Data >::begin(), Util::FPArray< Data, Capacity >::begin(), and Util::SSet< Data, Capacity >::begin().

12.114.3.2 setEnd() `template<typename Data >`

```
void Util::ConstPArrayIterator< Data >::setEnd (
             Data ** ptr ) [inline]
```

Set the value of the end pointer.

Parameters

ptr	Pointer to one element past end of array of Data∗ pointers.

Definition at line 66 of file ConstPArrayIterator.h.

Referenced by Util::PArray< Data >::begin(), Util::FPArray< Data, Capacity >::begin(), and Util::SSet< Data, Capacity >::begin().

12.114.3.3 setNull() `template<typename Data >`

`void Util::ConstPArrayIterator< Data >::setNull ( ) [inline]`

Nullify the iterator.

Definition at line 72 of file ConstPArrayIterator.h.

Referenced by Util::PArray< Data >::begin().

12.114.3.4 isEnd() `template<typename Data >`

`bool Util::ConstPArrayIterator< Data >::isEnd ( ) const [inline]`

Is the current pointer at the end of the array.

Returns

true if at end, false otherwise.

Definition at line 84 of file ConstPArrayIterator.h.

12.114.3.5 notEnd() `template<typename Data >`

`bool Util::ConstPArrayIterator< Data >::notEnd ( ) const [inline]`

Is the current pointer not at the end of the array?

Returns

true if not at end, false otherwise.

Definition at line 92 of file ConstPArrayIterator.h.

12.114.3.6 get() `template<typename Data >`

`const Data* Util::ConstPArrayIterator< Data >::get ( ) const [inline]`

Return a pointer to const current data.

Returns

true if at end, false otherwise.

Definition at line 100 of file ConstPArrayIterator.h.

12.114.3.7 operator*() `template<typename Data >`

`const Data& Util::ConstPArrayIterator< Data >::operator* ( ) const [inline]`

Return a const refererence to the current Data.

Returns

const reference to the Data object

Definition at line 111 of file ConstPArrayIterator.h.

12.114.3.8 operator->() `template<typename Data >`

`const Data* Util::ConstPArrayIterator< Data >::operator-> ( ) const [inline]`

Provide a pointer to the current Data object.

Returns

pointer to the Data object

Definition at line 119 of file ConstPArrayIterator.h.

12.114.3.9 operator++() template<typename Data >

`ConstPArrayIterator<Data>& Util::ConstPArrayIterator< Data >::operator++ ( ) [inline]`

Increment the current pointer.

Returns

this ConstPArrayIterator, after modification.

Definition at line 127 of file ConstPArrayIterator.h.

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

- ConstPArrayIterator.h

12.115 Util::DArray< Data > Class Template Reference

Dynamically allocatable contiguous array template.

`#include <DArray.h>`

Inheritance diagram for Util::DArray< Data >:

classUtil_1_1DArray-eps-converted-to.pdf

Public Member Functions

- DArray ()

  Default constructor.
- DArray (const DArray< Data > &other)

  Copy constructor.
- virtual ∼DArray ()

  Destructor.
- DArray< Data > & operator= (DArray< Data > const &other)

  Assignment operator.
- void allocate (int capacity)

  Allocate the underlying C array.
- void deallocate ()

  Dellocate the underlying C array.
- void reallocate (int capacity)

  Reallocate the underlying C array and copy to new location.
- bool isAllocated () const

  Return true if the DArray has been allocated, false otherwise.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize a DArray to/from an Archive.

Additional Inherited Members

12.115.1 Detailed Description

template<typename Data>
class Util::DArray< Data >

Dynamically allocatable contiguous array template.
A DArray wraps a dynamically allocated C Array, and stores the size of the array. A DArray can be allocated, deallocated or reallocated (i.e., resized and moved) by member functions.j
The Array<Data> base class provides bounds checking when compiled in debug mode.
Definition at line 31 of file DArray.h.

12.115.2 Constructor & Destructor Documentation

12.115.2.1 DArray() `[1/2]`

```
template<class Data >
Util::DArray< Data >::DArray
```

Default constructor.
Definition at line 121 of file DArray.h.

12.115.2.2 DArray() `[2/2]`

```
template<class Data >
Util::DArray< Data >::DArray (
            const DArray< Data > & other )
```

Copy constructor.
Allocates new memory and copies all elements by value.

Parameters

other	the DArray to be copied.

Definition at line 133 of file DArray.h.

12.115.2.3 ∼DArray()

```
template<class Data >
Util::DArray< Data >::∼DArray [virtual]
```

Destructor.
Deletes underlying C array, if allocated previously.
Definition at line 150 of file DArray.h.

12.115.3 Member Function Documentation

12.115.3.1 operator=()

```
template<class Data >
DArray< Data > & Util::DArray< Data >::operator= (
            DArray< Data > const & other )
```

Assignment operator.
If this DArray is not allocated, allocates and copies all elements.
If this and the other DArray are both allocated, the capacities must be exactly equal. If so, this method copies all elements.

Parameters

other	the RHS DArray

Definition at line 169 of file DArray.h.

12.115.3.2 allocate()

```
template<class Data >
void Util::DArray< Data >::allocate (
            int capacity )
```

Allocate the underlying C array.

Exceptions

Exception	if the DArray is already allocated.

Parameters

capacity	number of elements to allocate.

Definition at line 201 of file DArray.h.

Referenced by Pscf::TridiagonalSolver::allocate(), Util::CardinalBSpline::CardinalBSpline(), Pscf::Homogeneous::Mixture::computePhi(), Pscf::Pspg::FieldIo< D >::convertBasisToRGrid(), Pscf::Pspg::FieldIo< D >::convertRGridToBasis(), Util::Polynomial< double >::differentiate(), Util::Distribution::Distribution(), Util::IntDistribution::IntDistribution(), Util::Polynomial< double >::integrate(), Util::IntDistribution::operator=(), Util::Distribution::operator=(), Pscf::Pspg::FieldIo< D >::readFieldsBasis(), Pscf::Pspg::FieldIo< D >::readFieldsKGrid(), Pscf::Pspg::FieldIo< D >::readFieldsRGrid(), Util::RadialDistribution::readParameters(), Pscf::Homogeneous::Mixture::readParameters(), Util::IntDistribution::readParameters(), Util::Distribution::readParameters(), Pscf::Homogeneous::Mixture::setNMolecule(), Pscf::Homogeneous::Mixture::setNMonomer(), Util::RadialDistribution::setParam(), Util::IntDistribution::setParam(), Util::Distribution::setParam(), Pscf::PolymerTmpl< Block< D > >::solve(), Pscf::Pspg::FieldIo< D >::writeFieldsBasis(), Pscf::Pspg::FieldIo< D >::writeFieldsKGrid(), and Pscf::Pspg::FieldIo< D >::writeFieldsRGrid().

12.115.3.3 deallocate()

```
template<class Data >
void Util::DArray< Data >::deallocate
```

Dellocate the underlying C array.

Exceptions

Exception	if the DArray is not allocated.

Definition at line 219 of file DArray.h.

Referenced by Pscf::Pspg::FieldIo< D >::convertBasisToRGrid(), Pscf::Pspg::FieldIo< D >::convertRGridToBasis(), and Pscf::PolymerTmpl< Block< D > >::solve().

12.115.3.4 reallocate()

```
template<class Data >
void Util::DArray< Data >::reallocate (
            int capacity )
```

Reallocate the underlying C array and copy to new location.

The new capacity, given by the capacity parameter, must be greater than the existing array capacity.

Parameters

capacity	

Definition at line 232 of file DArray.h.

12.115.3.5 isAllocated() `template<class Data >`

`bool Util::DArray< Data >::isAllocated [inline]`

Return true if the DArray has been allocated, false otherwise.

Definition at line 249 of file DArray.h.

Referenced by Util::bcast(), Util::DArray< Pscf::Monomer >::DArray(), Util::DArray< Pscf::Monomer >::operator=(), Util::recv(), and Util::send().

12.115.3.6 serialize() `template<class Data >`

```
template<class Archive >
void Util::DArray< Data >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize a DArray to/from an Archive.

Parameters

ar	archive
version	archive version id

Definition at line 257 of file DArray.h.

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

- DArray.h

12.116 Util::DArrayParam< Type > Class Template Reference

A Parameter associated with a DArray container.

`#include <DArrayParam.h>`

Inheritance diagram for Util::DArrayParam< Type >:

classUtil_1_1DArrayParam-eps-converted-to.pdf

Public Member Functions

- void writeParam (std::ostream &out)

    Write parameter to stream.

Protected Member Functions

- virtual void readValue (std::istream &in)

    Read parameter value from an input stream.
- virtual void loadValue (Serializable::IArchive &ar)

    Load bare parameter value from an archive.
- virtual void saveValue (Serializable::OArchive &ar)

    Save parameter value to an archive.
- virtual void bcastValue ()

    Broadcast parameter value within the ioCommunicator.

Additional Inherited Members

12.116.1 Detailed Description

template<class Type>
class Util::DArrayParam< Type >

A Parameter associated with a DArray container.
Definition at line 26 of file DArrayParam.h.

12.116.2 Member Function Documentation

12.116.2.1 writeParam() `template<class Type >`

```
void Util::DArrayParam< Type >::writeParam (
            std::ostream & out )  [virtual]
```

Write parameter to stream.

Parameters

out	output stream

Implements Util::ParamComponent.
Definition at line 153 of file DArrayParam.h.
References Util::Parameter::Precision, UTIL_THROW, and Util::Parameter::Width.

12.116.2.2 readValue() `template<class Type >`

```
void Util::DArrayParam< Type >::readValue (
            std::istream & in )  [protected], [virtual]
```

Read parameter value from an input stream.

Parameters

in	input stream from which to read

Reimplemented from Util::Parameter.
Definition at line 97 of file DArrayParam.h.
References UTIL_THROW.

12.116.2.3 loadValue() `template<class Type >`

`void Util::DArrayParam< Type >::loadValue ( Serializable::IArchive & ar ) [protected], [virtual]`

Load bare parameter value from an archive.

Parameters

ar	input archive from which to load

Reimplemented from Util::Parameter.

Definition at line 114 of file DArrayParam.h.

References UTIL_THROW.

12.116.2.4 saveValue() `template<class Type >`

`void Util::DArrayParam< Type >::saveValue ( Serializable::OArchive & ar ) [protected], [virtual]`

Save parameter value to an archive.

Parameters

ar	output archive to which to save

Reimplemented from Util::Parameter.

Definition at line 129 of file DArrayParam.h.

References UTIL_THROW.

12.116.2.5 bcastValue() `template<class Type >`

`void Util::DArrayParam< Type >::bcastValue [protected], [virtual]`

Broadcast parameter value within the ioCommunicator.

Reimplemented from Util::Parameter.

Definition at line 145 of file DArrayParam.h.

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

- DArrayParam.h

12.117 Util::Dbl Class Reference

Wrapper for a double precision number, for formatted ostream output.

`#include <Dbl.h>`

Public Member Functions

Constructors

- Dbl ()
  Default constructor.
- Dbl (double value)
  Constructor, value only.
- Dbl (double value, int width)
  Constructor, value and width.
- Dbl (double value, int width, int precision, bool isFixed=false)
  Constructor: value, width, precision, and format.

Mutators

- void setValue (double value)

  Set value of associated double.
- void setWidth (int width)

  Set output field width.
- void setPrecision (int precision)

  Set output floating point precision.

Accessors

- double value ()

  Get value of associated double.
- int width ()

  Get field width.
- int precision ()

  Get floating point precision.
- std::istream & operator>> (std::istream &in, Dbl &object)

  Input stream extractor for an Dbl object.
- std::ostream & operator<< (std::ostream &out, const Dbl &object)

  Output stream inserter for an Dbl object.

12.117.1 Detailed Description

Wrapper for a double precision number, for formatted ostream output.
An Dbl object has a double precision numerical value, as well as members (width and precision) that control its output format. The << operator for an Dbl object uses the specified width and precision. The double precision number, the width and the precision may all be specified as parameters to one of several constructors. Values of width and precision that are not specified as parameters of a constructor are set within the constructor to default values given by Format::defaultWidth() and Format::defaultPrecision(), respectively.
A Dbl object may be passed to an ostream as a temporary object. For example, the expression:

```
std::cout « Dbl(2.0) « Dbl(3.0, 15, 8) « std::endl;
```

outputs the number 2.0 using the default width and precision, followed by the number 3.0 in a field of minimum width 15 and precision 8.
Definition at line 39 of file Dbl.h.

12.117.2 Constructor & Destructor Documentation

12.117.2.1 Dbl() [1/4] `Util::Dbl::Dbl ( )`

Default constructor.
Definition at line 17 of file Dbl.cpp.

12.117.2.2 Dbl() [2/4] `Util::Dbl::Dbl ( double value ) [explicit]`

Constructor, value only.
Definition at line 25 of file Dbl.cpp.

12.117.2.3 Dbl() [3/4] `Util::Dbl::Dbl (`

`double value,`
`int width )`

Constructor, value and width.

Definition at line 33 of file Dbl.cpp.

12.117.2.4 Dbl() [4/4] `Util::Dbl::Dbl (`

`double value,`
`int width,`
`int precision,`
`bool isFixed = false )`

Constructor: value, width, precision, and format.

Definition at line 41 of file Dbl.cpp.

12.117.3 Member Function Documentation

12.117.3.1 setValue() `void Util::Dbl::setValue (`

`double value )`

Set value of associated double.

Definition at line 48 of file Dbl.cpp.

References value().

12.117.3.2 setWidth() `void Util::Dbl::setWidth (`

`int width )`

Set output field width.

Definition at line 51 of file Dbl.cpp.

References width().

12.117.3.3 setPrecision() `void Util::Dbl::setPrecision (`

`int precision )`

Set output floating point precision.

Definition at line 54 of file Dbl.cpp.

References precision().

12.117.3.4 value() `double Util::Dbl::value ( )`

Get value of associated double.

Definition at line 57 of file Dbl.cpp.

Referenced by setValue().

12.117.3.5 width() `int Util::Dbl::width ( )`

Get field width.

Definition at line 60 of file Dbl.cpp.

Referenced by setWidth().

12.117.3.6 precision() `int Util::Dbl::precision ( )`

Get floating point precision.
Definition at line 63 of file Dbl.cpp.
Referenced by setPrecision().

12.117.4 Friends And Related Function Documentation

12.117.4.1 operator>> `std::istream& operator>> ( std::istream & in, Dbl & object ) [friend]`

Input stream extractor for an Dbl object.

Parameters

in	input stream
object	Dbl object to be read from stream

Returns

modified input stream

Definition at line 73 of file Dbl.cpp.

12.117.4.2 operator<< `std::ostream& operator<< ( std::ostream & out, const Dbl & object ) [friend]`

Output stream inserter for an Dbl object.

Parameters

out	output stream
object	Dbl to be written to stream

Returns

modified output stream

Definition at line 86 of file Dbl.cpp.
The documentation for this class was generated from the following files:

- Dbl.h
- Dbl.cpp

12.118 Util::Distribution Class Reference

A distribution (or histogram) of values for a real variable.
`#include <Distribution.h>`
Inheritance diagram for Util::Distribution:

classUtil_1_1Distribution-eps-converted-to.pdf

Public Member Functions

- Distribution ()

  Default constructor.
- Distribution (const Distribution &other)

  Copy constructor.
- Distribution & operator= (const Distribution &other)

  Assignment operator.
- virtual ∼Distribution ()

  Destructor.
- virtual void readParameters (std::istream &in)

  Read parameters from file and initialize.
- void setParam (double min, double max, int nBin)

  Set parameters and initialize.
- virtual void loadParameters (Serializable::IArchive &ar)

  Load internal state from an archive.
- virtual void save (Serializable::OArchive &ar)

  Save internal state to an archive.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize this Distribution to/from an archive.
- void sample (double value)

  Sample a value.
- virtual void clear ()

  Clear (i.e., zero) previously allocated histogram.
- void output (std::ostream &out)

  Output the distribution to file.
- int binIndex (double value) const

  Return the index of the bin for a value.
- double min () const

  Get minimum value in range of histogram.
- double max () const

  Get maximum value in range of histogram.
- double binWidth () const

  Get binWidth, the width of each bin.

- int nBin () const

  Get the number of bins.
- void reduce (MPI::Intracomm &communicator, int root)

  Reduce (add) distributions from multiple MPI processors.

Protected Attributes

- DArray< long > histogram_

  Histogram of occurences, one element per bin.
- double min_

  minimum value.
- double max_

  maximum value.
- double binWidth_

  width of bin = (max_-min_)/nBin_ .
- int nBin_

  number of bins.
- int nSample_

  Number of sampled values in Histogram.
- int nReject_

  Number of sampled values that were out of range.

Additional Inherited Members

12.118.1 Detailed Description

A distribution (or histogram) of values for a real variable.
Definition at line 23 of file Distribution.h.

12.118.2 Constructor & Destructor Documentation

12.118.2.1 Distribution() [1/2] `Util::Distribution::Distribution ( )`

Default constructor.
Definition at line 18 of file Distribution.cpp.
References Util::ParamComposite::setClassName().

12.118.2.2 Distribution() [2/2] `Util::Distribution::Distribution ( const Distribution & other )`

Copy constructor.

Parameters

other	Distribution to be copied.

Definition at line 31 of file Distribution.cpp.
References Util::DArray< Data >::allocate(), Util::Array< Data >::capacity(), histogram_, nBin_, nReject_, and n←Sample_.

12.118.2.3 ∼Distribution() `Util::Distribution::∼Distribution ( ) [virtual]`

Destructor.

Definition at line 94 of file Distribution.cpp.

12.118.3 Member Function Documentation

12.118.3.1 operator=() `Distribution & Util::Distribution::operator= ( const Distribution & other )`

Assignment operator.

Parameters

other	Distribution to be assigned.

Definition at line 57 of file Distribution.cpp.

References Util::DArray< Data >::allocate(), binWidth_, Util::Array< Data >::capacity(), histogram_, max_, min_, nBin_, nReject_, and nSample_.

Referenced by Util::RadialDistribution::operator=().

12.118.3.2 readParameters() `void Util::Distribution::readParameters ( std::istream & in ) [virtual]`

Read parameters from file and initialize.

Read values of min, max, and nBin from file. Allocate histogram array and clear all accumulators.

Parameters

in	input parameter file stream

Reimplemented from Util::ParamComposite.

Reimplemented in Util::RadialDistribution.

Definition at line 100 of file Distribution.cpp.

References Util::DArray< Data >::allocate(), binWidth_, clear(), histogram_, max_, min_, and nBin_.

12.118.3.3 setParam() `void Util::Distribution::setParam ( double min, double max, int nBin )`

Set parameters and initialize.

Parameters

min	lower bound of range
max	upper bound of range
nBin	number of bins in range [min, max]

Definition at line 117 of file Distribution.cpp.

References Util::DArray< Data >::allocate(), binWidth_, clear(), histogram_, max(), max_, min(), min_, nBin(), and nBin_.

12.118.3.4 loadParameters() `void Util::Distribution::loadParameters ( Serializable::IArchive & ar ) [virtual]`

Load internal state from an archive.

Parameters

ar	input/loading archive

Reimplemented from Util::ParamComposite.
Reimplemented in Util::RadialDistribution.
Definition at line 130 of file Distribution.cpp.
References binWidth_, Util::Array< Data >::capacity(), Util::feq(), histogram_, max_, min_, nBin_, nReject_, nSample←
_, and UTIL_THROW.

12.118.3.5 save() `void Util::Distribution::save ( Serializable::OArchive & ar ) [virtual]`

Save internal state to an archive.

Parameters

ar	output/saving archive

Reimplemented from Util::ParamComposite.
Reimplemented in Util::RadialDistribution.
Definition at line 152 of file Distribution.cpp.

12.118.3.6 serialize() `template<class Archive >`

```
void Util::Distribution::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize this Distribution to/from an archive.

Parameters

ar	input or output archive
version	file version id

Definition at line 198 of file Distribution.h.
References binWidth_, Util::Array< Data >::capacity(), Util::feq(), histogram_, max_, min_, nBin_, nReject_, nSample←
_, and UTIL_THROW.
Referenced by Util::RadialDistribution::serialize().

12.118.3.7 sample() `void Util::Distribution::sample ( double value )`

Sample a value.

Parameters

value	current value

Definition at line 170 of file Distribution.cpp.
References binIndex(), histogram_, max_, min_, nReject_, and nSample_.

12.118.3.8 clear() `void Util::Distribution::clear ( ) [virtual]`

Clear (i.e., zero) previously allocated histogram.
Reimplemented in Util::RadialDistribution.
Definition at line 158 of file Distribution.cpp.
References histogram_, nBin_, nReject_, and nSample_.
Referenced by Util::RadialDistribution::clear(), readParameters(), and setParam().

12.118.3.9 output() `void Util::Distribution::output ( std::ostream & out )`

Output the distribution to file.

Parameters

out	output stream

Definition at line 185 of file Distribution.cpp.
References binWidth_, histogram_, min_, nBin_, and nSample_.

12.118.3.10 binIndex() `int Util::Distribution::binIndex ( double value ) const [inline]`

Return the index of the bin for a value.

Parameters

value	sampled value

Definition at line 167 of file Distribution.h.
References binWidth_, and min_.
Referenced by sample().

12.118.3.11 min() `double Util::Distribution::min ( ) const [inline]`

Get minimum value in range of histogram.
Definition at line 173 of file Distribution.h.
References min_.
Referenced by setParam().

12.118.3.12 max() `double Util::Distribution::max ( ) const [inline]`

Get maximum value in range of histogram.
Definition at line 179 of file Distribution.h.
References max_.

Referenced by Util::RadialDistribution::setParam(), and setParam().

12.118.3.13 binWidth() `double Util::Distribution::binWidth ( ) const [inline]`

Get binWidth, the width of each bin.
Definition at line 185 of file Distribution.h.
References binWidth_.

12.118.3.14 nBin() `int Util::Distribution::nBin ( ) const [inline]`

Get the number of bins.
Definition at line 191 of file Distribution.h.
References nBin_.
Referenced by Util::RadialDistribution::setParam(), and setParam().

12.118.3.15 reduce()

```
void Util::Distribution::reduce (
        MPI::Intracomm & communicator,
        int root )
```

Reduce (add) distributions from multiple MPI processors.

Parameters

communicator	MPI communicator
root	rank of MPI root processor for reduction

Definition at line 200 of file Distribution.cpp.
References Util::Array< Data >::cArray(), histogram_, nBin_, nReject_, and nSample_.

12.118.4 Member Data Documentation

12.118.4.1 histogram_ `DArray<long> Util::Distribution::histogram_ [protected]`

Histogram of occurences, one element per bin.
Definition at line 152 of file Distribution.h.
Referenced by clear(), Distribution(), Util::RadialDistribution::loadParameters(), loadParameters(), operator=(), output(), Util::RadialDistribution::output(), Util::RadialDistribution::readParameters(), readParameters(), reduce(), sample(), serialize(), Util::RadialDistribution::setParam(), and setParam().

12.118.4.2 min_ `double Util::Distribution::min_ [protected]`

minimum value.
Definition at line 153 of file Distribution.h.
Referenced by binIndex(), Util::RadialDistribution::loadParameters(), loadParameters(), min(), operator=(), output(), Util::RadialDistribution::readParameters(), readParameters(), sample(), serialize(), Util::RadialDistribution::setParam(), and setParam().

12.118.4.3 max_ `double Util::Distribution::max_ [protected]`

maximum value.
Definition at line 154 of file Distribution.h.

Referenced by Util::RadialDistribution::loadParameters(), loadParameters(), max(), operator=(), Util::Radial↩Distribution::readParameters(), readParameters(), sample(), serialize(), Util::RadialDistribution::setParam(), and set↩Param().

12.118.4.4 binWidth_ `double Util::Distribution::binWidth_ [protected]`

width of bin = (max_-min_)/nBin_ .

Definition at line 155 of file Distribution.h.

Referenced by binIndex(), binWidth(), Util::RadialDistribution::loadParameters(), loadParameters(), operator=(), output(), Util::RadialDistribution::output(), Util::RadialDistribution::readParameters(), readParameters(), serialize(), Util::↩RadialDistribution::setParam(), and setParam().

12.118.4.5 nBin_ `int Util::Distribution::nBin_ [protected]`

number of bins.

Definition at line 156 of file Distribution.h.

Referenced by clear(), Distribution(), Util::RadialDistribution::loadParameters(), loadParameters(), nBin(), operator=(), output(), Util::RadialDistribution::output(), Util::RadialDistribution::readParameters(), readParameters(), reduce(), serialize(), Util::RadialDistribution::setParam(), and setParam().

12.118.4.6 nSample_ `int Util::Distribution::nSample_ [protected]`

Number of sampled values in Histogram.

Definition at line 157 of file Distribution.h.

Referenced by clear(), Distribution(), Util::RadialDistribution::loadParameters(), loadParameters(), operator=(), output(), reduce(), sample(), and serialize().

12.118.4.7 nReject_ `int Util::Distribution::nReject_ [protected]`

Number of sampled values that were out of range.

Definition at line 158 of file Distribution.h.

Referenced by clear(), Distribution(), Util::RadialDistribution::loadParameters(), loadParameters(), operator=(), reduce(), sample(), and serialize().

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

- Distribution.h
- Distribution.cpp

12.119 Util::DMatrix< Data > Class Template Reference

Dynamically allocated Matrix.

`#include <DMatrix.h>`

Inheritance diagram for Util::DMatrix< Data >:

classUtil_1_1DMatrix-eps-converted-to.pdf

Public Member Functions

- DMatrix ()

Constructor.
- DMatrix (const DMatrix< Data > &other)

  Copy constructor.
- DMatrix< Data > & operator= (const DMatrix< Data > &other)

  Assignment.
- ~DMatrix ()

  Destructor.
- void allocate (int capacity1, int capacity2)

  Allocate memory for a matrix.
- void deallocate ()

  Deallocate the underlying memory block.
- bool isAllocated () const

  Return true if the DMatrix has been allocated, false otherwise.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize a DMatrix to/from an Archive.

Additional Inherited Members

12.119.1 Detailed Description

template<typename Data>
class Util::DMatrix< Data >

Dynamically allocated Matrix.
Definition at line 24 of file DMatrix.h.

12.119.2 Constructor & Destructor Documentation

12.119.2.1 DMatrix() [1/2] `template<typename Data >`

`Util::DMatrix< Data >::DMatrix`
Constructor.
Definition at line 94 of file DMatrix.h.

12.119.2.2 DMatrix() [2/2] `template<typename Data >`

`Util::DMatrix< Data >::DMatrix ( const DMatrix< Data > & other )`
Copy constructor.
Definition at line 102 of file DMatrix.h.

12.119.2.3 ~DMatrix() `template<typename Data >`

`Util::DMatrix< Data >::~DMatrix`
Destructor.
Delete dynamically allocated C array.
Definition at line 156 of file DMatrix.h.

12.119.3 Member Function Documentation

12.119.3.1 operator=() template<typename Data >

`DMatrix< Data > & Util::DMatrix< Data >::operator= ( const DMatrix< Data > & other )`

Assignment.

Exceptions

Exception	if LHS and RHS dimensions do not match.

Definition at line 121 of file DMatrix.h.

12.119.3.2 allocate() template<typename Data >

`void Util::DMatrix< Data >::allocate ( int capacity1, int capacity2 )`

Allocate memory for a matrix.

Parameters

capacity1	number of rows (range of first index)
capacity2	number of columns (range of second index)

Definition at line 170 of file DMatrix.h.

Referenced by Pscf::Homogeneous::Mixture::computePhi(), Pscf::ChiInteraction::readParameters(), and Util::DSymm↩ MatrixParam< Type >::readValue().

12.119.3.3 deallocate() template<class Data >

`void Util::DMatrix< Data >::deallocate`

Deallocate the underlying memory block.

Exceptions

Exception	if the DMatrix is not allocated.

Definition at line 188 of file DMatrix.h.

12.119.3.4 isAllocated() template<class Data >

`bool Util::DMatrix< Data >::isAllocated [inline]`

Return true if the DMatrix has been allocated, false otherwise.

Definition at line 202 of file DMatrix.h.

Referenced by Util::bcast(), Util::recv(), and Util::send().

12.119.3.5 serialize() template<class Data >

`template<class Archive > void Util::DMatrix< Data >::serialize ( Archive & ar, const unsigned int version )`

Serialize a DMatrix to/from an Archive.

Parameters

ar	archive
version	archive version id

Definition at line 210 of file DMatrix.h.
The documentation for this class was generated from the following file:

- DMatrix.h

12.120 Util::DMatrixParam< Type > Class Template Reference

A Parameter associated with a 2D built-in C array.
`#include <DMatrixParam.h>`
Inheritance diagram for Util::DMatrixParam< Type >:

classUtil_1_1DMatrixParam-eps-converted-to.pdf

Public Member Functions

- DMatrixParam (const char *label, DMatrix< Type > &matrix, int m, int n, bool isRequired=true)

    Constructor.
- void writeParam (std::ostream &out)

    Write DMatrix to file.

Protected Member Functions

- virtual void readValue (std::istream &in)

    Read parameter value from an input stream.
- virtual void loadValue (Serializable::IArchive &ar)

    Load bare parameter value from an archive.
- virtual void saveValue (Serializable::OArchive &ar)

    Save parameter value to an archive.
- virtual void bcastValue ()

    Broadcast parameter value within the ioCommunicator.

Additional Inherited Members

12.120.1 Detailed Description

template<class Type>
class Util::DMatrixParam< Type >

A Parameter associated with a 2D built-in C array.
Definition at line 29 of file DMatrixParam.h.

12.120.2 Constructor & Destructor Documentation

12.120.2.1 DMatrixParam() template<class Type >

Util::DMatrixParam< Type >::DMatrixParam (
    const char * *label*,
    DMatrix< Type > & *matrix*,
    int *m*,
    int *n*,
    bool *isRequired* = *true*)

Constructor.

Parameters

label	parameter label (a literal C-string)
matrix	DMatrix<Type> object
m	number of rows
n	number of columns
isRequired	Is this a required parameter?

Definition at line 97 of file DMatrixParam.h.

12.120.3 Member Function Documentation

12.120.3.1 writeParam() template<class Type >

void Util::DMatrixParam< Type >::writeParam (
    std::ostream & *out*) [virtual]

Write DMatrix to file.
Implements Util::ParamComponent.
Definition at line 188 of file DMatrixParam.h.
References Util::Parameter::Precision, UTIL_THROW, and Util::Parameter::Width.

12.120.3.2 readValue() template<class Type >

void Util::DMatrixParam< Type >::readValue (
    std::istream & *in*) [protected], [virtual]

Read parameter value from an input stream.

Parameters

in	input stream from which to read

Reimplemented from Util::Parameter.
Definition at line 108 of file DMatrixParam.h.
References UTIL_THROW.

12.120.3.3 loadValue() template<class Type >

void Util::DMatrixParam< Type >::loadValue (
    Serializable::IArchive & *ar*) [protected], [virtual]

Load bare parameter value from an archive.

Parameters

ar	input archive from which to load

Reimplemented from Util::Parameter.
Definition at line 133 of file DMatrixParam.h.
References UTIL_THROW.

12.120.3.4 saveValue() template<class Type >

void Util::DMatrixParam< Type >::saveValue (Serializable::OArchive & *ar*) [protected], [virtual]

Save parameter value to an archive.

Parameters

ar	output archive to which to save

Reimplemented from Util::Parameter.
Definition at line 152 of file DMatrixParam.h.
References UTIL_THROW.

12.120.3.5 bcastValue() template<class Type >

void Util::DMatrixParam< Type >::bcastValue [protected], [virtual]

Broadcast parameter value within the ioCommunicator.
Reimplemented from Util::Parameter.
Definition at line 168 of file DMatrixParam.h.
References UTIL_THROW.
The documentation for this class was generated from the following file:

- DMatrixParam.h

12.121 Util::DPArray< Data > Class Template Reference

A dynamic array that only holds pointers to its elements.
#include <DPArray.h>
Inheritance diagram for Util::DPArray< Data >:

classUtil_1_1DPArray-eps-converted-to.pdf

Public Member Functions

- DPArray ()

  Constructor.
- DPArray (const DPArray< Data > &other)

Copy constructor, copy pointers.

- virtual ∼DPArray ()

  Destructor.
- DPArray< Data > & operator= (const DPArray< Data > &other)

  Assignment, element by element.
- void allocate (int capacity)

  Allocate an array of pointers to Data.
- void append (Data &data)

  Append an element to the end of the sequence.
- void clear ()

  Reset to empty state.
- bool isAllocated () const

  Is this DPArray allocated?

Additional Inherited Members

12.121.1 Detailed Description

template<typename Data>
class Util::DPArray< Data >

A dynamic array that only holds pointers to its elements.
Definition at line 24 of file DPArray.h.

12.121.2 Constructor & Destructor Documentation

12.121.2.1 DPArray() [1/2] `template<typename Data >`

`Util::DPArray< Data >::DPArray [inline]`

Constructor.
Definition at line 101 of file DPArray.h.

12.121.2.2 DPArray() [2/2] `template<typename Data >`

```
Util::DPArray< Data >::DPArray (
              const DPArray< Data > & other )
```

Copy constructor, copy pointers.
Allocates new Data∗ array and copies pointers to Data objects.

Parameters

other	the DPArray to be copied.

Allocates a new Data∗ array and copies all pointer values.

Parameters

other	the DPArray to be copied.

Definition at line 113 of file DPArray.h.
References Util::PArray< Data >::capacity_, Util::DPArray< Data >::isAllocated(), Util::PArray< Data >::ptrs_, Util::↩

PArray< Data >::size_, and UTIL_THROW.

12.121.2.3 ∼DPArray() template<typename Data >

`Util::DPArray< Data >::∼DPArray [virtual]`

Destructor.

Deletes array of pointers, if allocated previously. Does not delete the associated Data objects.

Definition at line 179 of file DPArray.h.

12.121.3 Member Function Documentation

12.121.3.1 operator=() template<typename Data >

```
DPArray< Data > & Util::DPArray< Data >::operator= (
            const DPArray< Data > & other )
```

Assignment, element by element.

Preconditions:

- Both this and other DPArrays must be allocated.
- Capacity of this DPArray must be >= size of RHS DPArray.

Parameters

other	the rhs DPArray

Definition at line 141 of file DPArray.h.

References Util::PArray< Data >::ptrs_, Util::PArray< Data >::size_, and UTIL_THROW.

12.121.3.2 allocate() template<typename Data >

```
void Util::DPArray< Data >::allocate (
            int capacity )
```

Allocate an array of pointers to Data.

Throw an Exception if the DPArray has already been allocated - A DPArray can only be allocated once.

Parameters

capacity	number of elements to allocate.

Definition at line 193 of file DPArray.h.

References UTIL_THROW.

12.121.3.3 append() template<typename Data >

```
void Util::DPArray< Data >::append (
            Data & data ) [inline]
```

Append an element to the end of the sequence.

Parameters

data	Data object to be appended

Definition at line 210 of file DPArray.h.
References UTIL_THROW.

12.121.3.4 clear() `template<typename Data >`

`void Util::DPArray< Data >::clear`

Reset to empty state.
Definition at line 226 of file DPArray.h.

12.121.3.5 isAllocated() `template<typename Data >`

`bool Util::DPArray< Data >::isAllocated [inline]`

Is this DPArray allocated?
Definition at line 237 of file DPArray.h.
Referenced by Util::DPArray< Data >::DPArray().
The documentation for this class was generated from the following file:

- DPArray.h

12.122 Util::DRaggedMatrix< Data > Class Template Reference

Dynamically allocated RaggedMatrix.
`#include <DRaggedMatrix.h>`
Inheritance diagram for Util::DRaggedMatrix< Data >:

classUtil_1_1DRaggedMatrix-eps-converted-to.pdf

Public Member Functions

- DRaggedMatrix ()

    Constructor.
- ∼DRaggedMatrix ()

    Destructor.
- void allocate (const DArray< int > &rowSizes)

    Allocate memory for a ragged matrix.
- bool isAllocated () const

    Return true iff this DRaggedMatrix has been allocated.

Additional Inherited Members

12.122.1 Detailed Description

template<typename Data>
class Util::DRaggedMatrix< Data >

Dynamically allocated RaggedMatrix.
Definition at line 24 of file DRaggedMatrix.h.

12.122.2 Constructor & Destructor Documentation

12.122.2.1 DRaggedMatrix() template<typename Data >

`Util::DRaggedMatrix< Data >::DRaggedMatrix`

Constructor.

Definition at line 65 of file DRaggedMatrix.h.

12.122.2.2 ∼DRaggedMatrix() template<typename Data >

`Util::DRaggedMatrix< Data >::~DRaggedMatrix`

Destructor.

Delete dynamically allocated C array.

Definition at line 73 of file DRaggedMatrix.h.

12.122.3 Member Function Documentation

12.122.3.1 allocate() template<typename Data >

```
void Util::DRaggedMatrix< Data >::allocate (
            const DArray< int > & rowSizes )
```

Allocate memory for a ragged matrix.

Definition at line 90 of file DRaggedMatrix.h.

References Util::Array< Data >::capacity(), and UTIL_THROW.

12.122.3.2 isAllocated() template<class Data >

`bool Util::DRaggedMatrix< Data >::isAllocated [inline]`

Return true iff this DRaggedMatrix has been allocated.

Definition at line 129 of file DRaggedMatrix.h.

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

- DRaggedMatrix.h

12.123 Util::DSArray< Data > Class Template Reference

Dynamically allocated array with variable logical size.

`#include <DSArray.h>`

Public Member Functions

- DSArray ()

  Constructor.
- DSArray (const DSArray< Data > &other)

  Copy constructor.
- DSArray< Data > & operator= (const DSArray< Data > &other)

  Assignment, element by element.
- virtual ∼DSArray ()

  Destructor.
- void allocate (int capacity)

  Allocates the underlying C array.

- void append (const Data &data)

  Append data to the end of the array.
- void resize (int size)

  Modify logical size without modifying data.
- void clear ()

  Set logical size to zero.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize a DSArray to/from an Archive.
- void begin (ArrayIterator< Data > &iterator)

  Set an ArrayIterator to the beginning of this Array.
- void begin (ConstArrayIterator< Data > &iterator) const

  Set a ConstArrayIterator to the beginning of this Array.
- Data & operator[ ] (int i)

  Mimic C array subscripting.
- const Data & operator[ ] (int i) const

  Mimic C array subscripting.
- int capacity () const

  Return physical capacity of array.
- int size () const

  Return logical size of this array (i.e., number of elements).
- bool isAllocated () const

  Return true if the DSArray has been allocated, false otherwise.

Protected Attributes

- Data ∗ data_

  C array of Data elements.
- int size_

  Logical size of array (number of elements used).
- int capacity_

  Capacity (physical size) of underlying C array.

12.123.1 Detailed Description

template<typename Data>
class Util::DSArray< Data >

Dynamically allocated array with variable logical size.
A DSArray < Data > is a wrapper for a dynamically allocated C array, with continuous elements and a logical size that may be less than or equal to its physical capacity. The logical size is the number of contiguous elements that have been added using the append() method.
Definition at line 30 of file DSArray.h.

12.123.2 Constructor & Destructor Documentation

12.123.2.1 DSArray() [1/2] template<class Data >

`Util::DSArray< Data >::DSArray`

Constructor.

Definition at line 170 of file DSArray.h.

12.123.2.2 DSArray() [2/2] template<class Data >

```
Util::DSArray< Data >::DSArray (
            const DSArray< Data > & other )
```

Copy constructor.

Parameters

other	the DSArray to be copied.

Definition at line 180 of file DSArray.h.

References Util::DSArray< Data >::capacity_, Util::DSArray< Data >::data_, Util::DSArray< Data >::isAllocated(), Util::DSArray< Data >::size_, and UTIL_THROW.

12.123.2.3 ∼DSArray() template<class Data >

`Util::DSArray< Data >::∼DSArray [virtual]`

Destructor.

Definition at line 233 of file DSArray.h.

12.123.3 Member Function Documentation

12.123.3.1 operator=() template<class Data >

```
DSArray< Data > & Util::DSArray< Data >::operator= (
            const DSArray< Data > & other )
```

Assignment, element by element.

Capacity of LHS must be either zero or equal that of RHS DSArray.

Parameters

other	the RHS DSArray

Definition at line 204 of file DSArray.h.

References Util::DSArray< Data >::capacity_, Util::DSArray< Data >::isAllocated(), Util::DSArray< Data >::size_, and UTIL_THROW.

12.123.3.2 allocate() template<class Data >

```
void Util::DSArray< Data >::allocate (
            int capacity )
```

Allocates the underlying C array.

Throw an exception if the DSArray has already been allocated - A DSArray can only be allocated once.

Parameters

capacity	number of elements to allocate

Definition at line 247 of file DSArray.h.
References UTIL_THROW.

12.123.3.3 append() template<class Data >

```
void Util::DSArray< Data >::append (
            const Data & data ) [inline]
```

Append data to the end of the array.

Parameters

data	Data to add to end of array.

Definition at line 345 of file DSArray.h.
References UTIL_THROW.

12.123.3.4 resize() template<class Data >

```
void Util::DSArray< Data >::resize (
            int size ) [inline]
```

Modify logical size without modifying data.
The size parameter must be non-negative and may not exceed the physical allocated capacity.
This function simply changes the logical size without modifying any elements of the underlying physical array. When the size increases, added elements are uninitialized.

Parameters

size	new logical size, 0 <= size < capacity.

The size parameter must be non-negative and may not exceed the capacity.
This function simply changes the logical size of without modifying any elements of the underlying physical array. If the size increases, added elements are uninitialized.

Parameters

size	new logical size, 0 <= size < capacity.

Definition at line 367 of file DSArray.h.

12.123.3.5 clear() template<class Data >

```
void Util::DSArray< Data >::clear [inline]
```

Set logical size to zero.
Definition at line 374 of file DSArray.h.

12.123.3.6 serialize() template<class Data >

```
template<class Archive >
void Util::DSArray< Data >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize a DSArray to/from an Archive.

Parameters

ar	archive
version	archive version id

Definition at line 265 of file DSArray.h.
References UTIL_THROW.

12.123.3.7 begin() [1/2] template<class Data >

```
void Util::DSArray< Data >::begin (
            ArrayIterator< Data > & iterator )  [inline]
```

Set an ArrayIterator to the beginning of this Array.

Parameters

iterator	ArrayIterator, initialized on output.

Definition at line 302 of file DSArray.h.
References Util::ArrayIterator< Data >::setCurrent(), and Util::ArrayIterator< Data >::setEnd().

12.123.3.8 begin() [2/2] template<class Data >

```
void Util::DSArray< Data >::begin (
            ConstArrayIterator< Data > & iterator ) const  [inline]
```

Set a ConstArrayIterator to the beginning of this Array.

Parameters

iterator	ConstArrayIterator, initialized on output.

Definition at line 313 of file DSArray.h.
References Util::ConstArrayIterator< Data >::setCurrent(), and Util::ConstArrayIterator< Data >::setEnd().

12.123.3.9 operator[]() [1/2] template<class Data >

```
Data & Util::DSArray< Data >::operator[] (
            int i )  [inline]
```

Mimic C array subscripting.

Parameters

i	array index

Returns

reference to element i

Definition at line 323 of file DSArray.h.

12.123.3.10 operator[]() [2/2] template<class Data >

```
const Data & Util::DSArray< Data >::operator[] (
            int i ) const  [inline]
```

Mimic C array subscripting.

Parameters

i	array index

Returns

const reference to element i

Definition at line 334 of file DSArray.h.

12.123.3.11 capacity() `template<class Data >`

```
int Util::DSArray< Data >::capacity  [inline]
```

Return physical capacity of array.
Definition at line 381 of file DSArray.h.

12.123.3.12 size() `template<class Data >`

```
int Util::DSArray< Data >::size  [inline]
```

Return logical size of this array (i.e., number of elements).
Definition at line 388 of file DSArray.h.

12.123.3.13 isAllocated() `template<class Data >`

```
bool Util::DSArray< Data >::isAllocated  [inline]
```

Return true if the DSArray has been allocated, false otherwise.
Definition at line 395 of file DSArray.h.
Referenced by Util::DSArray< Data >::DSArray(), and Util::DSArray< Data >::operator=().

12.123.4 Member Data Documentation

12.123.4.1 data_ `template<typename Data >`

```
Data* Util::DSArray< Data >::data_  [protected]
```

C array of Data elements.
Definition at line 154 of file DSArray.h.
Referenced by Util::DSArray< Data >::DSArray().

12.123.4.2 size_ `template<typename Data >`

```
int Util::DSArray< Data >::size_  [protected]
```

Logical size of array (number of elements used).
Definition at line 157 of file DSArray.h.
Referenced by Util::DSArray< Data >::DSArray(), and Util::DSArray< Data >::operator=().

12.123.4.3 capacity_ template<typename Data >

int Util::DSArray< Data >::capacity_ [protected]

Capacity (physical size) of underlying C array.
Definition at line 160 of file DSArray.h.
Referenced by Util::DSArray< Data >::DSArray(), and Util::DSArray< Data >::operator=().
The documentation for this class was generated from the following file:

- DSArray.h

12.124 Util::DSymmMatrixParam< Type > Class Template Reference

A Parameter associated with a symmetric DMatrix.
#include <DSymmMatrixParam.h>
Inheritance diagram for Util::DSymmMatrixParam< Type >:

classUtil_1_1DSymmMatrixParam-eps-converted-to.pdf

Public Member Functions

- DSymmMatrixParam (const char ∗label, DMatrix< Type > &matrix, int n, bool isRequired=true)

  Constructor.
- void writeParam (std::ostream &out)

  Write DMatrix to file.

Protected Member Functions

- virtual void readValue (std::istream &in)

  Read parameter value from an input stream.
- virtual void loadValue (Serializable::IArchive &ar)

  Load bare parameter value from an archive.
- virtual void saveValue (Serializable::OArchive &ar)

  Save parameter value to an archive.
- virtual void bcastValue ()

  Broadcast parameter value within the ioCommunicator.

Additional Inherited Members

12.124.1 Detailed Description

template<class Type>
class Util::DSymmMatrixParam< Type >

A Parameter associated with a symmetric DMatrix.
Definition at line 30 of file DSymmMatrixParam.h.

12.124.2 Constructor & Destructor Documentation

12.124.2.1 DSymmMatrixParam() template<class Type >

```
Util::DSymmMatrixParam< Type >::DSymmMatrixParam (
            const char * label,
            DMatrix< Type > & matrix,
            int n,
            bool isRequired = true )
```

Constructor.

Parameters

label	parameter label (a literal C-string)
matrix	DMatrix<Type> object
n	number of rows or columns
isRequired	Is this a required parameter?

Definition at line 94 of file DSymmMatrixParam.h.

12.124.3 Member Function Documentation

12.124.3.1 writeParam() template<class Type >

```
void Util::DSymmMatrixParam< Type >::writeParam (
            std::ostream & out ) [virtual]
```

Write DMatrix to file.
Implements Util::ParamComponent.
Definition at line 201 of file DSymmMatrixParam.h.
References Util::Parameter::Precision, UTIL_THROW, and Util::Parameter::Width.

12.124.3.2 readValue() template<class Type >

```
void Util::DSymmMatrixParam< Type >::readValue (
            std::istream & in ) [protected], [virtual]
```

Read parameter value from an input stream.

Parameters

in	input stream from which to read

Reimplemented from Util::Parameter.
Definition at line 104 of file DSymmMatrixParam.h.
References Util::DMatrix< Data >::allocate(), UTIL_CHECK, and UTIL_THROW.

12.124.3.3 loadValue() template<class Type >

```
void Util::DSymmMatrixParam< Type >::loadValue (
            Serializable::IArchive & ar ) [protected], [virtual]
```

Load bare parameter value from an archive.

Parameters

ar	input archive from which to load

Reimplemented from Util::Parameter.
Definition at line 146 of file DSymmMatrixParam.h.
References UTIL_THROW.

12.124.3.4 saveValue() `template<class Type >`

`void Util::DSymmMatrixParam< Type >::saveValue ( Serializable::OArchive & ar ) [protected], [virtual]`

Save parameter value to an archive.

Parameters

ar	output archive to which to save

Reimplemented from Util::Parameter.
Definition at line 165 of file DSymmMatrixParam.h.
References UTIL_THROW.

12.124.3.5 bcastValue() `template<class Type >`

`void Util::DSymmMatrixParam< Type >::bcastValue [protected], [virtual]`

Broadcast parameter value within the ioCommunicator.
Reimplemented from Util::Parameter.
Definition at line 181 of file DSymmMatrixParam.h.
References UTIL_THROW.
The documentation for this class was generated from the following file:

- DSymmMatrixParam.h

12.125 Util::End Class Reference

End bracket of a ParamComposite parameter block.
`#include <End.h>`
Inheritance diagram for Util::End:

classUtil_1_1End-eps-converted-to.pdf

Public Member Functions

- End ()

    Constructor.
- virtual ∼End ()

Destructor.

- virtual void readParam (std::istream &in)

  Read the closing bracket.
- virtual void writeParam (std::ostream &out)

  Write the closing bracket.
- void resetParam ()

  Do-nothing implementation of virtual resetParam function.

Additional Inherited Members

12.125.1 Detailed Description

End bracket of a ParamComposite parameter block.
Definition at line 24 of file End.h.

12.125.2 Constructor & Destructor Documentation

12.125.2.1 End() `Util::End::End ( )`

Constructor.
Definition at line 17 of file End.cpp.

12.125.2.2 ∼End() `Util::End::∼End ( ) [virtual]`

Destructor.
Definition at line 24 of file End.cpp.

12.125.3 Member Function Documentation

12.125.3.1 readParam() `void Util::End::readParam ( std::istream & in ) [virtual]`

Read the closing bracket.

Parameters

in	input stream

Implements Util::ParamComponent.
Definition at line 30 of file End.cpp.
References Util::ParamComponent::echo(), Util::Log::file(), Util::MpiFileIo::isIoProcessor(), and writeParam().
Referenced by Util::ParamComposite::readEnd().

12.125.3.2 writeParam() `void Util::End::writeParam ( std::ostream & out ) [virtual]`

Write the closing bracket.

Parameters

out	output stream

Implements Util::ParamComponent.
Definition at line 43 of file End.cpp.
References Util::ParamComponent::indent().
Referenced by Util::Manager< Data >::endReadManager(), Util::ParamComposite::load(), and readParam().

12.125.3.3 resetParam() `void Util::End::resetParam ( ) [virtual]`

Do-nothing implementation of virtual resetParam function.
Reimplemented from Util::ParamComponent.
Definition at line 49 of file End.cpp.
The documentation for this class was generated from the following files:

- End.h
- End.cpp

12.126 Util::Exception Class Reference

A user-defined exception.
`#include <Exception.h>`

Public Member Functions

- Exception (const char ∗function, const char ∗message, const char ∗file, int line, int echo=1)

  Constructor.
- Exception (const char ∗message, const char ∗file, int line, int echo=1)

  Constructor without function name parameter.
- virtual ∼Exception ()

  Destructor.
- void write (std::ostream &out)

  Write error message to output stream.
- std::string & message ()

  Return the error message.

Protected Attributes

- std::string message_

  Error message string.

12.126.1 Detailed Description

A user-defined exception.
Exceptions are usually thrown using the UTIL_THROW() macro.
Definition at line 24 of file Exception.h.

12.126.2 Constructor & Destructor Documentation

12.126.2.1 Exception() [1/2] Util::Exception::Exception (

const char * *function*,
const char * *message*,
const char * *file*,
int *line*,
int *echo* = *1*)

Constructor.

Constructs error message that includes file and line number. Values of the file and line parameters should be given by the built-in macros **FILE** and **LINE**, respectively, in the calling function. A typical call of the constructor is thus of the form:

```
throw Exception("MyClass::myFunction", "A terrible thing happened!",
                __FILE__, __LINE__ );
```

Parameters

function	name of the function from which the Exception was thrown
message	message describing the nature of the error
file	name of the file from which the Exception was thrown
line	line number in file
echo	if echo, then echo to Log::file() when constructed.

Definition at line 22 of file Exception.cpp.

References Util::Log::file(), message(), message_, and write().

12.126.2.2 Exception() [2/2] Util::Exception::Exception (

const char * *message*,
const char * *file*,
int *line*,
int *echo* = *1*)

Constructor without function name parameter.

Parameters

message	message describing the nature of the error
file	name of the file from which the Exception was thrown
line	line number in file
echo	if echo, then echo to std out when constructed.

Definition at line 48 of file Exception.cpp.

References Util::Log::file(), message(), message_, and write().

12.126.2.3 ∼Exception() Util::Exception::∼Exception ( ) [virtual]

Destructor.

Definition at line 71 of file Exception.cpp.

12.126.3 Member Function Documentation

12.126.3.1 write() void Util::Exception::write (

std::ostream & *out*)

Write error message to output stream.

Parameters

out	output stream

Definition at line 77 of file Exception.cpp.
References message_.
Referenced by Exception().

12.126.3.2 message() `std::string & Util::Exception::message ( )`

Return the error message.
Definition at line 83 of file Exception.cpp.
References message_.
Referenced by Exception(), and Util::MpiThrow().

12.126.4 Member Data Documentation

12.126.4.1 message_ `std::string Util::Exception::message_ [protected]`

Error message string.
Definition at line 80 of file Exception.h.
Referenced by Exception(), message(), and write().
The documentation for this class was generated from the following files:

- Exception.h
- Exception.cpp

12.127 Util::Factory< Data > Class Template Reference

Factory template.
`#include <Factory.h>`

Public Member Functions

- Factory ()

  Constructor.
- virtual ∼Factory ()

  Destructor.
- void addSubfactory (Factory< Data > &subfactory)

  Add a new subfactory to the list.
- virtual Data ∗ factory (const std::string &className) const =0

  Returns a pointer to a new instance of specified subclass.
- Data ∗ readObject (std::istream &in, ParamComposite &parent, std::string &className, bool &isEnd)

  Read a class name, instantiate an object, and read its parameters.
- Data ∗ loadObject (Serializable::IArchive &ar, ParamComposite &parent, std::string &className)

  Load a class name, instantiate an object, and load the object.

Protected Member Functions

- Data ∗ trySubfactories (const std::string &className) const

    Search through subfactories for match.
- void setIoCommunicator (MPI::Intracomm &communicator)

    Set associated Mpi communicator.
- bool hasIoCommunicator () const

    Does this factory have a param communicator?

12.127.1 Detailed Description

template<typename Data>
class Util::Factory< Data >

Factory template.
Definition at line 32 of file Factory.h.

12.127.2 Constructor & Destructor Documentation

12.127.2.1 Factory() `template<typename Data >`

`Util::Factory< Data >::Factory`

Constructor.
Definition at line 165 of file Factory.h.

12.127.2.2 ∼Factory() `template<typename Data >`

`Util::Factory< Data >::~Factory [virtual]`

Destructor.
Definition at line 173 of file Factory.h.

12.127.3 Member Function Documentation

12.127.3.1 addSubfactory() `template<typename Data >`

```
void Util::Factory< Data >::addSubfactory (
            Factory< Data > & subfactory )
```

Add a new subfactory to the list.

Parameters

subfactory	New subfactory to be added

Definition at line 204 of file Factory.h.

12.127.3.2 factory() `template<typename Data >`

```
virtual Data* Util::Factory< Data >::factory (
            const std::string & className ) const [pure virtual]
```

Returns a pointer to a new instance of specified subclass.
This method takes the name className of a subclass of Data as a parameter, and attempts to instantiate an object of

that class. If it recognizes the className, it creates an instance of that class and returns a Data∗ base class pointer to the new object. If it does not recognize the className, it returns a null pointer.
An implementation should first call "trySubfactories(className)" and immediately return if this returns a non-null pointer, before attempting to match the className against specific strings.

Parameters

className	name of subclass

Returns

base class pointer to new object, or a null pointer.

12.127.3.3 readObject()

```
template<typename Data >
Data * Util::Factory< Data >::readObject (
            std::istream & in,
            ParamComposite & parent,
            std::string & className,
            bool & isEnd )
```

Read a class name, instantiate an object, and read its parameters.
This method:

- reads a comment line of the form className + {
- invokes the factory method to create an instance of className
- invokes the readParam() method of the new object

When compiled with MPI, if the parent ParamComposite has a param communicator, this method reads the comment line on the Io processor, broadcasts it to all others, and then lets each processor independently match this string.

Exceptions

Exception	if className is not recognized.

Parameters

in	input stream
parent	parent ParamComposite object
className	(output) name of subclass of Data
isEnd	(output) is the input a closing bracket "}" ?

Returns

pointer to new instance of className

Definition at line 214 of file Factory.h.
References Util::ParamComposite::addParamComposite(), Util::ParamComponent::echo(), Util::Log::file(), Util::MpiFileIo::hasIoCommunicator(), Util::MpiFileIo::ioCommunicator(), Util::ParamComponent::setIndent(), UTIL_THROW, and Util::Begin::writeParam().

12.127.3.4 loadObject() `template<typename Data >`

```
Data * Util::Factory< Data >::loadObject (
            Serializable::IArchive & ar,
            ParamComposite & parent,
            std::string & className )
```

Load a class name, instantiate an object, and load the object.

This method:

- loads a className from an input archive
- invokes the factory method to create an instance of className
- invokes the load() method of the new object

When compiled with MPI, if the parent ParamComposite has a param communicator, this method loads the comment line on the Io processor, broadcasts it to all others, and then lets each processor independently match this string.

Exceptions

Exception	if className is not recognized.

Parameters

ar	input/loading archive
parent	parent ParamComposite object
className	(output) name of subclass of Data

Returns

pointer to new instance of className

Definition at line 303 of file Factory.h.

References Util::Log::file(), Util::MpiFileIo::hasIoCommunicator(), Util::MpiFileIo::ioCommunicator(), and Util::ParamComposite::loadParamComposite().

12.127.3.5 trySubfactories() `template<typename Data >`

```
Data * Util::Factory< Data >::trySubfactories (
            const std::string & className ) const [protected]
```

Search through subfactories for match.

This method iterates through all registered subfactories, calls the factory(const std::string&) method of each, and immediately returns a pointer to a new object if any of them returns a non-null pointer. If all of them return a null pointer, this method also returns a null pointer.

Parameters

className	name of subclass

Returns

base class pointer to new object, or a null pointer.

Definition at line 340 of file Factory.h.

12.127.3.6 setIoCommunicator() `template<typename Data >`

`void Util::Factory< Data >::setIoCommunicator ( MPI::Intracomm & communicator ) [protected]`

Set associated Mpi communicator.
Is not recursive (is not applied to subfactories).

Parameters

communicator	MPI Intra-communicator to use for input

Definition at line 181 of file Factory.h.
References UTIL_THROW.

12.127.3.7 hasIoCommunicator() `template<typename Data >`

`bool Util::Factory< Data >::hasIoCommunicator [protected]`

Does this factory have a param communicator?
Definition at line 196 of file Factory.h.
The documentation for this class was generated from the following file:

- Factory.h

12.128 Util::FArray< Data, Capacity > Class Template Reference

A fixed size (static) contiguous array template.
`#include <FArray.h>`

Public Member Functions

- FArray ()
  Constructor.
- FArray (const FArray< Data, Capacity > &other)
  Copy constructor.
- FArray< Data, Capacity > & operator= (const FArray< Data, Capacity > &other)
  Assignment, element by element.
- int size () const
  Return number of elements in this FArray.
- int capacity () const
  Return number of elements in this FArray.
- void begin (ArrayIterator< Data > &iterator)
  Set an ArrayIterator to the beginning of this Array.
- void begin (ConstArrayIterator< Data > &iterator) const
  Set a ConstArrayIterator to the beginning of this Array.
- Data & operator[ ] (int i)
  Mimic C array subscripting.
- const Data & operator[ ] (int i) const
  Mimic C array subscripting.
- Data ∗ cArray ()
  Return pointer to underlying C array.
- const Data ∗ cArray () const
  Return pointer to const to underlying C array.

- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize a FArray to/from an Archive.
- int packedSize ()

  Return packed size in a MemoryArchive, in bytes.

Static Public Member Functions

- static void commitMpiType ()

  Commit associated MPI DataType.

12.128.1 Detailed Description

template<typename Data, int Capacity>
class Util::FArray< Data, Capacity >

A fixed size (static) contiguous array template.
An FArray is a simple wraper for a fixed size C Array, with a capacity that is fixed at compile time. As in a C Array, or a DArray container, all of the elements are accessible. Unlike an FSArray, an FArray does not have logical size that is distinct from its physical capacity.
When bounds checking is on (i.e., when NDEBUG is not defined), the operator [] checks that the index is non-negative and less than the Capacity.
Advice: Use an FArray if you know exactly how many elements will be needed at compile time. Use an FSArray when you need a small statically allocated array for which the maximum capacity needed is known at compile time, but the logical size may be less than the capacity. Use a DArray if you need a large, dynamically allocated array that must be allocated after instantiation.
Definition at line 46 of file FArray.h.

12.128.2 Constructor & Destructor Documentation

12.128.2.1 FArray() [1/2] `template<typename Data , int Capacity>`

```
Util::FArray< Data, Capacity >::FArray
```

Constructor.
Definition at line 156 of file FArray.h.

12.128.2.2 FArray() [2/2] `template<typename Data , int Capacity>`

```
Util::FArray< Data, Capacity >::FArray (
            const FArray< Data, Capacity > & other )
```

Copy constructor.

Parameters

other	the FArray to be copied.

Definition at line 165 of file FArray.h.

12.128.3 Member Function Documentation

12.128.3.1 operator=() `template<typename Data , int Capacity>`

`FArray< Data, Capacity > & Util::FArray< Data, Capacity >::operator= ( const FArray< Data, Capacity > & other )`

Assignment, element by element.

Capacity of LHS FArray must be >= size of RHS FArray.

Parameters

other	the RHS FArray

Definition at line 181 of file FArray.h.

12.128.3.2 size() `template<typename Data , int Capacity>`

`int Util::FArray< Data, Capacity >::size [inline]`

Return number of elements in this FArray.

Definition at line 200 of file FArray.h.

12.128.3.3 capacity() `template<typename Data , int Capacity>`

`int Util::FArray< Data, Capacity >::capacity [inline]`

Return number of elements in this FArray.

Definition at line 207 of file FArray.h.

12.128.3.4 begin() [1/2] `template<typename Data , int Capacity>`

`void Util::FArray< Data, Capacity >::begin ( ArrayIterator< Data > & iterator ) [inline]`

Set an ArrayIterator to the beginning of this Array.

Parameters

iterator	ArrayIterator, initialized on output.

Definition at line 214 of file FArray.h.

12.128.3.5 begin() [2/2] `template<typename Data , int Capacity>`

`void Util::FArray< Data, Capacity >::begin ( ConstArrayIterator< Data > & iterator ) const [inline]`

Set a ConstArrayIterator to the beginning of this Array.

Parameters

iterator	ConstArrayIterator, initialized on output.

Definition at line 225 of file FArray.h.

12.128.3.6 operator[]() [1/2] `template<typename Data , int Capacity>`

`Data & Util::FArray< Data, Capacity >::operator[] ( int i ) [inline]`

Mimic C array subscripting.

Parameters

i	array index

Returns

reference to element i

Definition at line 235 of file FArray.h.

12.128.3.7 operator[]() [2/2]

```
template<typename Data , int Capacity>
const Data & Util::FArray< Data, Capacity >::operator[] (
            int i ) const [inline]
```

Mimic C array subscripting.

Parameters

i	array index

Returns

const reference to element i

Definition at line 246 of file FArray.h.

12.128.3.8 cArray() [1/2]

```
template<typename Data , int Capacity>
const Data * Util::FArray< Data, Capacity >::cArray [inline]
```

Return pointer to underlying C array.

Definition at line 257 of file FArray.h.

12.128.3.9 cArray() [2/2]

```
template<typename Data , int Capacity>
const Data* Util::FArray< Data, Capacity >::cArray ( ) const
```

Return pointer to const to underlying C array.

12.128.3.10 serialize()

```
template<class Data , int Capacity>
template<class Archive >
void Util::FArray< Data, Capacity >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize a FArray to/from an Archive.

Parameters

ar	archive
version	archive version id

Definition at line 272 of file FArray.h.

12.128.3.11 packedSize() `template<typename Data , int Capacity>`

`int Util::FArray< Data, Capacity >::packedSize`

Return packed size in a MemoryArchive, in bytes.
Packed size of FArray in a MemoryArchive, in bytes.
Definition at line 284 of file FArray.h.

12.128.3.12 commitMpiType() `template<typename Data , int Capacity>`

`void Util::FArray< Data, Capacity >::commitMpiType [static]`

Commit associated MPI DataType.
Definition at line 292 of file FArray.h.
The documentation for this class was generated from the following file:

- FArray.h

12.129 Util::FArrayParam< Type, N > Class Template Reference

A Parameter associated with a FArray container.
`#include <FArrayParam.h>`
Inheritance diagram for Util::FArrayParam< Type, N >:

classUtil_1_1FArrayParam-eps-converted-to.pdf

Public Member Functions

- FArrayParam (const char ∗label, FArray< Type, N > &array, bool isRequired=true)

    Constructor.
- void writeParam (std::ostream &out)

    Write FArray parameter to stream.

Protected Member Functions

- virtual void readValue (std::istream &in)

    Read parameter value from an input stream.
- virtual void loadValue (Serializable::IArchive &ar)

    Load bare parameter value from an archive.
- virtual void saveValue (Serializable::OArchive &ar)

    Save parameter value to an archive.
- virtual void bcastValue ()

    Broadcast parameter value within the ioCommunicator.

Additional Inherited Members

12.129.1 Detailed Description

template<class Type, int N>
class Util::FArrayParam< Type, N >

A Parameter associated with a FArray container.
Definition at line 26 of file FArrayParam.h.

12.129.2 Constructor & Destructor Documentation

12.129.2.1 FArrayParam() `template<class Type , int N>`

```
Util::FArrayParam< Type, N >::FArrayParam (
            const char * label,
            FArray< Type, N > & array,
            bool isRequired = true )
```

Constructor.

Parameters

label	label string for parameter file
array	associated FArray variable
isRequired	Is this a required parameter?

Definition at line 88 of file FArrayParam.h.

12.129.3 Member Function Documentation

12.129.3.1 writeParam() `template<class Type , int N>`

```
void Util::FArrayParam< Type, N >::writeParam (
            std::ostream & out )  [virtual]
```

Write FArray parameter to stream.

Parameters

out	output stream

Implements Util::ParamComponent.
Definition at line 131 of file FArrayParam.h.
References Util::Parameter::Precision, and Util::Parameter::Width.

12.129.3.2 readValue() `template<class Type , int N>`

```
void Util::FArrayParam< Type, N >::readValue (
            std::istream & in )  [protected], [virtual]
```

Read parameter value from an input stream.

Parameters

in	input stream from which to read

Reimplemented from Util::Parameter.
Definition at line 97 of file FArrayParam.h.

12.129.3.3 loadValue() template<class Type , int N>

```
void Util::FArrayParam< Type, N >::loadValue (
        Serializable::IArchive & ar ) [protected], [virtual]
```

Load bare parameter value from an archive.

Parameters

ar	input archive from which to load

Reimplemented from Util::Parameter.
Definition at line 108 of file FArrayParam.h.

12.129.3.4 saveValue() template<class Type , int N>

```
void Util::FArrayParam< Type, N >::saveValue (
        Serializable::OArchive & ar ) [protected], [virtual]
```

Save parameter value to an archive.

Parameters

ar	output archive to which to save

Reimplemented from Util::Parameter.
Definition at line 115 of file FArrayParam.h.

12.129.3.5 bcastValue() template<class Type , int N>

```
void Util::FArrayParam< Type, N >::bcastValue [protected], [virtual]
```

Broadcast parameter value within the ioCommunicator.
Reimplemented from Util::Parameter.
Definition at line 123 of file FArrayParam.h.
The documentation for this class was generated from the following file:

- FArrayParam.h

12.130 Util::FileMaster Class Reference

A FileMaster manages input and output files for a simulation.
`#include <FileMaster.h>`
Inheritance diagram for Util::FileMaster:

classUtil_1_1FileMaster-eps-converted-to.pdf

Public Member Functions

- FileMaster ()

  Constructor.
- FileMaster (const FileMaster &copy)

  Copy constructor.
- virtual ∼FileMaster ()

  Destructor.

Initialization

- void setRootPrefix (const std::string &rootPrefix)

  Set the path from current directory to root directory.
- void setDirectoryId (int directoryId)

  Set an integer directory identifier for this processor.
- void setCommonControl ()

  Enable "replicated" mode in multi-system simulations.
- void setParamFileName (const std::string &paramFileName)

  Set the parameter file name.
- void setCommandFileName (const std::string &commandFileName)

  Set the command file name.
- void setInputPrefix (const std::string &inputPrefix)

  Set the input file prefix string.
- void setOutputPrefix (const std::string &outputPrefix)

  Set the output file prefix string.
- virtual void readParameters (std::istream &in)

  Read parameter file.

Serialization

- virtual void loadParameters (Serializable::IArchive &ar)

  Load internal state from file.
- virtual void save (Serializable::OArchive &ar)

  Save internal state to file.

File Opening

- void open (const std::string &name, std::ifstream &in, std::ios_base::openmode mode=std::ios_base::in) const

  Open an input file with a known path and open mode.
- void open (const std::string &name, std::ofstream &out, std::ios_base::openmode mode=std::ios_base::out) const

  Open an output file with a known path and open mode.
- void openControlFile (const std::string &name, std::ifstream &in) const

Open an input parameter or command file.
- void openRestartIFile (const std::string &name, std::ifstream &in, std::ios_base::openmode mode=std::ios_←base::in) const
  Open an input restart dump file for reading.
- void openRestartOFile (const std::string &name, std::ofstream &out, std::ios_base::openmode mode=std←::ios_base::out) const
  Open an output restart file for writing.
- void openInputFile (const std::string &filename, std::ifstream &in, std::ios_base::openmode mode=std::ios_←base::in) const
  Open an input file.
- void openOutputFile (const std::string &filename, std::ofstream &out, std::ios_base::openmode mode=std←::ios_base::out) const
  Open an output file.

Control Files

- bool isCommonControl () const
  Is set for common param and command files?
- std::string paramFileName () const
  Return the param file name, if any.
- std::string commandFileName () const
  Return the command file name.
- std::istream & paramFile ()
  Get a default parameter stream by reference.
- std::istream & commandFile ()
  Get the command input stream by reference.

Additional Inherited Members

12.130.1 Detailed Description

A FileMaster manages input and output files for a simulation.

File types A FileMaster manages a set of input and output files for a molecular simulation. It provides methods to open four different types of file, which are located in different places within a standard directory structure. These file types are:

- Control input files, i.e., parameter and command files
- Restart files, which can be opened for input or output
- Input data files (e.g., input configuration files)
- Output data files (e.g., trajectories and analysis output files)

Several member functions are provided to open different types of file:

- openControlFile opens a control file for reading
- openRestartIFile opens a restart file for reading
- openRestartOFile opens a restart file for writing
- openInputFile opens an input data file
- openOutputFile opens an output data file

Each of these functions takes a base file name and a file stream as an argument, constructs a complete path by prepending an appropriate prefix to the base name, and opens the file.
Slightly different directory structures are used for file paths in simulations of a single system and for parallel simulations of multiple systems, as discussed below.

Single-system simulations In simulations of a single system, paths to the different types of file are constructed by prepending some combination of the following elements before a base file name:

- Root directory prefix: This is the path from the current working directory to the root level directory for all files associated with a simulation run. It defaults to an empty string and may be reset by calling the setRootPrefix() function.
- Input prefix: This string is prepended to the base name of all input data files.
- Output prefix: This string is prepended to the base name of all output data files.

Prefix strings that represent directories must end with the directory separator character "/" in order to give a valid path when prepended to a file name. The root directory prefix must be either the empty string or such a directory path. The input and output prefix strings are often also chosen to be directory paths, in order to place input and output files in different subdirectories (e.g., "in/" and "out/").
In simulations of a single systems, paths to control and restart files are constructed by concatenating the root prefix (if any) to the base file name.
Paths to input and output files are constructed are constructed by concatenating the root prefix (if any), the input or output prefix, and the base file.

Multi-system simulations A parallel simulation of multiple systems use a directory structure in which the root directory (specified by the root directory prefix) contains a set of numbered subdirectories named "0/", "1/", "2/", etc. Each such numbered directory contains input, output and restart files that are specific to a particular system. Each such numbered directory will be referred to in what follows as the system directory for the associated system. The integer index for the system, which is also the directory name, must be set by the setDirectoryId() method before any files are opened.
In such simulations, a path to a restart files for a system is constructed by concatentaing the system directory path and the restart file base name, thus placing these files in the system directory. Paths to input and output data files are constructed by concatenating the system directory path, an input or output prefix string, and the file base name.
Two different modes of operation are possible for simulations of multiple systems, which differ in the treatment of control files. In "independent" mode, simulations of multiple systems are assumed to be completely independent and to require separate parameter and command control files for each system. In this case, the openControlFile() function for each system opens a file in the system directory for that system. In "replicated" mode, all simulations are controlled by a single parameter file and a single command file, both of which are assumed to be in the shared root directory. "Independent" mode is enabled by default. "Replicated" mode may be chosen by calling the function "setCommonControl()" before opening any any control files.

Control Files The functions setParameterFileName() and setCommandFileName() can be used to programmatically set the parameter and command file base names before reading a parameter file. If setCommandFileName() is not called before readParameters(), the readParameters() function expects to find a "commandFileName" as the first parameter in the FileMaster parameter file block.
After parameter and command file names have been set, the paramFile() and commandFile() functions return references to the files. The first time each of these functions is called, it calls openControlFile() internally to open the appropriate file.
Definition at line 142 of file FileMaster.h.

12.130.2 Constructor & Destructor Documentation

12.130.2.1 FileMaster() [1/2] `Util::FileMaster::FileMaster ( )`

Constructor.

Definition at line 23 of file FileMaster.cpp.

References Util::ParamComposite::setClassName().

12.130.2.2 FileMaster() [2/2] `Util::FileMaster::FileMaster ( const FileMaster & copy )`

Copy constructor.

Parameters

copy	FileMaster object to be copied

Definition at line 39 of file FileMaster.cpp.

12.130.2.3 ∼FileMaster() `Util::FileMaster::∼FileMaster ( ) [virtual]`

Destructor.

Definition at line 55 of file FileMaster.cpp.

12.130.3 Member Function Documentation

12.130.3.1 setRootPrefix() `void Util::FileMaster::setRootPrefix ( const std::string & rootPrefix )`

Set the path from current directory to root directory.

Parameters

rootPrefix	root directory prefix string for all paths

Definition at line 70 of file FileMaster.cpp.

12.130.3.2 setDirectoryId() `void Util::FileMaster::setDirectoryId ( int directoryId )`

Set an integer directory identifier for this processor.

This method should be called only for simulations of multiple systems, to set an integer identifier for the physical system associated with this processor. After calling this function with an integer n, a directory id prefix "n/" will be prepended to the paths of input, output and restart files associated with that system.

Parameters

directory←Id	integer subdirectory name

Definition at line 76 of file FileMaster.cpp.

12.130.3.3 setCommonControl() `void Util::FileMaster::setCommonControl ( )`

Enable "replicated" mode in multi-system simulations.
Call this function to enable the use of single parameter and command files to control simulations of multiple systems.
Definition at line 88 of file FileMaster.cpp.

12.130.3.4 setParamFileName() `void Util::FileMaster::setParamFileName ( const std::string & paramFileName )`

Set the parameter file name.

Parameters

paramFileName	name of parameter file

Definition at line 106 of file FileMaster.cpp.
References paramFileName().

12.130.3.5 setCommandFileName() `void Util::FileMaster::setCommandFileName ( const std::string & commandFileName )`

Set the command file name.

Parameters

commandFileName	name of command file

Definition at line 112 of file FileMaster.cpp.
References commandFileName().

12.130.3.6 setInputPrefix() `void Util::FileMaster::setInputPrefix ( const std::string & inputPrefix )`

Set the input file prefix string.

Parameters

inputPrefix	input file prefix string

Definition at line 94 of file FileMaster.cpp.

12.130.3.7 setOutputPrefix() `void Util::FileMaster::setOutputPrefix ( const std::string & outputPrefix )`

Set the output file prefix string.

Parameters

outputPrefix	output file prefix string

Definition at line 100 of file FileMaster.cpp.

12.130.3.8 readParameters() `void Util::FileMaster::readParameters (`

```
std::istream & in )  [virtual]
```

Read parameter file.
Reads the inputPrefix and outputPrefix string variables.

Parameters

in	pararameter file input stream

Reimplemented from Util::ParamComposite.
Definition at line 118 of file FileMaster.cpp.

12.130.3.9 loadParameters()

```
void Util::FileMaster::loadParameters (
    Serializable::IArchive & ar )  [virtual]
```

Load internal state from file.

Parameters

ar	input/loading archive

Reimplemented from Util::ParamComposite.
Definition at line 130 of file FileMaster.cpp.
References Util::MpiLoader< IArchive >::load().

12.130.3.10 save()

```
void Util::FileMaster::save (
    Serializable::OArchive & ar )  [virtual]
```

Save internal state to file.

Parameters

ar	output/saving archive

Reimplemented from Util::ParamComposite.
Definition at line 145 of file FileMaster.cpp.

12.130.3.11 open() [1/2]

```
void Util::FileMaster::open (
    const std::string & name,
    std::ifstream & in,
    std::ios_base::openmode mode = std::ios_base::in ) const
```

Open an input file with a known path and open mode.
Adds error checking to C++ ifstream::open function.

Parameters

name	complete file path
in	ifstream object to associated with a file
mode	read mode

Definition at line 198 of file FileMaster.cpp.
References UTIL_THROW.

Referenced by openControlFile(), openInputFile(), openOutputFile(), openRestartIFile(), and openRestartOFile().

12.130.3.12 open() [2/2]

```
void Util::FileMaster::open (
    const std::string & name,
    std::ofstream & out,
    std::ios_base::openmode mode = std::ios_base::out ) const
```

Open an output file with a known path and open mode.
Add error checking to C++ ofstream::open function.

Parameters

name	complete file path
out	ofstream object to associated with a file
mode	write mode

Definition at line 213 of file FileMaster.cpp.
References UTIL_THROW.

12.130.3.13 openControlFile()

```
void Util::FileMaster::openControlFile (
    const std::string & name,
    std::ifstream & in ) const
```

Open an input parameter or command file.
The path to this file constructed by concatenating: [rootPrefix] + [directoryIdPrefix] + name + "." + ext
The directoryIdPrefix is included only if a directory id has not been set and the setCommonControl() function has not been called.

Parameters

name	base file name, without any prefix
in	ifstream object to open

Definition at line 227 of file FileMaster.cpp.
References open().
Referenced by commandFile(), and paramFile().

12.130.3.14 openRestartIFile()

```
void Util::FileMaster::openRestartIFile (
    const std::string & name,
    std::ifstream & in,
    std::ios_base::openmode mode = std::ios_base::in ) const
```

Open an input restart dump file for reading.
The path to this file constructed by concatenating: [rootPrefix] + [directoryIdPrefix] + name + "." + ext

Parameters

name	base file name, without any prefix or extension
in	ifstream object to open
mode	open mode

Definition at line 242 of file FileMaster.cpp.

References open().

12.130.3.15 openRestartOFile()

```
void Util::FileMaster::openRestartOFile (
    const std::string & name,
    std::ofstream & out,
    std::ios_base::openmode mode = std::ios_base::out ) const
```

Open an output restart file for writing.
The path to this file constructed by concatenating: [rootPrefix] + [directoryIdPrefix] + name

Parameters

name	base file name
out	ofstream object to open
mode	open mode

Definition at line 257 of file FileMaster.cpp.
References open().

12.130.3.16 openInputFile()

```
void Util::FileMaster::openInputFile (
    const std::string & filename,
    std::ifstream & in,
    std::ios_base::openmode mode = std::ios_base::in ) const
```

Open an input file.
The path to this file constructed by concatenating: [rootPrefix] + [directoryIdPrefix] + inputPrefix + filename.

Parameters

filename	file name, without any prefix
in	ifstream object to associated with a file
mode	bit mask that specifies opening mode

Definition at line 273 of file FileMaster.cpp.
References open().

12.130.3.17 openOutputFile()

```
void Util::FileMaster::openOutputFile (
    const std::string & filename,
    std::ofstream & out,
    std::ios_base::openmode mode = std::ios_base::out ) const
```

Open an output file.
The path to this file constructed by concatenating: [rootPrefix] + [directoryIdPrefix] + outputPrefix + filename.

Parameters

filename	file name, without any prefix
out	ofstream object to associated with a file
mode	bit mask that specifies opening mode

Definition at line 290 of file FileMaster.cpp.

References open().

12.130.3.18 isCommonControl() `bool Util::FileMaster::isCommonControl ( ) const`

Is set for common param and command files?
Definition at line 308 of file FileMaster.cpp.

12.130.3.19 paramFileName() `std::string Util::FileMaster::paramFileName ( ) const [inline]`

Return the param file name, if any.
Definition at line 466 of file FileMaster.h.
Referenced by setParamFileName().

12.130.3.20 commandFileName() `std::string Util::FileMaster::commandFileName ( ) const [inline]`

Return the command file name.
The base name of the command file is read from the parameter file by the readParameters() method.
Definition at line 472 of file FileMaster.h.
Referenced by setCommandFileName().

12.130.3.21 paramFile() `std::istream & Util::FileMaster::paramFile ( )`

Get a default parameter stream by reference.
If setDirectoryId() has not been called, of if setCommonControl() has been called, this method returns std::cin.
If setDirectoryId() has been called and setCommonControld() has not, this method returns a reference to a file "n/param".
This file is opened for reading the first time it is returned by this function.
Definition at line 155 of file FileMaster.cpp.
References openControlFile().

12.130.3.22 commandFile() `std::istream & Util::FileMaster::commandFile ( )`

Get the command input stream by reference.
If the commandFileName string is equal to the string literal "paramfile", this method returns the same stream as paramFile(). Otherwise, it returns a reference to a file whose name is given by the commandFileName string. If setDirectoryId(int) has not been called, the path to this file (absolute or relative to the working directory) is equal to the commandFileName string. If setDirectory() has been called with an integer argument n, the path to this file is obtained adding "n/" as a prefix to the commandFileName. In either case, if the commandFileName is not "paramfile", the required file is opened for reading the first time it is returned by this method.
Definition at line 178 of file FileMaster.cpp.
References openControlFile().
The documentation for this class was generated from the following files:

- FileMaster.h
- FileMaster.cpp

12.131 Util::FlagSet Class Reference

A set of boolean variables represented by characters.
`#include <FlagSet.h>`

Public Member Functions

- FlagSet ()

Default constructor.

- FlagSet (std::string allowed)

  Constructor.
- void setAllowed (std::string allowed)

  Set or reset the string of allowed flags.
- void setActualOrdered (std::string actual)

  Set the string of actual flag characters.
- bool isActive (char c) const

  Is the flag associated with character c active?
- const std::string & allowed () const

  Return the string of allowed characters.
- const std::string & actual () const

  Return the string of character for which flags are set.

12.131.1 Detailed Description

A set of boolean variables represented by characters.
A FlagSet has a string of allowed characters, each of which which represents a boolean variable (i.e., a flag), and a string of actual characters containing the subset of the allowed characters that should be set on (i.e., true).
Definition at line 28 of file FlagSet.h.

12.131.2 Constructor & Destructor Documentation

12.131.2.1 FlagSet() [1/2] `Util::FlagSet::FlagSet ( )`

Default constructor.
Definition at line 17 of file FlagSet.cpp.

12.131.2.2 FlagSet() [2/2] `Util::FlagSet::FlagSet ( std::string allowed )`

Constructor.
This function calls setAllowed(string) internally.

Parameters

allowed	string of all allowed characters.

Definition at line 23 of file FlagSet.cpp.
References allowed(), and setAllowed().

12.131.3 Member Function Documentation

12.131.3.1 setAllowed() `void Util::FlagSet::setAllowed ( std::string allowed )`

Set or reset the string of allowed flags.
This function sets isActive false for all flags and clears the actual string.

Parameters

allowed	string of all allowed characters

Definition at line 29 of file FlagSet.cpp.
References allowed().
Referenced by FlagSet().

12.131.3.2 setActualOrdered() `void Util::FlagSet::setActualOrdered ( std::string actual )`

Set the string of actual flag characters.
This function requires that the characters in the actual string appear in the same order as they do in the allowed string, but allows some allowed characters to be absent.
An Exception is thrown if actual contains a character that is not allowed, or if it is not in order.

Parameters

actual	string containing a subset of allowed characters

Definition at line 47 of file FlagSet.cpp.
References actual(), and UTIL_THROW.

12.131.3.3 isActive() `bool Util::FlagSet::isActive ( char c ) const [inline]`

Is the flag associated with character c active?

Parameters

c	character to be tested.

Definition at line 110 of file FlagSet.h.
References UTIL_THROW.

12.131.3.4 allowed() `const std::string & Util::FlagSet::allowed ( ) const [inline]`

Return the string of allowed characters.
Definition at line 123 of file FlagSet.h.
Referenced by FlagSet(), and setAllowed().

12.131.3.5 actual() `const std::string & Util::FlagSet::actual ( ) const [inline]`

Return the string of character for which flags are set.
Definition at line 130 of file FlagSet.h.
Referenced by setActualOrdered().
The documentation for this class was generated from the following files:

- FlagSet.h
- FlagSet.cpp

12.132 Util::FlexPtr< T > Class Template Reference

A pointer that may or may not own the object to which it points.

```
#include <FlexPtr.h>
```

Public Types

- typedef T element_type

  Type of object pointed to.

Public Member Functions

- FlexPtr ()

  Constructor.
- ~FlexPtr ()

  Destructor.
- void acquire (T *p)

  Copy a built-in pointer, and accept ownership.
- void copy (T *p)

  Copy a built-in pointer, without accepting ownership.
- T & operator* () const

  Dereference.
- T * operator-> () const

  Member access.
- T * get () const

  Return the built-in pointer.

12.132.1 Detailed Description

template<typename T>
class Util::FlexPtr< T >

A pointer that may or may not own the object to which it points.
FlexPtr overloads * and ->, and thus mimics a built-in pointer in most respects.
The acquire(T*) method copies a built-in pointer and accept ownership of the object to which it points, i.e., accepts responsibility for deleting the object, normally when the FlexPtr goes out of scope.
The copy(T*) method copies a built-in pointer without accepting ownership, i.e., without accepting responsibility for deleting the object to which it points.
Both acquire() and copy() destroy any object that is already owned by this FlexPtr before copying of a new pointer.
Definition at line 35 of file FlexPtr.h.

12.132.2 Member Typedef Documentation

12.132.2.1 element_type `template<typename T >`

```
typedef T Util::FlexPtr< T >::element_type
```

Type of object pointed to.
Definition at line 41 of file FlexPtr.h.

12.132.3 Constructor & Destructor Documentation

12.132.3.1 FlexPtr() template<typename T >

Util::FlexPtr< T >::FlexPtr ( ) [inline]

Constructor.

Definition at line 46 of file FlexPtr.h.

12.132.3.2 ∼FlexPtr() template<typename T >

Util::FlexPtr< T >::∼FlexPtr ( ) [inline]

Destructor.

Deletes any object that is owned by this FlexPtr.

Definition at line 56 of file FlexPtr.h.

12.132.4 Member Function Documentation

12.132.4.1 acquire() template<typename T >

```
void Util::FlexPtr< T >::acquire (
            T * p ) [inline]
```

Copy a built-in pointer, and accept ownership.

If this FlexPtr already owns an object, it will be deleted before acquiring a new pointer.

Throws an Exception if p is null.

Parameters

p	Built-in pointer to be acquired.

Definition at line 73 of file FlexPtr.h.

References UTIL_THROW.

12.132.4.2 copy() template<typename T >

```
void Util::FlexPtr< T >::copy (
            T * p ) [inline]
```

Copy a built-in pointer, without accepting ownership.

If this FlexPtr already owns an object, it will be deleted before copying a new pointer.

Parameters

p	Built-in pointer to be copied.

Definition at line 93 of file FlexPtr.h.

References UTIL_THROW.

12.132.4.3 operator∗() template<typename T >

T& Util::FlexPtr< T >::operator∗ ( ) const [inline]

Dereference.

Definition at line 108 of file FlexPtr.h.

12.132.4.4 operator->() template<typename T >

T∗ Util::FlexPtr< T >::operator-> ( ) const [inline]

Member access.
Definition at line 114 of file FlexPtr.h.

12.132.4.5 get() `template<typename T >`

`T* Util::FlexPtr< T >::get ( ) const [inline]`

Return the built-in pointer.
Definition at line 120 of file FlexPtr.h.
Referenced by Util::isNull().
The documentation for this class was generated from the following file:

- FlexPtr.h

12.133 Util::FMatrix< Data, M, N > Class Template Reference

Fixed Size Matrix.

```
#include <FMatrix.h>
```

Inheritance diagram for Util::FMatrix< Data, M, N >:

classUtil_1_1FMatrix-eps-converted-to.pdf

Public Member Functions

- FMatrix ()

  Default constructor.
- FMatrix (const FMatrix< Data, M, N > &other)

  Copy constructor.
- ∼FMatrix ()

  Destructor.
- FMatrix< Data, M, N > & operator= (const FMatrix< Data, M, N > &other)

  Assignment.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize an FMatrix to/from an Archive.

Additional Inherited Members

12.133.1 Detailed Description

template<typename Data, int M, int N>
class Util::FMatrix< Data, M, N >

Fixed Size Matrix.
The FMatrix class wraps a statically allocated 1D C array, but provides access to its elements via the A(i,j) Matrix syntax. Template parameters M and N are the number of rows and columns respectively, so that capacity1 = M and capacity2 = N.
Definition at line 28 of file FMatrix.h.

12.133.2 Constructor & Destructor Documentation

12.133.2.1 FMatrix() [1/2] template<typename Data , int M, int N>

Util::FMatrix< Data, M, N >::FMatrix

Default constructor.

Definition at line 77 of file FMatrix.h.

12.133.2.2 FMatrix() [2/2] template<typename Data , int M, int N>

```
Util::FMatrix< Data, M, N >::FMatrix (
            const FMatrix< Data, M, N > & other )
```

Copy constructor.

Definition at line 89 of file FMatrix.h.

12.133.2.3 ∼FMatrix() template<typename Data , int M, int N>

Util::FMatrix< Data, M, N >::∼FMatrix

Destructor.

Definition at line 104 of file FMatrix.h.

12.133.3 Member Function Documentation

12.133.3.1 operator=() template<typename Data , int M, int N>

```
FMatrix< Data, M, N > & Util::FMatrix< Data, M, N >::operator= (
            const FMatrix< Data, M, N > & other )
```

Assignment.

Definition at line 112 of file FMatrix.h.

12.133.3.2 serialize() template<class Data , int M, int N>

```
template<class Archive >
void Util::FMatrix< Data, M, N >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize an FMatrix to/from an Archive.

Parameters

ar	archive
version	archive version id

Definition at line 130 of file FMatrix.h.

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

- FMatrix.h

12.134 Util::Format Class Reference

Base class for output wrappers for formatted C++ ostream output.

`#include <Format.h>`

Static Public Member Functions

- static void initStatic ()
  Initialize or reset default width and precision values.
- static void setDefaultWidth (int width)
  Set the default output field width.
- static void setDefaultPrecision (int precision)
  Set the default output precision.
- static int defaultWidth ()
  Return the default output field width.
- static int defaultPrecision ()
  Return the default output precision.

12.134.1 Detailed Description

Base class for output wrappers for formatted C++ ostream output.
Public members are all getters and setters for static variables defaultWidth and defaultPrecision.
Definition at line 24 of file Format.h.

12.134.2 Member Function Documentation

12.134.2.1 initStatic() `void Util::Format::initStatic ( ) [static]`

Initialize or reset default width and precision values.
Definition at line 47 of file Format.cpp.
Referenced by Util::initStatic().

12.134.2.2 setDefaultWidth() `void Util::Format::setDefaultWidth ( int width ) [static]`

Set the default output field width.
Set Format::defaultWidth_.
Definition at line 20 of file Format.cpp.

12.134.2.3 setDefaultPrecision() `void Util::Format::setDefaultPrecision ( int precision ) [static]`

Set the default output precision.
Definition at line 26 of file Format.cpp.

12.134.2.4 defaultWidth() `int Util::Format::defaultWidth ( ) [static]`

Return the default output field width.
Definition at line 32 of file Format.cpp.

12.134.2.5 defaultPrecision() `int Util::Format::defaultPrecision ( ) [static]`

Return the default output precision.
Definition at line 38 of file Format.cpp.
The documentation for this class was generated from the following files:

- Format.h
- Format.cpp

12.135 Util::FPArray< Data, Capacity > Class Template Reference

Statically allocated pointer array.

`#include <FPArray.h>`

Public Member Functions

- FPArray ()

  Default constructor.
- FPArray (const FPArray< Data, Capacity > &other)

  Copy constructor.
- ∼FPArray ()

  Destructor.
- FPArray< Data, Capacity > & operator= (const FPArray< Data, Capacity > &other)

  Assignment, element by element.
- void append (Data &data)

  Append an element to the end of the array.
- void clear ()

  Set logical size to zero and nullify all elements.
- int capacity () const

  Return physical capacity of array.
- int size () const

  Return logical size of this array.
- void begin (PArrayIterator< Data > &iterator)

  Set an iterator to begin this container.
- void begin (ConstPArrayIterator< Data > &iterator) const

  Set a const iterator to begin this container.
- Data & operator[ ] (int i)

  Get an element by reference (mimic C-array subscripting).
- const Data & operator[ ] (int i) const

  Get an element by const reference (mimic C-array subscripting).

Protected Attributes

- Data ∗ ptrs_ [Capacity]

  Array of pointers to Data objects.
- int size_

  Logical size of array (number of elements used).

12.135.1 Detailed Description

template<typename Data, int Capacity>
class Util::FPArray< Data, Capacity >

Statically allocated pointer array.

A FPArray is a statically allocated array that actually holds pointers to Data objects, but for which the [] operator returns a reference to the associated object. It is implemented as a wrapper for a statically allocated C array of Data∗ pointers.

A FPArray is not responsible for destroying the associated Data objects.

The interface of an FPArray is identical to that of an FSArray. An FPArray has both a capacity that is set at compile time, which is the physical size of the underlying C array, and a size, which is the number of contiguous elements (indexed

from 0 to size-1) that contain valid pointers. The size can only be increased only by the append() method, which adds an element to the end of the array.
When compiled in debug mode, the operator [] checks that the index is less than the size and non-negative.
Definition at line 40 of file FPArray.h.

12.135.2 Constructor & Destructor Documentation

12.135.2.1 FPArray() [1/2] `template<typename Data , int Capacity>`

`Util::FPArray< Data, Capacity >::FPArray [inline]`

Default constructor.
Constructor.
Definition at line 139 of file FPArray.h.

12.135.2.2 FPArray() [2/2] `template<typename Data , int Capacity>`

```
Util::FPArray< Data, Capacity >::FPArray (
            const FPArray< Data, Capacity > & other )
```

Copy constructor.
Copies all pointers.

Parameters

other	the FPArray to be copied.

Definition at line 147 of file FPArray.h.
References Util::FPArray< Data, Capacity >::ptrs_, and Util::FPArray< Data, Capacity >::size_.

12.135.2.3 ∼FPArray() `template<typename Data , int Capacity>`

`Util::FPArray< Data, Capacity >::∼FPArray`

Destructor.
Definition at line 202 of file FPArray.h.

12.135.3 Member Function Documentation

12.135.3.1 operator=() `template<typename Data , int Capacity>`

```
FPArray< Data, Capacity > & Util::FPArray< Data, Capacity >::operator= (
            const FPArray< Data, Capacity > & other )
```

Assignment, element by element.

Parameters

other	the rhs FPArray

Definition at line 171 of file FPArray.h.
References Util::FPArray< Data, Capacity >::size_, and UTIL_THROW.

12.135.3.2 append() `template<typename Data , int Capacity>`

```
void Util::FPArray< Data, Capacity >::append (
            Data & data )  [inline]
```

Append an element to the end of the array.

Parameters

data	Data to add to end of array.

Definition at line 275 of file FPArray.h.
References UTIL_THROW.

12.135.3.3 clear() template<typename Data , int Capacity>

```
void Util::FPArray< Data, Capacity >::clear  [inline]
```

Set logical size to zero and nullify all elements.
Definition at line 288 of file FPArray.h.

12.135.3.4 capacity() template<typename Data , int Capacity>

```
int Util::FPArray< Data, Capacity >::capacity  [inline]
```

Return physical capacity of array.
Definition at line 209 of file FPArray.h.

12.135.3.5 size() template<typename Data , int Capacity>

```
int Util::FPArray< Data, Capacity >::size  [inline]
```

Return logical size of this array.
Definition at line 216 of file FPArray.h.

12.135.3.6 begin() [1/2] template<typename Data , int Capacity>

```
void Util::FPArray< Data, Capacity >::begin (
            PArrayIterator< Data > & iterator )  [inline]
```

Set an iterator to begin this container.

Parameters

iterator	PArrayIterator, initialized on output.

Definition at line 224 of file FPArray.h.
References Util::PArrayIterator< Data >::setCurrent(), and Util::PArrayIterator< Data >::setEnd().

12.135.3.7 begin() [2/2] template<typename Data , int Capacity>

```
void Util::FPArray< Data, Capacity >::begin (
            ConstPArrayIterator< Data > & iterator ) const  [inline]
```

Set a const iterator to begin this container.

Parameters

iterator	ConstPArrayIterator, initialized on output.

Definition at line 235 of file FPArray.h.
References Util::ConstPArrayIterator< Data >::setCurrent(), and Util::ConstPArrayIterator< Data >::setEnd().

12.135.3.8 operator[]() [1/2] template<typename Data , int Capacity>

```
Data & Util::FPArray< Data, Capacity >::operator[] (
            int i ) [inline]
```

Get an element by reference (mimic C-array subscripting).

Parameters

i	array index

Returns

reference to element i

Definition at line 248 of file FPArray.h.

12.135.3.9 operator[]() [2/2] template<typename Data , int Capacity>

```
const Data & Util::FPArray< Data, Capacity >::operator[] (
            int i ) const [inline]
```

Get an element by const reference (mimic C-array subscripting).

Parameters

i	array index

Returns

const reference to element i

Definition at line 262 of file FPArray.h.

12.135.4 Member Data Documentation

12.135.4.1 ptrs_ template<typename Data , int Capacity>

```
Data* Util::FPArray< Data, Capacity >::ptrs_[Capacity] [protected]
```

Array of pointers to Data objects.
Definition at line 126 of file FPArray.h.
Referenced by Util::FPArray< Data, Capacity >::FPArray().

12.135.4.2 size_ template<typename Data , int Capacity>

```
int Util::FPArray< Data, Capacity >::size_ [protected]
```

Logical size of array (number of elements used).
Definition at line 129 of file FPArray.h.
Referenced by Util::FPArray< Data, Capacity >::FPArray(), and Util::FPArray< Data, Capacity >::operator=().
The documentation for this class was generated from the following file:

- FPArray.h

12.136 Util::FSArray< Data, Capacity > Class Template Reference

A fixed capacity (static) contiguous array with a variable logical size.

```
#include <FSArray.h>
```

Public Member Functions

- FSArray ()

  Constructor.
- FSArray (const FSArray< Data, Capacity > &other)

  Copy constructor.
- FSArray< Data, Capacity > & operator= (const FSArray< Data, Capacity > &other)

  Assignment, element by element.
- virtual ~FSArray ()

  Destructor.
- int capacity () const

  Return physical capacity of array.
- int size () const

  Return logical size of this array (i.e., number of elements).
- void begin (ArrayIterator< Data > &iterator)

  Set an ArrayIterator to the beginning of this container.
- void begin (ConstArrayIterator< Data > &iterator) const

  Set a ConstArrayIterator to the beginning of this container.
- Data & operator[ ] (int i)

  Mimic C array subscripting.
- const Data & operator[ ] (int i) const

  Mimic C array subscripting.
- void append (const Data &data)

  Append data to the end of the array.
- void clear ()

  Set logical size to zero.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize to/from an archive.
- int packedSize ()

  Packed size of FSArray in a MemoryArchive, in bytes.

Protected Attributes

- Data data_ [Capacity]

  Array of Data elements.
- int size_

  Logical size of array (number of elements used).

12.136.1 Detailed Description

template<typename Data, int Capacity>
class Util::FSArray< Data, Capacity >

A fixed capacity (static) contiguous array with a variable logical size.
An FSArray < Data, Capacity > is a wrapper for a statically allocated C array containing Capacity objects of type Data. An FSArray has both a Capacity that is set at compile time, which is the physical size of the underlying C array, and a logical size, which is the number of contiguous elements (from 0 to one less than its size) that contain valid data. The size is initialized to zero, and can only be increased only by the append() method, which adds a new element to the end of the array.

When compiled in debug mode (i.e., when NDEBUG is defined) the subcript operator [] checks that the index is less than the logical size, and not merely less than the capacity.
Definition at line 37 of file FSArray.h.

12.136.2 Constructor & Destructor Documentation

12.136.2.1 FSArray() [1/2] template<class Data , int Capacity>

`Util::FSArray< Data, Capacity >::FSArray  [inline]`

Constructor.
Definition at line 148 of file FSArray.h.

12.136.2.2 FSArray() [2/2] template<class Data , int Capacity>

```
Util::FSArray< Data, Capacity >::FSArray (
            const FSArray< Data, Capacity > & other )
```

Copy constructor.

Parameters

other	the FSArray to be copied.

Definition at line 158 of file FSArray.h.

12.136.2.3 ∼FSArray() template<class Data , int Capacity>

`Util::FSArray< Data, Capacity >::∼FSArray  [virtual]`

Destructor.
Definition at line 193 of file FSArray.h.

12.136.3 Member Function Documentation

12.136.3.1 operator=() template<class Data , int Capacity>

```
FSArray< Data, Capacity > & Util::FSArray< Data, Capacity >::operator= (
            const FSArray< Data, Capacity > & other )
```

Assignment, element by element.
Capacity of LHS FSArray must be >= size of RHS FSArray.

Parameters

other	the RHS FSArray

Definition at line 175 of file FSArray.h.

12.136.3.2 capacity() `template<class Data , int Capacity>`

`int Util::FSArray< Data, Capacity >::capacity`

Return physical capacity of array.
Definition at line 200 of file FSArray.h.

12.136.3.3 size() `template<class Data , int Capacity>`

`int Util::FSArray< Data, Capacity >::size`

Return logical size of this array (i.e., number of elements).
Definition at line 207 of file FSArray.h.
Referenced by Pscf::UnitCellBase< 3 >::setParameters().

12.136.3.4 begin() [1/2] `template<class Data , int Capacity>`

```
void Util::FSArray< Data, Capacity >::begin (
            ArrayIterator< Data > & iterator )
```

Set an ArrayIterator to the beginning of this container.

Parameters

iterator	ArrayIterator, initialized on output.

Definition at line 216 of file FSArray.h.

12.136.3.5 begin() [2/2] `template<class Data , int Capacity>`

```
void Util::FSArray< Data, Capacity >::begin (
            ConstArrayIterator< Data > & iterator ) const
```

Set a ConstArrayIterator to the beginning of this container.

Parameters

iterator	ConstArrayIterator, initialized on output.

Definition at line 226 of file FSArray.h.

12.136.3.6 operator[]() [1/2] `template<class Data , int Capacity>`

```
Data & Util::FSArray< Data, Capacity >::operator[] (
            int i )
```

Mimic C array subscripting.

Parameters

i	array index

Returns

reference to element i

Definition at line 236 of file FSArray.h.

12.136.3.7 operator[]() [2/2] template<class Data , int Capacity>

```
const Data & Util::FSArray< Data, Capacity >::operator[] (
             int i ) const
```

Mimic C array subscripting.

Parameters

i	array index

Returns

const reference to element i

Definition at line 247 of file FSArray.h.

12.136.3.8 append() template<class Data , int Capacity>

```
void Util::FSArray< Data, Capacity >::append (
             const Data & data ) [inline]
```

Append data to the end of the array.

Parameters

data	Data to add to end of array.

Definition at line 258 of file FSArray.h.
Referenced by Pscf::UnitCellBase< 3 >::parameters().

12.136.3.9 clear() template<class Data , int Capacity>

```
void Util::FSArray< Data, Capacity >::clear [inline]
```

Set logical size to zero.
Definition at line 271 of file FSArray.h.

12.136.3.10 serialize() template<class Data , int Capacity>

```
template<class Archive >
void Util::FSArray< Data, Capacity >::serialize (
             Archive & ar,
             const unsigned int version ) [inline]
```

Serialize to/from an archive.

Parameters

ar	archive
version	archive version id

Definition at line 279 of file FSArray.h.

12.136.3.11 packedSize() `template<typename Data , int Capacity>`

`int Util::FSArray< Data, Capacity >::packedSize [inline]`

Packed size of FSArray in a MemoryArchive, in bytes.

Definition at line 295 of file FSArray.h.

12.136.4 Member Data Documentation

12.136.4.1 data_ `template<typename Data , int Capacity>`

`Data Util::FSArray< Data, Capacity >::data_[Capacity] [protected]`

Array of Data elements.

Definition at line 137 of file FSArray.h.

Referenced by Util::FSArray< double, 6 >::FSArray().

12.136.4.2 size_ `template<typename Data , int Capacity>`

`int Util::FSArray< Data, Capacity >::size_ [protected]`

Logical size of array (number of elements used).

Definition at line 140 of file FSArray.h.

Referenced by Util::FSArray< double, 6 >::FSArray(), and Util::FSArray< double, 6 >::operator=().

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

- FSArray.h

12.137 Util::GArray< Data > Class Template Reference

An automatically growable array, analogous to a std::vector.

`#include <GArray.h>`

Public Member Functions

- GArray ()

    Constructor.
- GArray (const GArray< Data > &other)

    Copy constructor, copy pointers.
- GArray< Data > & operator= (const GArray< Data > &other)

    Assignment, element by element.
- virtual ~GArray ()

    Destructor.
- void reserve (int capacity)

    Reserve memory for specified number of elements.
- void deallocate ()

    Deallocate (delete) underlying array of pointers.
- void clear ()

    Reset to empty state.
- template< class Archive >
  void serialize (Archive &ar, const unsigned int version)

    Serialize a GArray to/from an Archive.

- void begin (ArrayIterator< Data > &iterator)

    Set an ArrayIterator to the beginning of this Array.
- void begin (ConstArrayIterator< Data > &iterator) const

    Set a ConstArrayIterator to the beginning of this Array.
- void append (const Data &data)

    Append an element to the end of the sequence.
- void resize (int n)

    Resizes array so that it contains n elements.
- Data & operator[ ] (int i)

    Mimic C array subscripting.
- const Data & operator[ ] (int i) const

    Mimic C array subscripting.
- int capacity () const

    Return physical capacity of array.
- int size () const

    Return logical size of this array (i.e., current number of elements).
- bool isAllocated () const

    Is this array allocated?

12.137.1 Detailed Description

template<typename Data>
class Util::GArray< Data >

An automatically growable array, analogous to a std::vector.
An GArray is an array that grows as needed as objects are appended. It has a logical size that grows when objects are appended, which is always less than or equal to the current physical capacity. If an object is added when the size is already equal to the capacity, the array will be resized and copied to a new location in memory. The elements of a GArray are deleted when the GArray is destroyed or deallocated.
Definition at line 33 of file GArray.h.

12.137.2 Constructor & Destructor Documentation

12.137.2.1 GArray() `[1/2]` `template<typename Data >`

`Util::GArray< Data >::GArray`

Constructor.
Definition at line 191 of file GArray.h.

12.137.2.2 GArray() `[2/2]` `template<typename Data >`

```
Util::GArray< Data >::GArray (
            const GArray< Data > & other )
```

Copy constructor, copy pointers.
Allocates new C-array and copies pointers to Data objects.

Parameters

other	the GArray to be copied.

Definition at line 203 of file GArray.h.

12.137.2.3 ∼GArray() template<typename Data >

`Util::GArray< Data >::~GArray [virtual]`

Destructor.

Deletes array of pointers, if allocated previously. Does not delete the associated Data objects.

Definition at line 226 of file GArray.h.

12.137.3 Member Function Documentation

12.137.3.1 operator=() template<typename Data >

```
GArray< Data > & Util::GArray< Data >::operator= (
            const GArray< Data > & other )
```

Assignment, element by element.

Parameters

other	the rhs GArray

Definition at line 239 of file GArray.h.

12.137.3.2 reserve() template<typename Data >

```
void Util::GArray< Data >::reserve (
            int capacity )
```

Reserve memory for specified number of elements.

Resizes and copies array if requested capacity is less than the current capacity. Does nothing if requested capacity is greater than current capacity.

Parameters

capacity	number of elements for which to reserve space.

Definition at line 255 of file GArray.h.

Referenced by Util::Polynomial< double >::operator∗=(), Util::Polynomial< double >::operator=(), and Util::Polynomial< double >::Polynomial().

12.137.3.3 deallocate() template<typename Data >

`void Util::GArray< Data >::deallocate`

Deallocate (delete) underlying array of pointers.

Sets capacity and size to zero.

Definition at line 286 of file GArray.h.

Referenced by Util::Binomial::clear().

12.137.3.4 clear() template<typename Data >

`void Util::GArray< Data >::clear`

Reset to empty state.

Sets size to zero, but leaves capacity unchanged. Does not call destructor for deleted elements.
Definition at line 299 of file GArray.h.
Referenced by Util::Polynomial< double >::setToZero().

12.137.3.5 serialize() template<class Data >

```
template<class Archive >
void Util::GArray< Data >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize a GArray to/from an Archive.

Parameters

ar	archive
version	archive version id

Definition at line 380 of file GArray.h.

12.137.3.6 begin() [1/2] template<class Data >

```
void Util::GArray< Data >::begin (
            ArrayIterator< Data > & iterator )  [inline]
```

Set an ArrayIterator to the beginning of this Array.

Parameters

iterator	ArrayIterator, initialized on output.

Definition at line 405 of file GArray.h.

12.137.3.7 begin() [2/2] template<class Data >

```
void Util::GArray< Data >::begin (
            ConstArrayIterator< Data > & iterator ) const  [inline]
```

Set a ConstArrayIterator to the beginning of this Array.

Parameters

iterator	ConstArrayIterator, initialized on output.

Definition at line 416 of file GArray.h.

12.137.3.8 append() template<typename Data >

```
void Util::GArray< Data >::append (
            const Data & data )
```

Append an element to the end of the sequence.
Resizes array if space is inadequate.

Parameters

data	Data object to be appended

Definition at line 306 of file GArray.h.

Referenced by Util::Polynomial< double >::operator*=(), Util::Polynomial< double >::operator+=(), Util::Polynomial< double >::operator-=(), Util::Polynomial< double >::operator=(), and Util::Polynomial< double >::Polynomial().

12.137.3.9 resize() `template<typename Data >`

```
void Util::GArray< Data >::resize (
            int n )
```

Resizes array so that it contains n elements.

This function changes the size of the array to n, and changes the capacity iff necesary to accomodate the change in size. Upon return, size is set to n. In what follows, "size" and "capacity" refer to values on entry:

If n < size, size is reset, but no destructors are called If n > size, all added elements are value initialized If n > capacity, new memory is allocated and the array is moved

Parameters

n	desired number of elements

Definition at line 339 of file GArray.h.

Referenced by Util::Binomial::setup().

12.137.3.10 operator[]() [1/2] `template<class Data >`

```
Data & Util::GArray< Data >::operator[] (
            int i ) [inline]
```

Mimic C array subscripting.

Parameters

i	array index

Returns

reference to element i

Definition at line 426 of file GArray.h.

12.137.3.11 operator[]() [2/2] `template<class Data >`

```
const Data & Util::GArray< Data >::operator[] (
            int i ) const [inline]
```

Mimic C array subscripting.

Parameters

i	array index

Returns

const reference to element i

Definition at line 437 of file GArray.h.

12.137.3.12 capacity() `template<class Data >`

`int Util::GArray< Data >::capacity [inline]`

Return physical capacity of array.
Definition at line 448 of file GArray.h.
Referenced by Util::Binomial::clear().

12.137.3.13 size() `template<class Data >`

`int Util::GArray< Data >::size [inline]`

Return logical size of this array (i.e., current number of elements).
Definition at line 455 of file GArray.h.
Referenced by Util::Polynomial< double >::degree(), and Util::Polynomial< double >::operator∗=().

12.137.3.14 isAllocated() `template<class Data >`

`bool Util::GArray< Data >::isAllocated [inline]`

Is this array allocated?
Definition at line 462 of file GArray.h.
Referenced by Util::GArray< Rational >::GArray().
The documentation for this class was generated from the following file:

- GArray.h

12.138 Util::GPArray< Data > Class Template Reference

An automatically growable PArray.
`#include <GPArray.h>`
Inheritance diagram for Util::GPArray< Data >:

classUtil_1_1GPArray-eps-converted-to.pdf

Public Member Functions

- GPArray ()

  Constructor.
- GPArray (const GPArray< Data > &other)

  Copy constructor, copy pointers.
- GPArray< Data > & operator= (const GPArray< Data > &other)

  Assignment, element by element.
- virtual ∼GPArray ()

  Destructor.
- void append (Data &data)

  Append an element to the end of the sequence.
- void reserve (int capacity)

  Reserve memory for specified number of elements.
- void deallocate ()

  Deallocate (delete) underlying array of pointers.

- void clear ()

  Reset to empty state.
- bool isAllocated () const

  Is this GPArray allocated?

Additional Inherited Members

12.138.1 Detailed Description

template< typename Data >
class Util::GPArray< Data >

An automatically growable PArray.
An GPArray is a PArray that grows as needed as objects are appended. Like any PArray, it holds pointers to objects, rather than objects. The associated objects are not destroyed when a PArray is deallocated or destroyed.
Definition at line 28 of file GPArray.h.

12.138.2 Constructor & Destructor Documentation

12.138.2.1 GPArray() [1/2] `template<typename Data >`

`Util::GPArray< Data >::GPArray [inline]`
Constructor.
Definition at line 113 of file GPArray.h.

12.138.2.2 GPArray() [2/2] `template<typename Data >`

```
Util::GPArray< Data >::GPArray (
            const GPArray< Data > & other )
```
Copy constructor, copy pointers.
Allocates new Data∗ array and copies pointers to Data objects.

Parameters

other	the GPArray to be copied.

Allocates a new Data∗ array and copies all pointer values.

Parameters

other	the GPArray to be copied.

Definition at line 125 of file GPArray.h.
References Util::PArray< Data >::capacity_, Util::PArray< Data >::ptrs_, and Util::PArray< Data >::size_.

12.138.2.3 ∼GPArray() `template<typename Data >`

`Util::GPArray< Data >::~GPArray [virtual]`
Destructor.
Deletes array of pointers, if allocated previously. Does not delete the associated Data objects.
Definition at line 176 of file GPArray.h.
References Util::Memory::deallocate().

12.138.3 Member Function Documentation

12.138.3.1 operator=() template<typename Data >

```
GPArray< Data > & Util::GPArray< Data >::operator= (
            const GPArray< Data > & other )
```

Assignment, element by element.

Preconditions:

- Both this and other GPArrays must be allocated.
- Capacity of this GPArray must be >= size of RHS GPArray.

Parameters

other	the rhs GPArray

Definition at line 160 of file GPArray.h.
References Util::PArray< Data >::size_.

12.138.3.2 append() template<typename Data >

```
void Util::GPArray< Data >::append (
            Data & data )
```

Append an element to the end of the sequence.

Resizes array if space is inadequate.

Parameters

data	Data object to be appended

Definition at line 235 of file GPArray.h.

12.138.3.3 reserve() template<typename Data >

```
void Util::GPArray< Data >::reserve (
            int capacity )
```

Reserve memory for specified number of elements.

Resizes and copies array if requested capacity is less than the current capacity. Does nothing if requested capacity is greater than current capacity.

Parameters

capacity	number of elements for which to reserve space.

Definition at line 189 of file GPArray.h.
References UTIL_THROW.

12.138.3.4 deallocate() template<typename Data >

```
void Util::GPArray< Data >::deallocate
```

Deallocate (delete) underlying array of pointers.

Definition at line 221 of file GPArray.h.

12.138.3.5 clear()

`template<typename Data >`
`void Util::GPArray< Data >::clear [inline]`

Reset to empty state.

Definition at line 268 of file GPArray.h.

12.138.3.6 isAllocated()

`template<class Data >`
`bool Util::GPArray< Data >::isAllocated [inline]`

Is this GPArray allocated?

Definition at line 275 of file GPArray.h.

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

- GPArray.h

12.139 Util::Grid Class Reference

A grid of points indexed by integer coordinates.

`#include <Grid.h>`

Public Member Functions

- Grid ()

  Default constructor.
- Grid (const IntVector &dimensions)

  Constructor.
- void setDimensions (const IntVector &dimensions)

  Set the grid dimensions in all directions.
- IntVector dimensions () const

  Get an IntVector of the grid dimensions.
- int dimension (int i) const

  Get grid dimension along Cartesian direction i.
- int size () const

  Get total number of grid points.
- IntVector position (int rank) const

  Get the position IntVector of a grid point with a specified rank.
- int rank (const IntVector &position) const

  Get the rank of a grid point with specified position.
- bool isInGrid (int coordinate, int i) const

  Is this coordinate in range?
- bool isInGrid (IntVector &position) const

  Is this IntVector grid position within the grid?
- int shift (int &coordinate, int i) const

  Shift a periodic coordinate into range.
- IntVector shift (IntVector &position) const

  Shift a periodic position into primary grid.

12.139.1 Detailed Description

A grid of points indexed by integer coordinates.

The coordinates of a point on a grid form an IntVector, referred to here as a grid position. Each element of a grid position must lie in the range 0 <= position[i] < dimension(i), where i indexes a Cartesian axis, and dimension(i) is the dimension of the grid along axis i.

Each grid position is also assigned a non-negative integer rank.

Grid position ranks are ordered sequentially like elements in a multi-dimensional C array, with the last coordinate being the most rapidly varying.

Definition at line 33 of file Grid.h.

12.139.2 Constructor & Destructor Documentation

12.139.2.1 Grid() [1/2] `Util::Grid::Grid ( )`

Default constructor.

Definition at line 15 of file Grid.cpp.

References dimensions(), and setDimensions().

12.139.2.2 Grid() [2/2] `Util::Grid::Grid ( const IntVector & dimensions )`

Constructor.

Parameters

dimensions	IntVector of grid dimensions

Definition at line 24 of file Grid.cpp.

References dimensions(), and setDimensions().

12.139.3 Member Function Documentation

12.139.3.1 setDimensions() `void Util::Grid::setDimensions ( const IntVector & dimensions )`

Set the grid dimensions in all directions.

Parameters

dimensions	IntVector of grid dimensions.

Definition at line 32 of file Grid.cpp.

References Util::Dimension, dimensions(), and UTIL_THROW.

Referenced by Grid().

12.139.3.2 dimensions() `IntVector Util::Grid::dimensions ( ) const [inline]`

Get an IntVector of the grid dimensions.

Definition at line 156 of file Grid.h.

Referenced by Grid(), and setDimensions().

12.139.3.3 dimension() `int Util::Grid::dimension ( int i ) const [inline]`

Get grid dimension along Cartesian direction i.

Parameters

i	index of Cartesian direction 0 <=i < Util::Dimension

Definition at line 159 of file Grid.h.
References Util::Dimension.

12.139.3.4 size() `int Util::Grid::size ( ) const [inline]`

Get total number of grid points.
Definition at line 166 of file Grid.h.

12.139.3.5 position() `IntVector Util::Grid::position ( int rank ) const`

Get the position IntVector of a grid point with a specified rank.

Parameters

rank	integer rank of a grid point.

Returns

IntVector containing coordinates of specified point.

Definition at line 64 of file Grid.cpp.
References Util::Dimension.
Referenced by isInGrid(), rank(), and shift().

12.139.3.6 rank() `int Util::Grid::rank ( const IntVector & position ) const`

Get the rank of a grid point with specified position.

Parameters

position	integer position of a grid point

Returns

integer rank of specified grid point

Definition at line 49 of file Grid.cpp.
References Util::Dimension, and position().

12.139.3.7 isInGrid() [1/2] `bool Util::Grid::isInGrid (`

```
int coordinate,
int i ) const
```

Is this coordinate in range?

Parameters

coordinate	coordinate value for direction i
i	index for Cartesian direction

Returns

true iff 0 <= coordinate < dimension(i).

Definition at line 78 of file Grid.cpp.

12.139.3.8 isInGrid() [2/2]

```
bool Util::Grid::isInGrid (
IntVector & position ) const
```

Is this IntVector grid position within the grid?

Returns true iff 0 <= coordinate[i] < dimension(i) for all i.

Parameters

position	grid point position

Returns

true iff 0 <= coordinate[i] < dimension(i) for all i.

Definition at line 88 of file Grid.cpp.

References Util::Dimension, and position().

12.139.3.9 shift() [1/2]

```
int Util::Grid::shift (
int & coordinate,
int i ) const
```

Shift a periodic coordinate into range.

Upon return, the coordinate will be shifted to lie within the range 0 <= coordinate < dimension(i) by subtracting an integer multiple of dimension(i), giving coordinate - shift∗dimension(i). The return value is the required integer 'shift'.

Parameters

coordinate	coordinate in Cartesian direction i.
i	index of Cartesian direction, i >= 0.

Returns

multiple of dimension(i) subtracted from input value.

Definition at line 100 of file Grid.cpp.

Referenced by shift().

12.139.3.10 shift() [2/2] IntVector Util::Grid::shift (IntVector & position) const

Shift a periodic position into primary grid.

Upon return, each element of the parameter position is shifted to lie within the range 0 <= position[i] < dimension(i) by adding or subtracting an integer multiple of dimension(i). The IntVector of shift values is returned.

Parameters

position	IntVector position within a grid.

Returns

IntVector of integer shifts.

Definition at line 112 of file Grid.cpp.

References Util::Dimension, position(), and shift().

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

- Grid.h
- Grid.cpp

12.140 Util::GridArray< Data > Class Template Reference

Multi-dimensional array with the dimensionality of space.

`#include <GridArray.h>`

Public Member Functions

- GridArray ()

  Constructor.
- GridArray (const GridArray< Data > &other)

  Copy constructor.
- ∼GridArray ()

  Destructor.
- GridArray< Data > & operator= (const GridArray< Data > &other)

  Assignment.
- void allocate (const IntVector &dimensions)

  Allocate memory for a matrix.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize a GridArray to/from an Archive.
- bool isAllocated () const

  Return true if the GridArray has been allocated, false otherwise.
- const IntVector & dimensions ()

  Get all dimensions of array as an IntVector.
- int dimension (int i) const

  Get number of grid points along direction i.
- int size () const

  Get total number of grid points.
- IntVector position (int rank) const

  Get the position IntVector of a grid point with a specified rank.

- int rank (const IntVector &position) const

  Get the rank of a grid point with specified position.
- bool isInGrid (int coordinate, int i) const

  Is this 1D coordinate in range?
- bool isInGrid (IntVector &position) const

  Is this position within the grid?
- int shift (int &coordinate, int i) const

  Shift a periodic 1D coordinate into primary range.
- IntVector shift (IntVector &position) const

  Shift a periodic position into primary grid.
- const Data & operator[ ] (int rank) const

  Return element by const reference, indexed by 1D rank.
- Data & operator[ ] (int rank)

  Return element by reference, indexed by 1D rank.
- const Data & operator() (const IntVector &position) const

  Return element by const reference, indexed by IntVector position.
- Data & operator() (const IntVector &position)

  Return element by reference, indexed by IntVector position.

12.140.1 Detailed Description

template<typename Data>
class Util::GridArray< Data >

Multi-dimensional array with the dimensionality of space.
The memory for a GridArray is stored in a single one-dimensional C array. The subscript [] operator is overloaded to return an element indexed by a one-dimensional rank, and the () operator is overloaded to return an element indexed by an IntVector of grid coordinates.
Definition at line 28 of file GridArray.h.

12.140.2 Constructor & Destructor Documentation

12.140.2.1 GridArray() [1/2] `template<typename Data >`

`Util::GridArray< Data >::GridArray [inline]`
Constructor.
Constructor (protected).
Definition at line 217 of file GridArray.h.

12.140.2.2 GridArray() [2/2] `template<typename Data >`

```
Util::GridArray< Data >::GridArray (
            const GridArray< Data > & other )
```
Copy constructor.
Definition at line 242 of file GridArray.h.
References Util::GridArray< Data >::allocate(), Util::GridArray< Data >::isAllocated(), and UTIL_THROW.

12.140.2.3 ∼GridArray() `template<typename Data >`

`Util::GridArray< Data >::~GridArray`

Destructor.

Delete dynamically allocated C array, if allocated.

Definition at line 230 of file GridArray.h.

12.140.3 Member Function Documentation

12.140.3.1 operator=() `template<typename Data >`

```
GridArray< Data > & Util::GridArray< Data >::operator= (
            const GridArray< Data > & other )
```

Assignment.

Definition at line 272 of file GridArray.h.

References UTIL_THROW.

12.140.3.2 allocate() `template<typename Data >`

```
void Util::GridArray< Data >::allocate (
            const IntVector & dimensions )
```

Allocate memory for a matrix.

Parameters

dimensions	IntVector containing dimensions

Definition at line 312 of file GridArray.h.

References Util::Dimension, and UTIL_THROW.

Referenced by Util::GridArray< Data >::GridArray().

12.140.3.3 serialize() `template<class Data >`

```
template<class Archive >
void Util::GridArray< Data >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize a GridArray to/from an Archive.

Parameters

ar	archive
version	archive version id

Definition at line 336 of file GridArray.h.

12.140.3.4 isAllocated() `template<class Data >`

`bool Util::GridArray< Data >::isAllocated [inline]`

Return true if the GridArray has been allocated, false otherwise.

Definition at line 517 of file GridArray.h.

Referenced by Util::GridArray< Data >::GridArray().

12.140.3.5 dimensions()

```
template<typename Data >
const IntVector & Util::GridArray< Data >::dimensions [inline]
```

Get all dimensions of array as an IntVector.

Returns

IntVector containing the number of elements in each direction.

Definition at line 357 of file GridArray.h.

12.140.3.6 dimension()

```
template<class Data >
int Util::GridArray< Data >::dimension (
            int i ) const [inline]
```

Get number of grid points along direction i.

Parameters

i	index of Cartesian direction 0 <=i < 3.

Definition at line 364 of file GridArray.h.

12.140.3.7 size()

```
template<class Data >
int Util::GridArray< Data >::size [inline]
```

Get total number of grid points.
Definition at line 371 of file GridArray.h.

12.140.3.8 position()

```
template<typename Data >
IntVector Util::GridArray< Data >::position (
            int rank ) const
```

Get the position IntVector of a grid point with a specified rank.

Parameters

rank	integer rank of a grid point.

Returns

IntVector containing coordinates of specified point.

Definition at line 404 of file GridArray.h.
References Util::Dimension.

12.140.3.9 rank()

```
template<typename Data >
int Util::GridArray< Data >::rank (
            const IntVector & position ) const [inline]
```

Get the rank of a grid point with specified position.

Parameters

position	integer position of a grid point

Returns

integer rank of specified grid point

Definition at line 394 of file GridArray.h.

12.140.3.10 isInGrid() [1/2] template<typename Data >

```
bool Util::GridArray< Data >::isInGrid (
             int coordinate,
             int i ) const
```

Is this 1D coordinate in range?
Returns true iff 0 <= coordinate < dimension(i).

Parameters

coordinate	coordinate value for direction i
i	index for Cartesian direction

Definition at line 422 of file GridArray.h.

12.140.3.11 isInGrid() [2/2] template<typename Data >

```
bool Util::GridArray< Data >::isInGrid (
             IntVector & position ) const
```

Is this position within the grid?
Returns true iff 0 <= coordinate[i] < dimension(i) for all i.

Parameters

position	grid point position

Definition at line 436 of file GridArray.h.
References Util::Dimension.

12.140.3.12 shift() [1/2] template<typename Data >

```
int Util::GridArray< Data >::shift (
             int & coordinate,
             int i ) const
```

Shift a periodic 1D coordinate into primary range.
Upon return, the coordinate will be shifted to lie within the range 0 <= coordinate < dimension(i) by subtracting an integer multiple of dimension(i), giving coordinate - shift∗dimension(i). The return value is the required integer 'shift'.

Parameters

coordinate	coordinate in Cartesian direction i.
i	index of Cartesian direction, i >= 0.

Returns

multiple of dimension(i) subtracted from input value.

Definition at line 452 of file GridArray.h.

12.140.3.13 shift() [2/2]

```
template<typename Data >
IntVector Util::GridArray< Data >::shift (
            IntVector & position ) const
```

Shift a periodic position into primary grid.

Upon return, each element of the parameter position is shifted to lie within the range 0 <= position[i] < dimension(i) by adding or subtracting an integer multiple of dimension(i). The IntVector of shift values is returned.

Parameters

position	IntVector position within a grid.

Returns

IntVector of integer shifts.

Definition at line 468 of file GridArray.h.
References Util::Dimension.

12.140.3.14 operator[]() [1/2]

```
template<typename Data >
const Data & Util::GridArray< Data >::operator[ ] (
            int rank ) const [inline]
```

Return element by const reference, indexed by 1D rank.

Parameters

rank	1D array index of element

Definition at line 481 of file GridArray.h.

12.140.3.15 operator[]() [2/2]

```
template<typename Data >
Data & Util::GridArray< Data >::operator[ ] (
            int rank ) [inline]
```

Return element by reference, indexed by 1D rank.

Parameters

rank	1D rank of element

Definition at line 488 of file GridArray.h.

12.140.3.16 operator()() [1/2]

```
template<typename Data >
const Data & Util::GridArray< Data >::operator() (
            const IntVector & position ) const [inline]
```

Return element by const reference, indexed by IntVector position.

Parameters

position	IntVector of coordinates.

Definition at line 496 of file GridArray.h.

12.140.3.17 operator()() [2/2] template<typename Data >

Data & Util::GridArray< Data >::operator() (
const IntVector & *position*) [inline]

Return element by reference, indexed by IntVector position.

Parameters

position	IntVector of coordinates.

Definition at line 503 of file GridArray.h.

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

- GridArray.h

12.141 Util::GStack< Data > Class Template Reference

An automatically growable Stack.

`#include <GStack.h>`

Public Member Functions

- GStack ()

  Constructor.
- GStack (const GStack< Data > &other)

  Copy constructor, copy pointers.
- ∼GStack ()

  Destructor.
- GStack< Data > & operator= (const GStack< Data > &other)

  Assignment, element by element.
- void reserve (int capacity)

  Reserve memory for specified number of elements.
- void deallocate ()

  Deallocate (delete) underlying array of pointers.
- void clear ()

  Reset to empty state.
- void push (Data &data)

  Push an element onto the stack.
- Data & pop ()

  Pop an element off the stack.
- Data & peek ()

  Return a reference to the top element (don't pop).
- const Data & peek () const

  Return a const ref to the top element (don't pop).
- int capacity () const

  Return allocated size.
- int size () const

  Return logical size.

- bool isAllocated () const

  Is this GStack allocated?
- bool isValid () const

  Is this GStack in a valid internal state?

12.141.1 Detailed Description

template<typename Data>
class Util::GStack< Data >

An automatically growable Stack.
A GStack is stack that is implemented as a growable pointer array. Like any pointer array it holds pointers to objects, rather than objects, and associated objects are not destroyed when the container is deallocated or destroyed.
Definition at line 28 of file GStack.h.

12.141.2 Constructor & Destructor Documentation

12.141.2.1 GStack() [1/2] `template<typename Data >`

`Util::GStack< Data >::GStack [inline]`
Constructor.
Definition at line 157 of file GStack.h.

12.141.2.2 GStack() [2/2] `template<typename Data >`

```
Util::GStack< Data >::GStack (
        const GStack< Data > & other )
```
Copy constructor, copy pointers.
Allocates new Data* array and copies pointers to Data objects.

Parameters

other	the GStack to be copied.

Allocates a new Data* array and copies all pointer values.

Parameters

other	the GStack to be copied.

Definition at line 171 of file GStack.h.

12.141.2.3 ∼GStack() `template<typename Data >`

`Util::GStack< Data >::∼GStack`
Destructor.
Deletes array of pointers, if allocated previously. Does not delete the associated Data objects.
Definition at line 208 of file GStack.h.
References Util::Memory::deallocate().

12.141.3 Member Function Documentation

12.141.3.1 operator=() `template<typename Data >`

```
GStack< Data > & Util::GStack< Data >::operator= (
            const GStack< Data > & other )
```

Assignment, element by element.
Preconditions:

- Both this and other GStacks must be allocated.
- Capacity of this GStack must be >= size of RHS GStack.

Parameters

other	the rhs GStack

Definition at line 221 of file GStack.h.

12.141.3.2 reserve() `template<typename Data >`

```
void Util::GStack< Data >::reserve (
            int capacity )
```

Reserve memory for specified number of elements.
Resizes and copies array if requested capacity is less than the current capacity. Does nothing if requested capacity is greater than current capacity.

Parameters

capacity	number of elements for which to reserve space.

Definition at line 237 of file GStack.h.
References UTIL_THROW.

12.141.3.3 deallocate() `template<typename Data >`

```
void Util::GStack< Data >::deallocate
```

Deallocate (delete) underlying array of pointers.
Definition at line 277 of file GStack.h.

12.141.3.4 clear() `template<typename Data >`

```
void Util::GStack< Data >::clear  [inline]
```

Reset to empty state.
Definition at line 291 of file GStack.h.

12.141.3.5 push() `template<typename Data >`

```
void Util::GStack< Data >::push (
            Data & data )
```

Push an element onto the stack.
Resizes array if space is inadequate.

Parameters

data	element to be added to stack.

Definition at line 298 of file GStack.h.

12.141.3.6 pop() `template<typename Data >`

`Data & Util::GStack< Data >::pop`

Pop an element off the stack.

Returns the top element by reference and removes it, decrementing the size by one.

Returns

the top element (which is popped off stack).

Definition at line 340 of file GStack.h.

References UTIL_THROW.

12.141.3.7 peek() `[1/2]` `template<typename Data >`

`const Data & Util::GStack< Data >::peek [inline]`

Return a reference to the top element (don't pop).

Definition at line 355 of file GStack.h.

12.141.3.8 peek() `[2/2]` `template<typename Data >`

`const Data& Util::GStack< Data >::peek ( ) const`

Return a const ref to the top element (don't pop).

12.141.3.9 capacity() `template<typename Data >`

`int Util::GStack< Data >::capacity [inline]`

Return allocated size.

Returns

Number of elements allocated in array.

Definition at line 369 of file GStack.h.

12.141.3.10 size() `template<typename Data >`

`int Util::GStack< Data >::size [inline]`

Return logical size.

Returns

logical size of this array.

Definition at line 376 of file GStack.h.

12.141.3.11 isAllocated() `template<class Data >`

`bool Util::GStack< Data >::isAllocated [inline]`

Is this GStack allocated?

Definition at line 383 of file GStack.h.

12.141.3.12 isValid() `template<typename Data >`

`bool Util::GStack< Data >::isValid`

Is this GStack in a valid internal state?

Definition at line 390 of file GStack.h.

References UTIL_THROW.

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

- GStack.h

12.142 Util::IFunctor< T > Class Template Reference

Interface for functor that wraps a void function with one argument (abstract).

`#include <IFunctor.h>`

Public Member Functions

- virtual ~IFunctor ()

  Destructor (virtual)
- virtual void operator() (const T &t)=0

  Call the associated function.

12.142.1 Detailed Description

template<typename T = void>
class Util::IFunctor< T >

Interface for functor that wraps a void function with one argument (abstract).

The operator (const T& t) invokes the associated one-parameter function.

Definition at line 24 of file IFunctor.h.

12.142.2 Constructor & Destructor Documentation

12.142.2.1 ~IFunctor() `template<typename T = void>`

`virtual Util::IFunctor< T >::~IFunctor ( ) [inline], [virtual]`

Destructor (virtual)

Definition at line 31 of file IFunctor.h.

12.142.3 Member Function Documentation

12.142.3.1 operator()() `template<typename T = void>`

```
virtual void Util::IFunctor< T >::operator() (
            const T & t ) [pure virtual]
```

Call the associated function.

Parameters

t	parameter value passed to associated function.

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

- IFunctor.h

12.143 Util::IFunctor< void > Class Reference

Interface for functor that wraps a void function with no arguments (abstract).

`#include <IFunctor.h>`

Inheritance diagram for Util::IFunctor< void >:

classUtil_1_1IFunctor_3_01void_01_4-eps-converted-to.pdf

Public Member Functions

- virtual ~IFunctor ()

    Destructor (virtual)
- virtual void operator() ()=0

    Call a specific member function with one parameter.

12.143.1 Detailed Description

Interface for functor that wraps a void function with no arguments (abstract).

The operator () invokes the associated zero-parameter function.

Definition at line 50 of file IFunctor.h.

12.143.2 Constructor & Destructor Documentation

12.143.2.1 ~IFunctor() `virtual Util::IFunctor< void >::~IFunctor ( ) [inline], [virtual]`

Destructor (virtual)

Definition at line 57 of file IFunctor.h.

12.143.3 Member Function Documentation

12.143.3.1 operator()() `virtual void Util::IFunctor< void >::operator() ( ) [pure virtual]`

Call a specific member function with one parameter.

Implemented in Util::MethodFunctor< Object, void >.

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

- IFunctor.h

12.144 Util::Int Class Reference

Wrapper for an int, for formatted ostream output.

`#include <Int.h>`

Public Member Functions

Constructors

- Int ()

Default constructor.
- Int (int value)

  Constructor, value only.
- Int (int value, int width)

  Constructor, value and width.

Setters

- void setValue (int value)

  Set the integer value.
- void setWidth (int width)

  Set the output field width.

Accessors

- int value ()

  Get the integer value.
- int width ()

  Get the minimum field width.
- std::istream & operator>> (std::istream &in, Int &object)

  Input stream extractor for an Int object.
- std::ostream & operator<< (std::ostream &out, const Int &object)

  Output stream inserter for an Int object.

12.144.1 Detailed Description

Wrapper for an int, for formatted ostream output.
An Int object has a int numerical value, and a minimum output field width. The << operator for an Int uses the specified width. The numerical value and width may both be specified as parameters to a constructor. If the width is not specified as a constructor parameter, it is set within the constructor to a default value equal to Format::defaultWidth().
An Int object may be passed to an ostream as a temporary object. For example, the expression:

```
std::cout « Int(13) « Int(25, 10) « std::endl;
```

outputs the number 13 using the default width, followed by the number 25 in a field of minimum width 10.
Definition at line 36 of file Int.h.

12.144.2 Constructor & Destructor Documentation

12.144.2.1 Int() [1/3] `Util::Int::Int ( )`

Default constructor.
Definition at line 19 of file Int.cpp.

12.144.2.2 Int() [2/3] `Util::Int::Int ( int value ) [explicit]`

Constructor, value only.
Definition at line 27 of file Int.cpp.

12.144.2.3 Int() [3/3] `Util::Int::Int ( int value, int width )`

Constructor, value and width.
Definition at line 35 of file Int.cpp.

12.144.3 Member Function Documentation

12.144.3.1 setValue() `void Util::Int::setValue ( int value )`

Set the integer value.

Parameters

value	value of the associated int variable

Definition at line 43 of file Int.cpp.
References value().

12.144.3.2 setWidth() `void Util::Int::setWidth ( int width )`

Set the output field width.

Parameters

width	output field width

Definition at line 49 of file Int.cpp.
References width().

12.144.3.3 value() `int Util::Int::value ( )`

Get the integer value.
Definition at line 55 of file Int.cpp.
Referenced by setValue().

12.144.3.4 width() `int Util::Int::width ( )`

Get the minimum field width.
Definition at line 61 of file Int.cpp.
Referenced by setWidth().

12.144.4 Friends And Related Function Documentation

12.144.4.1 operator>> `std::istream& operator>> ( std::istream & in, Int & object ) [friend]`

Input stream extractor for an Int object.

Parameters

in	input stream
object	Int object to be read from stream

Returns

modified input stream

Definition at line 71 of file Int.cpp.

12.144.4.2 operator<<

```
std::ostream& operator<< (
    std::ostream & out,
    const Int & object ) [friend]
```

Output stream inserter for an Int object.

Parameters

out	output stream
object	Int to be written to stream

Returns

modified output stream

Definition at line 84 of file Int.cpp.

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

- Int.h
- Int.cpp

12.145 Util::IntDistribution Class Reference

A distribution (or histogram) of values for an int variable.

`#include <IntDistribution.h>`

Inheritance diagram for Util::IntDistribution:

classUtil_1_1IntDistribution-eps-converted-to.pdf

Public Member Functions

- IntDistribution ()

    Default constructor.
- IntDistribution (const IntDistribution &other)

    Copy constructor.
- IntDistribution & operator= (const IntDistribution &other)

    Assignment operator.
- virtual ~IntDistribution ()

    Destructor.

- void readParameters (std::istream &in)

  Read parameters from file and initialize.
- void setParam (int min, int max)

  Set parameters and initialize.
- virtual void loadParameters (Serializable::IArchive &ar)

  Load state from an archive.
- virtual void save (Serializable::OArchive &ar)

  Save state to an archive.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize to/from an archive.
- void clear ()

  Clear (i.e., zero) previously allocated histogram.
- void sample (int value)

  Sample a value.
- void output (std::ostream &out)

  Output the distribution to file.
- int binIndex (int value)

  Return the index of the bin for a value.
- int min () const

  Get minimum value in range of histogram.
- int max () const

  Get maximum value in range of histogram.
- int nBin () const

  Get the number of bins.
- const DArray< long > & data () const

  Get histogram array.

Protected Attributes

- DArray< long > histogram_

  Histogram array.
- int min_

  minimum value.
- int max_

  maximum value.
- int nBin_

  number of bins.
- int nSample_

  Number of sampled values in Histogram.
- int nReject_

  Number of sampled values that were out of range.

Additional Inherited Members

12.145.1 Detailed Description

A distribution (or histogram) of values for an int variable.
Definition at line 22 of file IntDistribution.h.

12.145.2 Constructor & Destructor Documentation

12.145.2.1 IntDistribution() [1/2] Util::IntDistribution::IntDistribution ( )

Default constructor.

Definition at line 18 of file IntDistribution.cpp.

References Util::ParamComposite::setClassName().

12.145.2.2 IntDistribution() [2/2] Util::IntDistribution::IntDistribution (const IntDistribution & *other*)

Copy constructor.

Parameters

other	object to be copied

Definition at line 30 of file IntDistribution.cpp.

References Util::DArray< Data >::allocate(), Util::Array< Data >::capacity(), histogram_, max_, min_, nBin_, nReject↩ _, and nSample_.

12.145.2.3 ∼IntDistribution() Util::IntDistribution::∼IntDistribution ( ) [virtual]

Destructor.

Definition at line 96 of file IntDistribution.cpp.

12.145.3 Member Function Documentation

12.145.3.1 operator=() IntDistribution & Util::IntDistribution::operator= (const IntDistribution & *other*)

Assignment operator.

Parameters

other	object to be assigned

Definition at line 57 of file IntDistribution.cpp.

References Util::DArray< Data >::allocate(), Util::Array< Data >::capacity(), histogram_, max_, min_, nBin_, nReject↩ _, and nSample_.

12.145.3.2 readParameters() void Util::IntDistribution::readParameters (std::istream & *in*) [virtual]

Read parameters from file and initialize.

Read values of min, max, and nBin from file. Allocate histogram array and clear all accumulators.

Parameters

in	input parameter file stream

Reimplemented from Util::ParamComposite.
Definition at line 102 of file IntDistribution.cpp.
References Util::DArray< Data >::allocate(), clear(), histogram_, max_, min_, and nBin_.

12.145.3.3 setParam()

```
void Util::IntDistribution::setParam (
        int min,
        int max )
```

Set parameters and initialize.

Parameters

min	lower bound of range
max	upper bound of range

Definition at line 114 of file IntDistribution.cpp.
References Util::DArray< Data >::allocate(), clear(), histogram_, max(), max_, min(), min_, and nBin_.

12.145.3.4 loadParameters()

```
void Util::IntDistribution::loadParameters (
        Serializable::IArchive & ar )  [virtual]
```

Load state from an archive.

Parameters

ar	binary loading (input) archive.

Reimplemented from Util::ParamComposite.
Definition at line 126 of file IntDistribution.cpp.
References Util::Array< Data >::capacity(), histogram_, max_, min_, nBin_, nReject_, nSample_, and UTIL_THROW.

12.145.3.5 save()

```
void Util::IntDistribution::save (
        Serializable::OArchive & ar )  [virtual]
```

Save state to an archive.

Parameters

ar	binary saving (output) archive.

Reimplemented from Util::ParamComposite.
Definition at line 147 of file IntDistribution.cpp.

12.145.3.6 serialize()

```
template<class Archive >
void Util::IntDistribution::serialize (
        Archive & ar,
        const unsigned int version )
```

Serialize to/from an archive.

Parameters

ar	archive

Parameters

version	archive version id

Definition at line 184 of file IntDistribution.h.
References histogram_, max_, min_, nBin_, nReject_, and nSample_.

12.145.3.7 clear() `void Util::IntDistribution::clear ( )`

Clear (i.e., zero) previously allocated histogram.
Definition at line 153 of file IntDistribution.cpp.
References histogram_, nBin_, nReject_, and nSample_.
Referenced by readParameters(), and setParam().

12.145.3.8 sample() `void Util::IntDistribution::sample ( int value )`

Sample a value.

Parameters

value	current value

Definition at line 165 of file IntDistribution.cpp.
References binIndex(), histogram_, max_, min_, nReject_, and nSample_.

12.145.3.9 output() `void Util::IntDistribution::output ( std::ostream & out )`

Output the distribution to file.

Parameters

out	output stream

Definition at line 180 of file IntDistribution.cpp.
References histogram_, min_, and nBin_.

12.145.3.10 binIndex() `int Util::IntDistribution::binIndex ( int value ) [inline]`

Return the index of the bin for a value.

Parameters

value	sampled value

Definition at line 159 of file IntDistribution.h.
References min_.
Referenced by sample().

12.145.3.11 min() `int Util::IntDistribution::min ( ) const [inline]`

Get minimum value in range of histogram.
Definition at line 165 of file IntDistribution.h.
References min_.
Referenced by setParam().

12.145.3.12 max() `int Util::IntDistribution::max ( ) const [inline]`

Get maximum value in range of histogram.
Definition at line 171 of file IntDistribution.h.
References max_.
Referenced by setParam().

12.145.3.13 nBin() `int Util::IntDistribution::nBin ( ) const [inline]`

Get the number of bins.
Definition at line 177 of file IntDistribution.h.
References nBin_.

12.145.3.14 data() `const DArray<long>& Util::IntDistribution::data ( ) const [inline]`

Get histogram array.
Each element of the histogram array simply contains the number of times that a particular value has been passed to the sample function since the histogram was last cleared.
Definition at line 140 of file IntDistribution.h.
References histogram_.

12.145.4 Member Data Documentation

12.145.4.1 histogram_ `DArray<long> Util::IntDistribution::histogram_ [protected]`

Histogram array.
Definition at line 145 of file IntDistribution.h.
Referenced by clear(), data(), IntDistribution(), loadParameters(), operator=(), output(), readParameters(), sample(), serialize(), and setParam().

12.145.4.2 min_ `int Util::IntDistribution::min_ [protected]`

minimum value.
Definition at line 146 of file IntDistribution.h.
Referenced by binIndex(), IntDistribution(), loadParameters(), min(), operator=(), output(), readParameters(), sample(), serialize(), and setParam().

12.145.4.3 max_ `int Util::IntDistribution::max_ [protected]`

maximum value.
Definition at line 147 of file IntDistribution.h.
Referenced by IntDistribution(), loadParameters(), max(), operator=(), readParameters(), sample(), serialize(), and set←Param().

12.145.4.4 nBin_ `int Util::IntDistribution::nBin_ [protected]`

number of bins.

Definition at line 148 of file IntDistribution.h.

Referenced by clear(), IntDistribution(), loadParameters(), nBin(), operator=(), output(), readParameters(), serialize(), and setParam().

12.145.4.5 nSample_ `int Util::IntDistribution::nSample_ [protected]`

Number of sampled values in Histogram.

Definition at line 149 of file IntDistribution.h.

Referenced by clear(), IntDistribution(), loadParameters(), operator=(), sample(), and serialize().

12.145.4.6 nReject_ `int Util::IntDistribution::nReject_ [protected]`

Number of sampled values that were out of range.

Definition at line 150 of file IntDistribution.h.

Referenced by clear(), IntDistribution(), loadParameters(), operator=(), sample(), and serialize().

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

- IntDistribution.h
- IntDistribution.cpp

12.146 Util::IntVector Class Reference

An IntVector is an integer Cartesian vector.

`#include <IntVector.h>`

Public Member Functions

Constructors

- IntVector ()
  Default constructor.
- IntVector (const IntVector &v)
  Copy constructor.
- IntVector (int scalar)
  Constructor, initialize all elements to the same scalar.
- IntVector (const int ∗v)
  Construct IntVector from C int[3] array.
- IntVector (int x, int y, int z=0)
  Construct IntVector from its coordinates.
- IntVector & zero ()
  Set all elements of a 3D vector to zero.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)
  Serialize to/from an archive.

Assignment

- IntVector & operator= (const IntVector &v)
  Copy assignment.
- IntVector & operator= (const int ∗v)
  Assignment from C int[] array.

Arithmetic Assignment

- void operator+= (const IntVector &dv)

  Add vector dv to this vector.
- void operator-= (const IntVector &dv)

  Subtract vector dv from this vector.
- void operator∗= (int s)

  Multiply this vector by scalar s.

Array Subscript

- const int & operator[ ] (int i) const

  Return one Cartesian element by value.
- int & operator[ ] (int i)

  Return a reference to one element of the vector.

Scalar valued functions

- int square () const

  Return square magnitude of this vector.
- int dot (const IntVector &v) const

  Return dot product of this vector and vector v.

IntVector valued functions (result assigned to invoking object)

- IntVector & add (const IntVector &v1, const IntVector &v2)

  Add vectors v1 and v2.
- IntVector & subtract (const IntVector &v1, const IntVector &v2)

  Subtract vector v2 from v1.
- IntVector & multiply (const IntVector &v, int s)

  Multiply a vector v by a scalar s.
- IntVector & cross (const IntVector &v1, const IntVector &v2)

  Calculate cross product of vectors v1 and v2.

Static Members

- static const IntVector Zero = IntVector(0)

  Zero IntVector.
- static void initStatic ()

  Initialize static IntVector::Zero.
- static void commitMpiType ()

  Commit MPI datatype MpiTraits<IntVector>::type.
- bool operator== (const IntVector &v1, const IntVector &v2)

  Equality for IntVectors.
- bool operator== (const IntVector &v1, const int ∗v2)

  Equality of IntVector and C array.
- std::istream & operator>> (std::istream &in, IntVector &vector)

  istream extractor for a IntVector.
- std::ostream & operator<< (std::ostream &out, const IntVector &vector)

  ostream inserter for a IntVector.

12.146.1 Detailed Description

An IntVector is an integer Cartesian vector.

The Cartesian elements of a IntVector can be accessed using array notation: The elements of a three dimensional IntVector v are v[0], v[1], and v[2]. The subscript operator [] returns elements as references, which can be used on either the left or right side of an assignment operator.

The arithmetic assignment operators +=, -=, *=, and /= are overloaded. The operators += and -= represent increment or decrement by a vector, while *= and /= represent multiplication or division by an integer.

All other unary and binary mathematical operations are implemented as methods. Operations that yield a scalar result, such as a dot product, are implemented as methods that return the resulting value. Operations that yield a IntVector, such as vector addition, are implemented by methods that assign the result to the invoking vector, and return a reference to the invoking vector. For example,

```
IntVector a, b, c;
int s;
a[0] = 0.0
a[1] = 1.0
a[2] = 2.0
b[0] =  0.5
b[1] = -0.5
b[2] = -1.5
// Set s = a.b
s = dot(a, b)
// Set c = a + b
c.add(a, b)
// Set a = a + b
a += b
// Set b = b*2
b *= 2
```

This syntax for IntVector valued operations avoids dynamic allocation of temporary IntVector objects, by requiring that the invoking function provide an object to hold the result.

For efficiency, all methods in this class are inlined.

Definition at line 73 of file IntVector.h.

12.146.2 Constructor & Destructor Documentation

12.146.2.1 IntVector() [1/5] `Util::IntVector::IntVector ( ) [inline]`

Default constructor.

Definition at line 84 of file IntVector.h.

12.146.2.2 IntVector() [2/5] `Util::IntVector::IntVector ( const IntVector & v ) [inline]`

Copy constructor.

Definition at line 90 of file IntVector.h.

12.146.2.3 IntVector() [3/5] `Util::IntVector::IntVector ( int scalar ) [inline], [explicit]`

Constructor, initialize all elements to the same scalar.

Parameters

scalar	initial value for all elements.

Definition at line 102 of file IntVector.h.

12.146.2.4 IntVector() [4/5] Util::IntVector::IntVector (

const int * *v*) [inline], [explicit]

Construct IntVector from C int[3] array.

Parameters

v	array of 3 coordinates

Definition at line 114 of file IntVector.h.

12.146.2.5 IntVector() [5/5] Util::IntVector::IntVector (

int *x*,
int *y*,
int *z* = 0) [inline]

Construct IntVector from its coordinates.

Parameters

x	x-axis coordinate
y	y-axis coordinate
z	z-axis coordinate

Definition at line 128 of file IntVector.h.

12.146.3 Member Function Documentation

12.146.3.1 zero() IntVector& Util::IntVector::zero ( ) [inline]

Set all elements of a 3D vector to zero.
Definition at line 140 of file IntVector.h.

12.146.3.2 serialize() template<class Archive >

void Util::IntVector::serialize (
Archive & *ar*,
const unsigned int *version*) [inline]

Serialize to/from an archive.
Implementation uses syntax of Boost::serialize.

Parameters

ar	archive
version	archive version id

Definition at line 456 of file IntVector.h.

12.146.3.3 operator=() [1/2] IntVector& Util::IntVector::operator= (

const IntVector & *v*) [inline]

Copy assignment.

Parameters

v	IntVector to assign.

Definition at line 167 of file IntVector.h.

12.146.3.4 operator=() [2/2]

`IntVector& Util::IntVector::operator= ( const int * v ) [inline]`

Assignment from C int[] array.

Parameters

v	array of coordinates

Definition at line 180 of file IntVector.h.

12.146.3.5 operator+=()

`void Util::IntVector::operator+= ( const IntVector & dv ) [inline]`

Add vector dv to this vector.
Upon return, ∗this = this + dv.

Parameters

dv	vector increment (input)

Definition at line 199 of file IntVector.h.

12.146.3.6 operator-=()

`void Util::IntVector::operator-= ( const IntVector & dv ) [inline]`

Subtract vector dv from this vector.
Upon return, ∗this = this + dv.

Parameters

dv	vector increment (input)

Definition at line 213 of file IntVector.h.

12.146.3.7 operator∗=()

`void Util::IntVector::operator*= ( int s ) [inline]`

Multiply this vector by scalar s.
Upon return, ∗this = this∗s.

Parameters

s	scalar multiplier

Definition at line 227 of file IntVector.h.

12.146.3.8 operator[]() [1/2] `const int& Util::IntVector::operator[] ( int i ) const [inline]`

Return one Cartesian element by value.

Parameters

i	element index

Returns

element i of the vector

Definition at line 244 of file IntVector.h.
References Util::Dimension.

12.146.3.9 operator[]() [2/2] `int& Util::IntVector::operator[] ( int i ) [inline]`

Return a reference to one element of the vector.

Parameters

i	element index

Returns

element i of the vector

Definition at line 257 of file IntVector.h.
References Util::Dimension.

12.146.3.10 square() `int Util::IntVector::square ( ) const [inline]`

Return square magnitude of this vector.

Returns

square magnitude of this vector

Definition at line 273 of file IntVector.h.

12.146.3.11 dot() `int Util::IntVector::dot ( const IntVector & v ) const [inline]`

Return dot product of this vector and vector v.

Parameters

v	input vector

Returns

dot product of this vector and vector v

Definition at line 282 of file IntVector.h.

12.146.3.12 add()

```
IntVector& Util::IntVector::add (
        const IntVector & v1,
        const IntVector & v2 )  [inline]
```

Add vectors v1 and v2.
Upon return, ∗this = v1 + v2.

Parameters

v1	vector (input)
v2	vector (input)

Definition at line 299 of file IntVector.h.

12.146.3.13 subtract()

```
IntVector& Util::IntVector::subtract (
        const IntVector & v1,
        const IntVector & v2 )  [inline]
```

Subtract vector v2 from v1.
Upon return, ∗this = v1 - v2.

Parameters

v1	vector (input)
v2	vector (input)

Returns

modified invoking vector

Definition at line 316 of file IntVector.h.

12.146.3.14 multiply()

```
IntVector& Util::IntVector::multiply (
        const IntVector & v,
        int s )  [inline]
```

Multiply a vector v by a scalar s.
Upon return, ∗this = v∗s.

Parameters

v	vector input
s	scalar input

Returns

modified invoking vector

Definition at line 333 of file IntVector.h.

12.146.3.15 cross() `IntVector& Util::IntVector::cross ( const IntVector & v1, const IntVector & v2 ) [inline]`

Calculate cross product of vectors v1 and v2.
Upon return, ∗this = v1 x v2.

Parameters

v1	input vector
v2	input vector

Returns

modified invoking vector

Definition at line 350 of file IntVector.h.

12.146.3.16 initStatic() `void Util::IntVector::initStatic ( ) [static]`

Initialize static IntVector::Zero.
Definition at line 113 of file IntVector.cpp.
Referenced by Util::initStatic().

12.146.3.17 commitMpiType() `void Util::IntVector::commitMpiType ( ) [static]`

Commit MPI datatype MpiTraits<IntVector>::type.
Definition at line 92 of file IntVector.cpp.
References Util::MpiStructBuilder::addMember(), Util::MpiStructBuilder::commit(), and Util::MpiStructBuilder::setBase().

12.146.4 Friends And Related Function Documentation

12.146.4.1 operator== [1/2] `bool operator== ( const IntVector & v1, const IntVector & v2 ) [friend]`

Equality for IntVectors.
Definition at line 24 of file IntVector.cpp.

12.146.4.2 operator== [2/2] `bool operator== ( const IntVector & v1, const int * v2 ) [friend]`

Equality of IntVector and C array.
Definition at line 35 of file IntVector.cpp.

12.146.4.3 operator>> `std::istream& operator>> ( std::istream & in, IntVector & vector ) [friend]`

istream extractor for a IntVector.

Input elements of a vector from stream, without line breaks.

Parameters

in	input stream
vector	IntVector to be read from stream

Returns

modified input stream

Definition at line 64 of file IntVector.cpp.

12.146.4.4 operator<< `std::ostream& operator<< ( std::ostream & out, const IntVector & vector ) [friend]`

ostream inserter for a IntVector.

Output elements of a vector to stream, without line breaks.

Parameters

out	output stream
vector	IntVector to be written to stream

Returns

modified output stream

Definition at line 75 of file IntVector.cpp.

12.146.5 Member Data Documentation

12.146.5.1 Zero `const IntVector Util::IntVector::Zero = IntVector(0) [static]`

Zero IntVector.

Definition at line 364 of file IntVector.h.

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

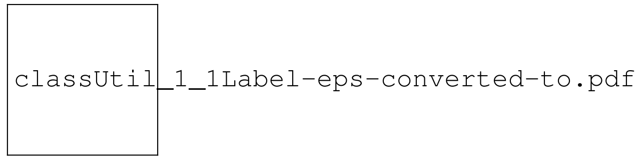
- IntVector.h
- IntVector.cpp

12.147 Util::Label Class Reference

A label string in a file format.

`#include <Label.h>`

Inheritance diagram for Util::Label:



Public Member Functions

- Label (bool isRequired=true)

  Constructor.
- Label (std::string string, bool isRequired=true)

  Constructor.
- Label (const char ∗string, bool isRequired=true)

  Constructor.
- Label (const Label &other)

  Copy constructor.
- virtual ∼Label ()

  Destructor.
- void setString (std::string string)

  Set the label string.
- bool match (std::istream &in)

  Read and attempt to match next word in an input stream.
- std::string string () const

  Return label string.
- bool isRequired () const

  Is this the label for a required component?

Static Public Member Functions

- static void clear ()

  Reset buffer and flags to initial state.
- static bool isClear ()

  Is the input buffer clear?
- static bool isMatched ()

  Did the most recent attempt to match a Label succeed?

Static Public Attributes

- static const int LabelWidth = 20

  Width of label field in file output format.

Friends

- std::istream & operator>> (std::istream &in, Label label)

  Extractor for Label.
- std::ostream & operator<< (std::ostream &out, Label label)

  Inserter for Label.

12.147.1 Detailed Description

A label string in a file format.
The operator $>>$ for a label checks if the expected label was found. The operator $<<$ outputs the expected label.
The constructor takes a parameter isRequired that determines whether the label must be matched (isRequired == true), or if it is optional (isRequired == false). If the input value read by the $>>$ operator does not match the expected value and isRequired is true, the $>>$ operator will print an error message to the Log::file() and then throw an Exception. If the input value does not match and isRequired is false, the $>>$ operator stores the input value in a string buffer, and will compare it to subsequent values until a match is found.
Definition at line 36 of file Label.h.

12.147.2 Constructor & Destructor Documentation

12.147.2.1 Label() [1/4]

```
Util::Label::Label (
    bool isRequired = true )  [explicit]
```

Constructor.

Parameters

isRequired	Is this label a required entry? (true by default)

Definition at line 45 of file Label.cpp.

12.147.2.2 Label() [2/4]

```
Util::Label::Label (
    std::string string,
    bool isRequired = true )
```

Constructor.

Parameters

string	label string that precedes value in file format
isRequired	Is this label a required entry? (true by default)

Definition at line 53 of file Label.cpp.

12.147.2.3 Label() [3/4]

```
Util::Label::Label (
    const char * string,
    bool isRequired = true )
```

Constructor.

Parameters

string	label string that precedes value in file format
isRequired	Is this label a required entry? (true by default)

Definition at line 61 of file Label.cpp.

12.147.2.4 Label() [4/4]

```
Util::Label::Label (
```

`const Label & other )`

Copy constructor.

Parameters

other	Label object being copied.

Definition at line 69 of file Label.cpp.

12.147.2.5 ∼Label() `Util::Label::∼Label ( ) [virtual]`

Destructor.

Definition at line 77 of file Label.cpp.

12.147.3 Member Function Documentation

12.147.3.1 clear() `void Util::Label::clear ( ) [static]`

Reset buffer and flags to initial state.

Clears buffer, sets isClear = true and isMatched = false.

Definition at line 27 of file Label.cpp.

12.147.3.2 isClear() `bool Util::Label::isClear ( ) [static]`

Is the input buffer clear?

Definition at line 37 of file Label.cpp.

Referenced by Util::Begin::readParam().

12.147.3.3 isMatched() `bool Util::Label::isMatched ( ) [inline], [static]`

Did the most recent attempt to match a Label succeed?

Returns true after a succesful match by operator >> or the match() function. Returns false before any attempt to match any Label, after a failed attempt with an an optional label.

Definition at line 203 of file Label.h.

Referenced by Util::Parameter::readParam().

12.147.3.4 setString() `void Util::Label::setString (`

`std::string string )`

Set the label string.

Parameters

string	label string that precedes value in file format

Definition at line 83 of file Label.cpp.

References string().

Referenced by Util::Begin::Begin().

12.147.3.5 match() `bool Util::Label::match (`

`std::istream & in )`

Read and attempt to match next word in an input stream.

This is a convenience function that invokes operator >> to read a word and then returns the value of Label::isMatched(). For an optional Label, this returns true upon a successful match and false otherwise. For a required label, returns true upon a successful match or throws an Exception.

Definition at line 95 of file Label.cpp.

12.147.3.6 string() `std::string Util::Label::string ( ) const`

Return label string.

Definition at line 89 of file Label.cpp.

Referenced by Util::Parameter::label(), setString(), and Util::Begin::writeParam().

12.147.3.7 isRequired() `bool Util::Label::isRequired ( ) const [inline]`

Is this the label for a required component?

Definition at line 197 of file Label.h.

Referenced by Util::Begin::isRequired(), Util::Parameter::isRequired(), and Util::operator>>().

12.147.4 Friends And Related Function Documentation

12.147.4.1 operator>> `std::istream& operator>> (`

`std::istream & in,`

`Label label ) [friend]`

Extractor for Label.

Parameters

in	input stream
label	Label to be read from file

Definition at line 104 of file Label.cpp.

12.147.4.2 operator<< `std::ostream& operator<< (`

`std::ostream & out,`

`Label label ) [friend]`

Inserter for Label.

Parameters

out	output stream
label	Label to be written to file

Definition at line 158 of file Label.cpp.

12.147.5 Member Data Documentation

12.147.5.1 LabelWidth `const int Util::Label::LabelWidth = 20 [static]`

Width of label field in file output format.
Definition at line 44 of file Label.h.
Referenced by Util::operator<<().
The documentation for this class was generated from the following files:

- Label.h
- Label.cpp

12.148 Util::List< Data > Class Template Reference

Linked list class template.
`#include <List.h>`

Public Member Functions

- List ()

  Default constructor.
- virtual ∼List ()

  Destructor (does nothing).
- void initialize (Node< Data > ∗nodes, int capacity)

  Provide an array of Node<Data> objects for this List.
- int size () const

  Get the number of elements.
- int capacity () const

  Get capacity of the array.
- void pushBack (Node< Data > &node)

  Push a node onto the the back of the List.
- void pushFront (Node< Data > &node)

  Push a node onto the the front of the List.
- Node< Data > & popBack ()

  Remove a node from the back of the list.
- Node< Data > & popFront ()

  Remove a node from the front of the list.
- void insertNext (Node< Data > &node, Node< Data > &newNode)

  Insert newNode into list after node.
- void insertPrev (Node< Data > &node, Node< Data > &newNode)

  Insert newNode into list before node.
- void insert (Node< Data > &node)

  Insert node into list in sequential order.
- void remove (Node< Data > &node)

  Remove node from list.
- void begin (ListIterator< Data > &iterator) const

  Set an iterator to the front of this List.
- bool isValid () const

  Check validity of linked list.

12.148.1 Detailed Description

template<typename Data>
class Util::List< Data >

Linked list class template.
This list implementation is based on an underlying C array of Node<Data> objects. This array may be used by several List objects, and so must be allocated outside the Link class and provided via the initialize method.
Definition at line 32 of file List.h.

12.148.2 Constructor & Destructor Documentation

12.148.2.1 List() `template<typename Data >`

`Util::List< Data >::List`

Default constructor.
Definition at line 192 of file List.h.

12.148.2.2 ~List() `template<typename Data >`

`virtual Util::List< Data >::~List ( ) [inline], [virtual]`

Destructor (does nothing).
Definition at line 45 of file List.h.

12.148.3 Member Function Documentation

12.148.3.1 initialize() `template<typename Data >`

```
void Util::List< Data >::initialize (
            Node< Data > * nodes,
            int capacity )
```

Provide an array of Node<Data> objects for this List.
Definition at line 206 of file List.h.

12.148.3.2 size() `template<typename Data >`

`int Util::List< Data >::size [inline]`

Get the number of elements.

Returns

Number of elements in this list.

Definition at line 217 of file List.h.

12.148.3.3 capacity() `template<typename Data >`

`int Util::List< Data >::capacity [inline]`

Get capacity of the array.

Returns

Number of elements allocated in the associated arrays.

Definition at line 224 of file List.h.

12.148.3.4 pushBack()

```
template<typename Data >
void Util::List< Data >::pushBack (
            Node< Data > & node )
```

Push a node onto the the back of the List.

Parameters

node	Node object from associated node array.

Definition at line 231 of file List.h.
References Util::Node< Data >::attachNext(), Util::Node< Data >::setList(), and Util::Node< Data >::setNext().

12.148.3.5 pushFront()

```
template<typename Data >
void Util::List< Data >::pushFront (
            Node< Data > & node )
```

Push a node onto the the front of the List.

Parameters

node	Node object from associated node array.

Definition at line 251 of file List.h.
References Util::Node< Data >::attachPrev(), Util::Node< Data >::setList(), and Util::Node< Data >::setPrev().

12.148.3.6 popBack()

```
template<typename Data >
Node< Data > & Util::List< Data >::popBack
```

Remove a node from the back of the list.

Returns

Node that was removed from this list.

Definition at line 273 of file List.h.
References Util::Node< Data >::clear(), Util::Node< Data >::prev(), and UTIL_THROW.

12.148.3.7 popFront()

```
template<typename Data >
Node< Data > & Util::List< Data >::popFront
```

Remove a node from the front of the list.

Returns

Node that was removed from this list.

Definition at line 298 of file List.h.
References Util::Node< Data >::clear(), Util::Node< Data >::next(), and UTIL_THROW.

12.148.3.8 insertNext()

```
template<typename Data >
void Util::List< Data >::insertNext (
            Node< Data > & node,
            Node< Data > & newNode )
```

Insert newNode into list after node.

Parameters

node	Node in the existing list.
newNode	new node, to be inserted as the next after node.

Definition at line 323 of file List.h.
References Util::Node< Data >::attachNext(), Util::Node< Data >::next(), and Util::Node< Data >::setNext().

12.148.3.9 insertPrev() `template<typename Data >`

```
void Util::List< Data >::insertPrev (
            Node< Data > & node,
            Node< Data > & newNode )
```

Insert newNode into list before node.

Parameters

node	Node in the existing list.
newNode	new Node, to be inserted previous to node.

Definition at line 340 of file List.h.
References Util::Node< Data >::attachPrev(), Util::Node< Data >::prev(), and Util::Node< Data >::setPrev().

12.148.3.10 insert() `template<typename Data >`

```
void Util::List< Data >::insert (
            Node< Data > & node )
```

Insert node into list in sequential order.

Parameters

node	Node to be inserted into the list.

Definition at line 386 of file List.h.
References Util::Node< Data >::attachNext(), Util::Node< Data >::list(), Util::Node< Data >::next(), Util::Node< Data >::prev(), Util::Node< Data >::setList(), Util::Node< Data >::setNext(), and Util::Node< Data >::setPrev().

12.148.3.11 remove() `template<typename Data >`

```
void Util::List< Data >::remove (
            Node< Data > & node )
```

Remove node from list.

Parameters

node	Node to be removed from the list.

Definition at line 357 of file List.h.
References Util::Node< Data >::clear(), Util::Node< Data >::list(), Util::Node< Data >::next(), and Util::Node< Data >::prev().

12.148.3.12 begin() template<typename Data >

```
void Util::List< Data >::begin (
            ListIterator< Data > & iterator ) const
```

Set an iterator to the front of this List.

Parameters

iterator	ListIterator, initialized on output.

Definition at line 449 of file List.h.
References Util::ListIterator< Data >::setCurrent().
Referenced by Util::ListIterator< Data >::ListIterator().

12.148.3.13 isValid() template<typename Data >

```
bool Util::List< Data >::isValid
```

Check validity of linked list.

Returns

true if the list is valid, false otherwise.

Definition at line 459 of file List.h.
References Util::Node< Data >::list(), Util::Node< Data >::next(), Util::Node< Data >::prev(), and UTIL_THROW.
The documentation for this class was generated from the following file:

- List.h

12.149 Util::ListArray< Data > Class Template Reference

An array of objects that are accessible by one or more linked List objects.

```
#include <ListArray.h>
```

Public Member Functions

- ListArray ()
  Constructor.
- virtual ∼ListArray ()
  Destructor.
- void allocate (int capacity, int nList)
  Allocate arrays of Node and List objects.
- int nList () const
  Get the number of associated linked lists.
- int capacity () const
  Return allocated size of underlying array of nodes.
- Data & operator[ ] (int i)
  Return data for node element i.
- const Data & operator[ ] (int i) const
  Return const refereence to Data in Node element number i.
- List< Data > & list (int i)
  Return a reference to a specific List.
- const List< Data > & list (int i) const
  Return a const reference to a specific List.

- Node< Data > & node (int i)

    Return reference to node number i.

- bool isValid () const

    Return true if the ListAray is valid, or throw an exception.

12.149.1 Detailed Description

template<typename Data>
class Util::ListArray< Data >

An array of objects that are accessible by one or more linked List objects.
A ListArray is an allocatable array of data objects that also provides access to some or all of its via one or more associated List objects. Each element of the array may be part of at most one List.
Definition at line 30 of file ListArray.h.

12.149.2 Constructor & Destructor Documentation

12.149.2.1 ListArray() `template<typename Data >`

`Util::ListArray< Data >::ListArray`

Constructor.
Definition at line 148 of file ListArray.h.

12.149.2.2 ~ListArray() `template<typename Data >`

`Util::ListArray< Data >::~ListArray [virtual]`

Destructor.
Delete dynamically allocated arrays of Node and List objects.
Definition at line 161 of file ListArray.h.

12.149.3 Member Function Documentation

12.149.3.1 allocate() `template<typename Data >`

```
void Util::ListArray< Data >::allocate (
            int capacity,
            int nList )
```

Allocate arrays of Node and List objects.

Parameters

capacity	size of array Node<Data> objects
nList	size of array of List<Data> linked list objects

Definition at line 178 of file ListArray.h.

12.149.3.2 nList() `template<typename Data >`

`int Util::ListArray< Data >::nList [inline]`

Get the number of associated linked lists.

Returns

size of array of associated List<Data> objects.

Definition at line 201 of file ListArray.h.

12.149.3.3 capacity() `template<typename Data >`

`int Util::ListArray< Data >::capacity ( ) const [inline]`

Return allocated size of underlying array of nodes.

Returns

Number of elements allocated in array.

Definition at line 67 of file ListArray.h.

12.149.3.4 operator[]() [1/2] `template<typename Data >`

```
Data& Util::ListArray< Data >::operator[] (
            int i ) [inline]
```

Return data for node element i.

Parameters

i	array index

Returns

reference to element i

Definition at line 76 of file ListArray.h.

12.149.3.5 operator[]() [2/2] `template<typename Data >`

```
const Data& Util::ListArray< Data >::operator[] (
            int i ) const [inline]
```

Return const refereence to Data in Node element number i.

Parameters

i	array index

Returns

const reference to element i

Definition at line 90 of file ListArray.h.

12.149.3.6 list() [1/2] `template<typename Data >`

```
List< Data > & Util::ListArray< Data >::list (
            int i )
```

Return a reference to a specific List.

Parameters

i	array index

Returns

reference to List number i

Definition at line 211 of file ListArray.h.

12.149.3.7 list() [2/2]

```
template<typename Data >
const List< Data > & Util::ListArray< Data >::list (
            int i ) const
```

Return a const reference to a specific List.

Parameters

i	array index

Returns

reference to List number i

Definition at line 227 of file ListArray.h.

12.149.3.8 node()

```
template<typename Data >
Node< Data > & Util::ListArray< Data >::node (
            int i )
```

Return reference to node number i.

Parameters

i	array index

Returns

reference to Data object element number i

Definition at line 243 of file ListArray.h.

12.149.3.9 isValid()

```
template<typename Data >
bool Util::ListArray< Data >::isValid
```

Return true if the ListAray is valid, or throw an exception.
Definition at line 256 of file ListArray.h.
References UTIL_THROW.
The documentation for this class was generated from the following file:

- ListArray.h

12.150 Util::ListIterator< Data > Class Template Reference

Bidirectional iterator for a List.

```
#include <ListIterator.h>
```

Public Member Functions

- ListIterator ()

  Default constructor.
- ListIterator (const List< Data > &list)

  Constructor for initialized iterator.
- void setCurrent (Node< Data > *nodePtr)

  Point the iterator at a Node.
- bool isEnd () const

  Has the end of the list been reached?
- bool isBack () const

  Is this the back of the List?
- bool isFront () const

  Is this the front of the List?

Operators

- const Data & operator* () const

  Get a const reference to the associated Data object.
- Data & operator* ()

  Get the associated Data object.
- const Data * operator-> () const

  Get a pointer to const to the associated Data object.
- Data * operator-> ()

  Get a pointer to the associated Data object.
- ListIterator< Data > & operator++ ()

  Go to the next element in a linked list.
- ListIterator< Data > & operator-- ()

  Go to the previous element in a linked list.

12.150.1 Detailed Description

template<typename Data>
class Util::ListIterator< Data >

Bidirectional iterator for a List.
A ListIterator provides bidirectional input/output access to a linked list, similar to an STL bidirectional iterator. An * operator returns a reference to an associated Data object. The ++ and – operators change the current pointer to the next or prev element in a list.
The isEnd() method returns true if either end of the list has already been passed by a previous ++ or – operation. When isEnd() is true, the iterator is no longer usable, since it no longer points to a Node and cannot be incremented or decremented.
Definition at line 20 of file List.h.

12.150.2 Constructor & Destructor Documentation

12.150.2.1 ListIterator() [1/2]

`template<typename Data >`
`Util::ListIterator< Data >::ListIterator ( ) [inline]`

Default constructor.

Creates a "dead" iterator, for which isEnd()==true. Before it can be used, such an iterator must be initialized by either the ListIterator<Data>::setCurrent() method or the List<Data>::begin() method of an associated List.

Definition at line 40 of file ListIterator.h.

12.150.2.2 ListIterator() [2/2]

`template<typename Data >`
`Util::ListIterator< Data >::ListIterator ( const List< Data > & list ) [inline], [explicit]`

Constructor for initialized iterator.

Creates an iterator that points to the front of a List. Calls List<Data>::begin(*this) internally.

Parameters

list	parent List

Definition at line 52 of file ListIterator.h.

References Util::List< Data >::begin().

12.150.3 Member Function Documentation

12.150.3.1 setCurrent()

`template<typename Data >`
`void Util::ListIterator< Data >::setCurrent ( Node< Data > * nodePtr ) [inline]`

Point the iterator at a Node.

Parameters

nodePtr	pointer to current Node in a List, or null.

Definition at line 61 of file ListIterator.h.

Referenced by Util::List< Data >::begin().

12.150.3.2 isEnd()

`template<typename Data >`
`bool Util::ListIterator< Data >::isEnd ( ) const [inline]`

Has the end of the list been reached?

Return true if the current pointer is null, indicating that the previous increment or decrement passed an end of the list.

Returns

true if current node is null, false otherwise.

Definition at line 72 of file ListIterator.h.

12.150.3.3 isBack()

`template<typename Data >`
`bool Util::ListIterator< Data >::isBack ( ) const [inline]`

Is this the back of the List?

Returns

true if current node is the back Node, false otherwise.

Definition at line 80 of file ListIterator.h.

12.150.3.4 isFront() `template<typename Data >`

`bool Util::ListIterator< Data >::isFront ( ) const [inline]`

Is this the front of the List?

Returns

true if current node is the front Node, false otherwise.

Definition at line 88 of file ListIterator.h.

12.150.3.5 operator*() `[1/2]` `template<typename Data >`

`const Data& Util::ListIterator< Data >::operator* ( ) const [inline]`

Get a const reference to the associated Data object.

Returns

const reference to the associated Data object

Definition at line 99 of file ListIterator.h.

12.150.3.6 operator*() `[2/2]` `template<typename Data >`

`Data& Util::ListIterator< Data >::operator* ( ) [inline]`

Get the associated Data object.

Returns

reference to associated Data object

Definition at line 107 of file ListIterator.h.

12.150.3.7 operator->() `[1/2]` `template<typename Data >`

`const Data* Util::ListIterator< Data >::operator-> ( ) const [inline]`

Get a pointer to const to the associated Data object.

Returns

pointer to associated Data object

Definition at line 115 of file ListIterator.h.

12.150.3.8 operator->() `[2/2]` `template<typename Data >`

`Data* Util::ListIterator< Data >::operator-> ( ) [inline]`

Get a pointer to the associated Data object.

Returns

pointer to associated Data object

Definition at line 123 of file ListIterator.h.

12.150.3.9 operator++() `template<typename Data >`

`ListIterator<Data>& Util::ListIterator< Data >::operator++ ( ) [inline]`

Go to the next element in a linked list.

This method assigns the current pointer to the address of the next Node in the list, and then returns *this. If there is no next Node, the current pointer is set null, and any subsequent call to isEnd() will return true.

Returns

this ListIterator, after modification.

Definition at line 136 of file ListIterator.h.

12.150.3.10 operator--() `template<typename Data >`

`ListIterator<Data>& Util::ListIterator< Data >::operator-- ( ) [inline]`

Go to the previous element in a linked list.

This method assigns the current Node to the previous in the List, and returns a reference to *this.

Returns

this ListIterator

Definition at line 150 of file ListIterator.h.

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

- List.h
- ListIterator.h

12.151 Util::Lng Class Reference

Wrapper for a long int, for formatted ostream output.

`#include <Lng.h>`

Public Member Functions

Constructors

- Lng ()
  Default constructor.
- Lng (long int value)
  Constructor, value only.
- Lng (long int value, int width)
  Constructor, value and width.

Setters

- void setValue (long int value)
  Set value of long int.
- void setWidth (int width)
  Set field width.

Accessors

- long int value ()
  Get value of long int.
- int width ()
  Get field width.

- std::istream & operator>> (std::istream &in, Lng &object)

  Input stream extractor for an Lng object.
- std::ostream & operator<< (std::ostream &out, const Lng &object)

  Output stream inserter for an Lng object.

12.151.1 Detailed Description

Wrapper for a long int, for formatted ostream output.

An Lng object has a long int numerical value, and a minimum output field width. The << operator for an Lng uses the specified width. The numerical value and width may both be optionally specified as parameters to a constructor. If the width is not specified, it is is is set to a default value equal to Format::defaultWidth().

An Lng object may be passed to an ostream as a temporary object. For example, the expression:

```
std::cout « Lng(13) « Lng(25, 10) « std::endl;
```

outputs the number 13 using the default width, followed by the number 25 in a field of minimum width 10.

Definition at line 35 of file Lng.h.

12.151.2 Constructor & Destructor Documentation

12.151.2.1 Lng() [1/3] `Util::Lng::Lng ( )`

Default constructor.

Definition at line 15 of file Lng.cpp.

12.151.2.2 Lng() [2/3] `Util::Lng::Lng ( long int value ) [explicit]`

Constructor, value only.

Parameters

value	associated long int

Definition at line 22 of file Lng.cpp.

12.151.2.3 Lng() [3/3] `Util::Lng::Lng ( long int value, int width )`

Constructor, value and width.

Parameters

value	associated long int
width	field width

Definition at line 29 of file Lng.cpp.

12.151.3 Member Function Documentation

12.151.3.1 setValue() `void Util::Lng::setValue ( long int value )`

Set value of long int.

Parameters

value	associated long int

Definition at line 34 of file Lng.cpp.
References value().

12.151.3.2 setWidth() `void Util::Lng::setWidth ( int width )`

Set field width.

Parameters

width	field width

Definition at line 37 of file Lng.cpp.
References width().

12.151.3.3 value() `long int Util::Lng::value ( )`

Get value of long int.
Definition at line 40 of file Lng.cpp.
Referenced by setValue().

12.151.3.4 width() `int Util::Lng::width ( )`

Get field width.
Definition at line 43 of file Lng.cpp.
Referenced by setWidth().

12.151.4 Friends And Related Function Documentation

12.151.4.1 operator>> `std::istream& operator>> ( std::istream & in, Lng & object ) [friend]`

Input stream extractor for an Lng object.

Parameters

in	input stream
object	Lng object to be read from stream

Returns

modified input stream

Definition at line 53 of file Lng.cpp.

12.151.4.2 operator<<

```
std::ostream& operator<< (
    std::ostream & out,
    const Lng & object ) [friend]
```

Output stream inserter for an Lng object.

Parameters

out	output stream
object	Lng to be written to stream

Returns

modified output stream

Definition at line 66 of file Lng.cpp.
The documentation for this class was generated from the following files:

- Lng.h
- Lng.cpp

12.152 Util::Log Class Reference

A static class that holds a log output stream.
`#include <Log.h>`

Static Public Member Functions

- static void initStatic ()

  Initialize static members.
- static void setFile (std::ofstream &file)

  Set the log ostream to a file.
- static void close ()

  Close log file, if any.
- static std::ostream & file ()

  Get log ostream by reference.

12.152.1 Detailed Description

A static class that holds a log output stream.
The Log class has one a static pointer member that points to an ostream that should be used by all other classes to output log and error messages. This stream is accessed by the file() method.
The log file initialized to point to std::cout. It may be reset to point to a ofstream file object using the static setFile() method.
Definition at line 30 of file Log.h.

12.152.2 Member Function Documentation

12.152.2.1 initStatic() `void Util::Log::initStatic ( ) [static]`

Initialize static members.

Definition at line 23 of file Log.cpp.

Referenced by Util::initStatic().

12.152.2.2 setFile() `void Util::Log::setFile ( std::ofstream & file ) [static]`

Set the log ostream to a file.

Parameters

file	ofstream open for writing.

Definition at line 36 of file Log.cpp.

References file().

12.152.2.3 close() `void Util::Log::close ( ) [static]`

Close log file, if any.

Definition at line 45 of file Log.cpp.

Referenced by Util::MpiThrow().

12.152.2.4 file() `std::ostream & Util::Log::file ( ) [static]`

Get log ostream by reference.

Definition at line 57 of file Log.cpp.

Referenced by Pscf::SpaceGroup< D >::checkMeshDimensions(), Util::checkString(), Util::Manager< Data >::endReadManager(), Util::Exception::Exception(), Util::XmlStartTag::finish(), Pscf::Pspg::Continuous::AmIterator< D >::isConverged(), Util::Parameter::load(), Util::ParamComposite::load(), Util::Factory< Data >::loadObject(), Util::ParamComposite::loadOptional(), Pscf::Basis< D >::makeBasis(), Util::XmlEndTag::match(), Util::XmlStartTag::matchLabel(), Util::MpiThrow(), Util::operator>>(), Pscf::Pspg::Continuous::System< D >::readCommands(), Util::Factory< Data >::readObject(), Util::End::readParam(), Util::Blank::readParam(), Util::Begin::readParam(), Util::Parameter::readParam(), setFile(), Pscf::Pspg::Continuous::System< D >::setOptions(), and Pscf::Pspg::Continuous::AmIterator< D >::solve().

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

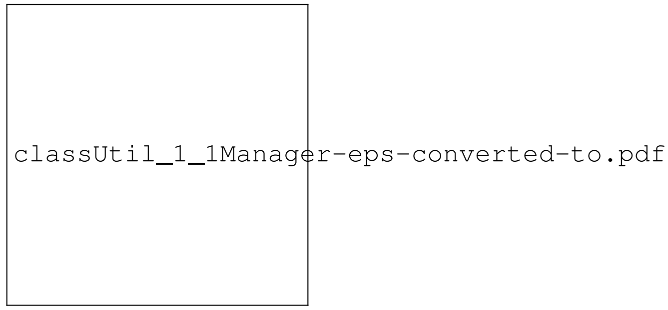
- Log.h
- Log.cpp

12.153 Util::Manager< Data > Class Template Reference

Template container for pointers to objects with a common base class.

`#include <Manager.h>`

Inheritance diagram for Util::Manager< Data >:



Public Member Functions

- Manager (bool uniqueNames=false)

  Constructor.
- virtual ∼Manager ()

  Destructor.
- void addSubfactory (Factory< Data > &subfactory)

  Set a SubFactory for this Manager.
- void setFactory (Factory< Data > &factory)

  Associate a Factory with this Manager.
- void setFactory (Factory< Data > ∗factoryPtr)

  Associated a Factory with this Manager (pass by pointer).
- virtual void readParam (std::istream &in)

  Read and create a set of objects.
- virtual void readParamOptional (std::istream &in)

  Optionally read and create a set of objects.
- virtual void readParameters (std::istream &in)

  Read child blocks, return when closing bracket encountered.
- virtual void loadParameters (Serializable::IArchive &ar)

  Load a set of objects to an output archive.
- virtual void save (Serializable::OArchive &ar)

  Save a set of objects to an output archive.
- void append (Data &data, const std::string &name)

  Append a Data object to the end of the sequence.
- int size () const

  Get logical size.
- std::string className (int i) const

  Get the subclass name for object number i.
- Factory< Data > & factory ()

  Return a reference to the factory.
- bool hasFactory () const

  Return true if this Manager has a Factory, false otherwise.
- Data & operator[ ] (int i) const

  Mimic C array subscripting.
- Data ∗ findFirst (std::string const &className)

  Return pointer to first object with specified class name.

Protected Member Functions

- void beginReadManager (std::istream &in)

  Read (or attempt to read) opening line: "ManagerName{".
- void endReadManager ()

  Add closing bracket to output format.
- virtual void initFactory ()

  Create factory if necessary.
- virtual Factory< Data > ∗ newDefaultFactory () const

  Create an instance of the default Factory<Data> class.

Protected Attributes

- Factory< Data > ∗ factoryPtr_

  Pointer to an associated Factory<Data> object.

Additional Inherited Members

12.153.1 Detailed Description

template<typename Data>
class Util::Manager< Data >

Template container for pointers to objects with a common base class.
A Manager<Data> has an array of Data∗ pointers to Data objects, an array of corresponding subclass names, and a pointer to a Factory<Data> object. The default implementation of the Manager<Data>::readParam() method uses an associated Factory<Data> object to recognize the class name string that begins a polymorphic block in a parameter file (which must refer to a known subclass of Data) and to instantiate an object of the specified subclass.
Subclasses of Manager<Data> are used to manage arrays of Species, McMove, and Analyzer objects.
Definition at line 38 of file Manager.h.

12.153.2 Constructor & Destructor Documentation

12.153.2.1 Manager() `template<typename Data >`

```
Util::Manager< Data >::Manager (
            bool uniqueNames = false )
```

Constructor.

Parameters

uniqueNames	set true to require unique element class names.

Definition at line 232 of file Manager.h.

12.153.2.2 ∼Manager() `template<typename Data >`

```
Util::Manager< Data >::∼Manager [virtual]
```

Destructor.
Definition at line 246 of file Manager.h.

12.153.3 Member Function Documentation

12.153.3.1 addSubfactory() template<typename Data >

```
void Util::Manager< Data >::addSubfactory (
            Factory< Data > & subfactory )
```

Set a SubFactory for this Manager.

Parameters

subfactory	Referernce to a sub-Factory to be added.

Definition at line 275 of file Manager.h.

12.153.3.2 setFactory() [1/2] template<typename Data >

```
void Util::Manager< Data >::setFactory (
            Factory< Data > & factory )
```

Associate a Factory with this Manager.

Parameters

factory	Reference to a Factory object

Definition at line 284 of file Manager.h.

12.153.3.3 setFactory() [2/2] template<typename Data >

```
void Util::Manager< Data >::setFactory (
            Factory< Data > * factoryPtr )
```

Associated a Factory with this Manager (pass by pointer).

Parameters

factoryPtr	pointer to a Factory<Data> object.

Definition at line 297 of file Manager.h.

12.153.3.4 readParam() template<typename Data >

```
void Util::Manager< Data >::readParam (
            std::istream & in )  [virtual]
```

Read and create a set of objects.

The default implementation of this method reads a sequence of blocks for different subclasses of Data, terminated by a closing bracket. For each block it:

- reads a className string for a subclass of Data,
- uses factory object to create a new instance of className.
- invokes the readParam() method of the new object.

The implementation of the factory must recognize all valid className string values, and invoke the appropriate constructor for each. The loop over blocks terminates when it encounters a closing bracket '}' surrounded by white space.

Parameters

in	input stream

Reimplemented from Util::ParamComposite.
Definition at line 310 of file Manager.h.

12.153.3.5 readParamOptional() template<typename Data >

```
void Util::Manager< Data >::readParamOptional (
            std::istream & in ) [virtual]
```

Optionally read and create a set of objects.
Equivalent to readParam(), except that this function does nothing if the first line does not match the expected label, whereas readParam() throws an Exception

Parameters

in	input stream

Reimplemented from Util::ParamComposite.
Definition at line 323 of file Manager.h.

12.153.3.6 readParameters() template<typename Data >

```
void Util::Manager< Data >::readParameters (
            std::istream & in ) [virtual]
```

Read child blocks, return when closing bracket encountered.

Parameters

in	input stream

Reimplemented from Util::ParamComposite.
Definition at line 356 of file Manager.h.
References UTIL_THROW.

12.153.3.7 loadParameters() template<typename Data >

```
void Util::Manager< Data >::loadParameters (
            Serializable::IArchive & ar ) [virtual]
```

Load a set of objects to an output archive.

Parameters

ar	input/loading archive

Reimplemented from Util::ParamComposite.
Definition at line 411 of file Manager.h.
References UTIL_THROW.

12.153.3.8 save() template<typename Data >

```
void Util::Manager< Data >::save (
             Serializable::OArchive & ar )  [virtual]
```

Save a set of objects to an output archive.

Parameters

ar	output/saving archive

Reimplemented from Util::ParamComposite.
Definition at line 448 of file Manager.h.

12.153.3.9 append() `template<typename Data >`

```
void Util::Manager< Data >::append (
             Data & data,
             const std::string & name )
```

Append a Data object to the end of the sequence.

Parameters

data	Data object to be appended
name	subclass name

Definition at line 462 of file Manager.h.

12.153.3.10 size() `template<typename Data >`

```
int Util::Manager< Data >::size  [inline]
```

Get logical size.

Returns

logical size of this array.

Definition at line 494 of file Manager.h.

12.153.3.11 className() `template<typename Data >`

```
std::string Util::Manager< Data >::className (
             int i ) const
```

Get the subclass name for object number i.

Parameters

i	integer index of object

Returns

class name of managed object

Definition at line 473 of file Manager.h.
References UTIL_THROW.

12.153.3.12 factory() `template<typename Data >`

`Factory< Data > & Util::Manager< Data >::factory`

Return a reference to the factory.

Definition at line 265 of file Manager.h.

12.153.3.13 hasFactory() `template<typename Data >`

`bool Util::Manager< Data >::hasFactory`

Return true if this Manager has a Factory, false otherwise.

Definition at line 485 of file Manager.h.

12.153.3.14 operator[]() `template<typename Data >`

```
Data & Util::Manager< Data >::operator[] (
             int i ) const  [inline]
```

Mimic C array subscripting.

Parameters

i	array index

Returns

reference to element i

Definition at line 501 of file Manager.h.

12.153.3.15 findFirst() `template<typename Data >`

```
Data * Util::Manager< Data >::findFirst (
             std::string const & className )
```

Return pointer to first object with specified class name.

Parameters

className	desired class name string

Returns

pointer to specified objectd, or null if not found.

Definition at line 512 of file Manager.h.

12.153.3.16 beginReadManager() `template<typename Data >`

```
void Util::Manager< Data >::beginReadManager (
             std::istream & in )  [protected]
```

Read (or attempt to read) opening line: "ManagerName{".

Definition at line 338 of file Manager.h.

References Util::ParamComposite::className(), and Util::Begin::isActive().

12.153.3.17 endReadManager() `template<typename Data >`

`void Util::Manager< Data >::endReadManager [protected]`

Add closing bracket to output format.
Definition at line 399 of file Manager.h.
References Util::ParamComponent::echo(), Util::Log::file(), and Util::End::writeParam().

12.153.3.18 initFactory() `template<typename Data >`

`void Util::Manager< Data >::initFactory [protected], [virtual]`
Create factory if necessary.
Definition at line 528 of file Manager.h.

12.153.3.19 newDefaultFactory() `template<typename Data >`

`Factory< Data > * Util::Manager< Data >::newDefaultFactory [protected], [virtual]`
Create an instance of the default Factory<Data> class.

Returns

a pointer to a new Factory<Data> object.

Definition at line 541 of file Manager.h.
References UTIL_THROW.

12.153.4 Member Data Documentation

12.153.4.1 factoryPtr_ `template<typename Data >`

`Factory<Data>* Util::Manager< Data >::factoryPtr_ [protected]`
Pointer to an associated Factory<Data> object.
Definition at line 182 of file Manager.h.
The documentation for this class was generated from the following file:

- Manager.h

12.154 Util::Matrix< Data > Class Template Reference

Two-dimensional array container template (abstract).
`#include <Matrix.h>`
Inheritance diagram for Util::Matrix< Data >:

classUtil_1_1Matrix-eps-converted-to.pdf

Public Member Functions

- virtual ~Matrix ()

    Destructor.
- int capacity1 () const

    Get number of rows (range of the first array index).
- int capacity2 () const

    Get number of columns (range of the second array index).

- const Data & operator() (int i, int j) const

    Return element (i,j) of matrix by const reference.
- Data & operator() (int i, int j)

    Return element (i,j) of matrix by reference.
- Data ∗ cArray ()

    Return pointer to underlying one-dimensional C array.
- const Data ∗ cArray () const

    Return pointer to const to underlying one-dimensional C array.

Protected Member Functions

- Matrix ()

    Default constructor.

Protected Attributes

- Data ∗ data_

    Pointer to 1D C array of all elements.
- int capacity1_

    Number of rows (range of first index).
- int capacity2_

    Number of columns (range of first index).

12.154.1 Detailed Description

template<typename Data>
class Util::Matrix< Data >

Two-dimensional array container template (abstract).
An Matrix object A is a two-dimensional array in which the operator A(i, j) returns a reference to element in column j of row i.
The memory for a Matrix is stored in a single one-dimensional C array, in which each row is stored as a consecutive block.
Class Matrix is an abstract class because it cannot allocate memory. Concrete subclasses include DMatrix and FMatrix.
Definition at line 31 of file Matrix.h.

12.154.2 Constructor & Destructor Documentation

12.154.2.1 ∼Matrix() `template<typename Data >`

`Util::Matrix< Data >::∼Matrix [virtual]`
Destructor.
Definition at line 129 of file Matrix.h.

12.154.2.2 Matrix() `template<typename Data >`

`Util::Matrix< Data >::Matrix [inline], [protected]`
Default constructor.
Constructor (protected).
Protected to prevent direct instantiation.
Definition at line 119 of file Matrix.h.

12.154.3 Member Function Documentation

12.154.3.1 capacity1() template<typename Data >

int Util::Matrix< Data >::capacity1 [inline]

Get number of rows (range of the first array index).

Definition at line 136 of file Matrix.h.

Referenced by Util::bcast(), Pscf::LuSolver::computeLU(), Util::recv(), and Util::send().

12.154.3.2 capacity2() template<typename Data >

int Util::Matrix< Data >::capacity2 [inline]

Get number of columns (range of the second array index).

Definition at line 143 of file Matrix.h.

Referenced by Util::bcast(), Pscf::LuSolver::computeLU(), Util::recv(), and Util::send().

12.154.3.3 operator()() [1/2] template<typename Data >

const Data & Util::Matrix< Data >::operator() (
    int *i*,
    int *j*) const [inline]

Return element (i,j) of matrix by const reference.

Parameters

i	row index.
j	column index.

Definition at line 150 of file Matrix.h.

12.154.3.4 operator()() [2/2] template<typename Data >

Data & Util::Matrix< Data >::operator() (
    int *i*,
    int *j*) [inline]

Return element (i,j) of matrix by reference.

Parameters

i	row index.
j	column index.

Definition at line 164 of file Matrix.h.

12.154.3.5 cArray() [1/2] template<typename Data >

const Data * Util::Matrix< Data >::cArray [inline]

Return pointer to underlying one-dimensional C array.

Definition at line 178 of file Matrix.h.

Referenced by Pscf::LuSolver::inverse().

12.154.3.6 cArray() [2/2] template<typename Data >

const Data∗ Util::Matrix< Data >::cArray ( ) const

Return pointer to const to underlying one-dimensional C array.

12.154.4 Member Data Documentation

12.154.4.1 data_ template<typename Data >

Data∗ Util::Matrix< Data >::data_ [protected]

Pointer to 1D C array of all elements.

Definition at line 84 of file Matrix.h.

Referenced by Util::DMatrix< Type >::DMatrix(), and Util::DMatrix< Type >::operator=().

12.154.4.2 capacity1_ template<typename Data >

int Util::Matrix< Data >::capacity1_ [protected]

Number of rows (range of first index).

Definition at line 87 of file Matrix.h.

Referenced by Util::DMatrix< Type >::DMatrix(), and Util::DMatrix< Type >::operator=().

12.154.4.3 capacity2_ template<typename Data >

int Util::Matrix< Data >::capacity2_ [protected]

Number of columns (range of first index).

Definition at line 90 of file Matrix.h.

Referenced by Util::DMatrix< Type >::DMatrix(), and Util::DMatrix< Type >::operator=().

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

- Matrix.h

12.155 Util::MeanSqDispArray< Data > Class Template Reference

Mean-squared displacement (MSD) vs.

#include <MeanSqDispArray.h>

Inheritance diagram for Util::MeanSqDispArray< Data >:

classUtil_1_1MeanSqDispArray-eps-converted-to.pdf

Public Member Functions

- void readParameters (std::istream &in)

  Read parameters, allocate memory and clear history.
- void setParam (int ensembleCapacity, int bufferCapacity)

Set parameters, allocate memory, and clear history.
- virtual void loadParameters (Serializable::IArchive &ar)

  Load internal state from an archive.
- virtual void save (Serializable::OArchive &ar)

  Save internal state to an archive.
- template< class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize this MeanSqDispArray to/from an archive.
- void setNEnsemble (int nEnsemble)

  Set actual number of sequences in ensemble.
- void clear ()

  Reset to empty state.
- void sample (const Array< Data > &values)

  Sample an array of current values.
- void output (std::ostream &out)

  Output the autocorrelation function.
- int bufferCapacity ()

  Return capacity of the history buffer for each sequence.
- int nEnsemble ()

  Return number of sequences in the ensemble.
- int nSample ()

  Return number of values sampled from each sequence thus far.

Additional Inherited Members

12.155.1 Detailed Description

template< typename Data >
class Util::MeanSqDispArray< Data >

Mean-squared displacement (MSD) vs.
time for an ensembles of sequences.
This class calculates the mean-squared difference <|x(i) - x(i-j)|^2> for an ensemble of statistically equivalent sequences x(i) of values of a variable of type Data. The meaning of |a - b|^2 is defined for int, double, and Vector data by explicit specializations of the private method double sqDiff(Data&, Data).
Definition at line 41 of file MeanSqDispArray.h.

12.155.2 Member Function Documentation

12.155.2.1 readParameters() `template<typename Data >`

```
void Util::MeanSqDispArray< Data >::readParameters (
            std::istream & in ) [virtual]
```

Read parameters, allocate memory and clear history.
Reads parameters nEnsemble and capacity, allocates memory, and then calls clear().

Parameters

in	input parameter stream

Reimplemented from Util::ParamComposite.

Definition at line 217 of file MeanSqDispArray.h.

12.155.2.2 setParam() template<typename Data >

```
void Util::MeanSqDispArray< Data >::setParam (
            int ensembleCapacity,
            int bufferCapacity )
```

Set parameters, allocate memory, and clear history.

Sets parameters nEnsemble and capacity, allocates memory and then calls clear().

Parameters

ensembleCapacity	number of sequence in ensemble
bufferCapacity	number of variable values per sequence

Definition at line 230 of file MeanSqDispArray.h.

12.155.2.3 loadParameters() template<typename Data >

```
void Util::MeanSqDispArray< Data >::loadParameters (
            Serializable::IArchive & ar )  [virtual]
```

Load internal state from an archive.

Parameters

ar	input/loading archive

Reimplemented from Util::ParamComposite.

Definition at line 256 of file MeanSqDispArray.h.

12.155.2.4 save() template<typename Data >

```
void Util::MeanSqDispArray< Data >::save (
            Serializable::OArchive & ar )  [virtual]
```

Save internal state to an archive.

Parameters

ar	output/saving archive

Reimplemented from Util::ParamComposite.

Definition at line 287 of file MeanSqDispArray.h.

12.155.2.5 serialize() template<typename Data >

```
template<class Archive >
void Util::MeanSqDispArray< Data >::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize this MeanSqDispArray to/from an archive.

Parameters

ar	input or output archive
version	file version id

Definition at line 272 of file MeanSqDispArray.h.

12.155.2.6 setNEnsemble()

```
template<typename Data >
void Util::MeanSqDispArray< Data >::setNEnsemble (
        int nEnsemble )
```

Set actual number of sequences in ensemble.

Precondition

readParameters() or setParam() must have been called previously

nEnsemble <= ensembleCapacity

Parameters

nEnsemble	actual number of sequences in ensemble

Definition at line 243 of file MeanSqDispArray.h.
References UTIL_THROW.

12.155.2.7 clear()

```
template<typename Data >
void Util::MeanSqDispArray< Data >::clear
```

Reset to empty state.
Definition at line 294 of file MeanSqDispArray.h.
References Util::setToZero().

12.155.2.8 sample()

```
template<typename Data >
void Util::MeanSqDispArray< Data >::sample (
        const Array< Data > & values )  [inline]
```

Sample an array of current values.

Parameters

values	Array of current values

Definition at line 340 of file MeanSqDispArray.h.

12.155.2.9 output()

```
template<typename Data >
void Util::MeanSqDispArray< Data >::output (
        std::ostream & out )
```

Output the autocorrelation function.
Definition at line 408 of file MeanSqDispArray.h.

12.155.2.10 bufferCapacity() template<typename Data >

`int Util::MeanSqDispArray< Data >::bufferCapacity ( ) [inline]`

Return capacity of the history buffer for each sequence.
Definition at line 130 of file MeanSqDispArray.h.

12.155.2.11 nEnsemble() template<typename Data >

`int Util::MeanSqDispArray< Data >::nEnsemble ( ) [inline]`

Return number of sequences in the ensemble.
Definition at line 136 of file MeanSqDispArray.h.

12.155.2.12 nSample() template<typename Data >

`int Util::MeanSqDispArray< Data >::nSample ( ) [inline]`

Return number of values sampled from each sequence thus far.
Definition at line 142 of file MeanSqDispArray.h.
The documentation for this class was generated from the following file:

- MeanSqDispArray.h

12.156 Util::Memory Class Reference

Provides method to allocate array.

`#include <Memory.h>`

Static Public Member Functions

- template<typename Data >
  static void allocate (Data ∗&ptr, size_t size)

  Allocate a C++ array.
- template<typename Data >
  static void deallocate (Data ∗&ptr, size_t size)

  Deallocate a C++ array.
- template<typename Data >
  static void reallocate (Data ∗&ptr, size_t oldSize, size_t newSize)

  Reallocate a C++ array.
- static int nAllocate ()

  Return number of times allocate() was called.
- static int nDeallocate ()

  Return number of times deallocate() was called.
- static int total ()

  Return total amount of memory currently allocated.
- static int max ()

  Return the maximum amount of allocated heap memory thus far.
- static int max (MPI::Intracomm &communicator)

  Return max for any processor in communicator.
- static void initStatic ()

  Call this just to guarantee initialization of static memory.

12.156.1 Detailed Description

Provides method to allocate array.
The Memory::allocate() method invokes the new operator within a try catch block, and keeps track of the total memory allocated.
Definition at line 28 of file Memory.h.

12.156.2 Member Function Documentation

12.156.2.1 allocate() template<typename Data >

```
void Util::Memory::allocate (
            Data *& ptr,
            size_t size )  [static]
```

Allocate a C++ array.
Uses new to allocates a Data array of size elements, assigns ptr the address of the first element.

Parameters

ptr	reference to pointer (output)
size	number of elements

Definition at line 132 of file Memory.h.
References UTIL_THROW.
Referenced by Pscf::Field< T >::Field(), and reallocate().

12.156.2.2 deallocate() template<typename Data >

```
void Util::Memory::deallocate (
            Data *& ptr,
            size_t size )  [static]
```

Deallocate a C++ array.
Uses free to deallocate a Data array of size elements.

Parameters

ptr	reference to pointer (intput, ptr = 0 on output)
size	number of elements in existing array

Definition at line 152 of file Memory.h.
References UTIL_CHECK.
Referenced by reallocate(), Util::GPArray< Data >::∼GPArray(), and Util::GStack< Data >::∼GStack().

12.156.2.3 reallocate() template<typename Data >

```
void Util::Memory::reallocate (
            Data *& ptr,
            size_t oldSize,
            size_t newSize )  [static]
```

Reallocate a C++ array.
This function calls allocate to allocate a new array, copies all existing elements and deallocates old and calls deallocate to free the old array. On outputs, ptr is the address of the new array.

Precondition: On input, newSize > oldSize.

Parameters

ptr	reference to pointer (input/output)
oldSize	number of elements in existing array
newSize	number of elements in new array

Definition at line 168 of file Memory.h.
References allocate(), deallocate(), and UTIL_CHECK.

12.156.2.4 nAllocate() `int Util::Memory::nAllocate ( ) [static]`

Return number of times allocate() was called.
Each call to reallocate() also increments nAllocate(), because allocate() is called internally.
Definition at line 34 of file Memory.cpp.

12.156.2.5 nDeallocate() `int Util::Memory::nDeallocate ( ) [static]`

Return number of times deallocate() was called.
Each call to reallocate() also increments nDeallocate(), because deallocate() is called internally.
Definition at line 40 of file Memory.cpp.

12.156.2.6 total() `int Util::Memory::total ( ) [static]`

Return total amount of memory currently allocated.
Definition at line 46 of file Memory.cpp.

12.156.2.7 max() `[1/2]` `int Util::Memory::max ( ) [static]`

Return the maximum amount of allocated heap memory thus far.
This function returns the temporal maximum of total().
Definition at line 52 of file Memory.cpp.

12.156.2.8 max() `[2/2]` `int Util::Memory::max ( MPI::Intracomm & communicator ) [static]`

Return max for any processor in communicator.
Definition at line 56 of file Memory.cpp.

12.156.2.9 initStatic() `void Util::Memory::initStatic ( ) [static]`

Call this just to guarantee initialization of static memory.
Definition at line 28 of file Memory.cpp.
Referenced by Util::initStatic().
The documentation for this class was generated from the following files:

- Memory.h
- Memory.cpp

12.157 Util::MemoryCounter Class Reference

Archive to computed packed size of a sequence of objects, in bytes.

`#include <MemoryCounter.h>`

Public Member Functions

- MemoryCounter ()
  Constructor.
- ∼MemoryCounter ()
  Destructor.
- void clear ()
  Resets the size counter to zero.
- template<typename T >
  MemoryCounter & operator& (T &data)
  Add packed size of one object.
- template<typename T >
  MemoryCounter & operator<< (T &data)
  Add packed size of one object.
- template<typename T >
  void count (const T &data)
  Add size of one object in memory.
- template<typename T >
  void count (T ∗array, int n)
  Compute the size in memory of a C array.
- size_t size () const
  Return size required for archive, in Bytes.

Static Public Member Functions

- static bool is_saving ()
  Returns true.
- static bool is_loading ()
  Returns false.

12.157.1 Detailed Description

Archive to computed packed size of a sequence of objects, in bytes.
This class computes the number of bytes required to pack a sequence of objects within a MemoryOArchive. The interface is that of a loading Archive, but the << and & operators are overloaded to compute the size required for an object and to increment a size counter, rather than to actually save data.
The size() method returns the number of bytes required to pack all of the objects serialized thus far. The size counter is set to zero upon construction. The clear() method resets the size counter to zero.
Definition at line 35 of file MemoryCounter.h.

12.157.2 Constructor & Destructor Documentation

12.157.2.1 MemoryCounter() `Util::MemoryCounter::MemoryCounter ( )`

Constructor.
Definition at line 16 of file MemoryCounter.cpp.

12.157.2.2 ∼MemoryCounter() Util::MemoryCounter::∼MemoryCounter ( )

Destructor.
Definition at line 24 of file MemoryCounter.cpp.

12.157.3 Member Function Documentation

12.157.3.1 is_saving() bool Util::MemoryCounter::is_saving ( ) [inline], [static]

Returns true.
Definition at line 139 of file MemoryCounter.h.

12.157.3.2 is_loading() bool Util::MemoryCounter::is_loading ( ) [inline], [static]

Returns false.
Definition at line 142 of file MemoryCounter.h.

12.157.3.3 clear() void Util::MemoryCounter::clear ( )

Resets the size counter to zero.
Definition at line 30 of file MemoryCounter.cpp.

12.157.3.4 operator&() template<typename T >

MemoryCounter & Util::MemoryCounter::operator& (
T & *data*) [inline]

Add packed size of one object.
Definition at line 155 of file MemoryCounter.h.

12.157.3.5 operator<<() template<typename T >

MemoryCounter & Util::MemoryCounter::operator<< (
T & *data*) [inline]

Add packed size of one object.
Definition at line 165 of file MemoryCounter.h.

12.157.3.6 count() [1/2] template<typename T >

void Util::MemoryCounter::count (
const T & *data*) [inline]

Add size of one object in memory.
This method just increments the size by sizeof(T). It is appropriate only for primitive C++ variables and POD types for which a bitwise copy is appropriate.
Definition at line 177 of file MemoryCounter.h.

12.157.3.7 count() [2/2] template<typename T >

void Util::MemoryCounter::count (
T ∗ *array*,
int *n*) [inline]

Compute the size in memory of a C array.
This method increments the size by n∗sizeof(T). It is appropriate for C arrays of primitive variables and of POD types for which a bitwise copy is appropriate.

Definition at line 184 of file MemoryCounter.h.

12.157.3.8 size() `size_t Util::MemoryCounter::size ( ) const [inline]`

Return size required for archive, in Bytes.
Definition at line 148 of file MemoryCounter.h.
Referenced by Util::memorySize().
The documentation for this class was generated from the following files:

- MemoryCounter.h
- MemoryCounter.cpp

12.158 Util::MemoryIArchive Class Reference

Input archive for packed heterogeneous binary data.
`#include <MemoryIArchive.h>`

Public Member Functions

- MemoryIArchive ()

    Constructor.
- ∼MemoryIArchive ()

    Destructor.
- void allocate (size_t capacity)

    Allocate memory block.
- MemoryIArchive & operator= (MemoryOArchive &other)

    Assignment from MemoryOArchive.
- void reset ()

    Reset the cursor to the beginning (for rereading).
- void clear ()

    Reset to empty state.
- void release ()

    Release memory obtained by assignment.
- template<typename T >
  MemoryIArchive & operator& (T &data)

    Load one object.
- template<typename T >
  MemoryIArchive & operator>> (T &data)

    Load one object.
- template<typename T >
  void unpack (T &data)

    Unpack one object of type T.
- template<typename T >
  void unpack (T ∗array, int n)

    Read a C-array of objects of type T.
- template<typename T >
  void unpack (T ∗array, int m, int n, int np)

    Unpack a 2D C array.
- void recv (MPI::Intracomm &comm, int source)

    Receive packed data via MPI.

- Byte ∗ begin () const

  Return pointer to beginning of block.
- Byte ∗ cursor () const

  Return pointer to current position (cursor).
- Byte ∗ end () const

  Return pointer to end of packed block (one Byte past the last).
- size_t capacity () const

  Return capacity in Bytes.
- bool isAllocated () const

  Has memory been allocated?

Static Public Member Functions

- static bool is_saving ()

  Returns true;.
- static bool is_loading ()

  Returns false;.

12.158.1 Detailed Description

Input archive for packed heterogeneous binary data.
Definition at line 31 of file MemoryIArchive.h.

12.158.2 Constructor & Destructor Documentation

12.158.2.1 MemoryIArchive() `Util::MemoryIArchive::MemoryIArchive ( )`

Constructor.
Definition at line 20 of file MemoryIArchive.cpp.

12.158.2.2 ∼MemoryIArchive() `Util::MemoryIArchive::∼MemoryIArchive ( )`

Destructor.
Definition at line 35 of file MemoryIArchive.cpp.

12.158.3 Member Function Documentation

12.158.3.1 is_saving() `bool Util::MemoryIArchive::is_saving ( ) [inline], [static]`

Returns true;.
Definition at line 196 of file MemoryIArchive.h.

12.158.3.2 is_loading() `bool Util::MemoryIArchive::is_loading ( ) [inline], [static]`

Returns false;.
Definition at line 199 of file MemoryIArchive.h.

12.158.3.3 allocate()

```
void Util::MemoryIArchive::allocate (
        size_t capacity )
```

Allocate memory block.

Parameters

capacity	sizeof of block, in Bytes.

Definition at line 46 of file MemoryIArchive.cpp.
References capacity(), and UTIL_THROW.

12.158.3.4 operator=() `MemoryIArchive & Util::MemoryIArchive::operator= ( MemoryOArchive & other )`

Assignment from MemoryOArchive.
Definition at line 64 of file MemoryIArchive.cpp.
References isAllocated(), and UTIL_THROW.

12.158.3.5 reset() `void Util::MemoryIArchive::reset ( )`

Reset the cursor to the beginning (for rereading).
Definition at line 84 of file MemoryIArchive.cpp.
References begin(), isAllocated(), and UTIL_THROW.

12.158.3.6 clear() `void Util::MemoryIArchive::clear ( )`

Reset to empty state.
Resets cursor and end pointers to beginning of memory block.
Definition at line 95 of file MemoryIArchive.cpp.
References begin(), isAllocated(), and UTIL_THROW.

12.158.3.7 release() `void Util::MemoryIArchive::release ( )`

Release memory obtained by assignment.
Definition at line 110 of file MemoryIArchive.cpp.
References UTIL_THROW.

12.158.3.8 operator&() `template<typename T >`

```
MemoryIArchive & Util::MemoryIArchive::operator& (
              T & data )  [inline]
```

Load one object.

Parameters

data	object to be loaded from this archive.

Definition at line 240 of file MemoryIArchive.h.

12.158.3.9 operator>>() `template<typename T >`

```
MemoryIArchive & Util::MemoryIArchive::operator>> (
              T & data )  [inline]
```

Load one object.

Parameters

data	object to be loaded from this archive.

Definition at line 250 of file MemoryIArchive.h.

12.158.3.10 unpack() [1/3] `template<typename T >`

```
void Util::MemoryIArchive::unpack (
            T & data )
```

Unpack one object of type T.

Parameters

data	object to be loaded from this archive.

Definition at line 260 of file MemoryIArchive.h.
References UTIL_THROW.

12.158.3.11 unpack() [2/3] `template<typename T >`

```
void Util::MemoryIArchive::unpack (
            T * array,
            int n )
```

Read a C-array of objects of type T.

Parameters

array	array into which data should be loaded.
n	expected number of elements in the array.

Definition at line 275 of file MemoryIArchive.h.
References UTIL_THROW.

12.158.3.12 unpack() [3/3] `template<typename T >`

```
void Util::MemoryIArchive::unpack (
            T * array,
            int m,
            int n,
            int np )
```

Unpack a 2D C array.
Unpack m rows of n elements into array of type T array[mp][np], with m <= mp and n <= np.

Parameters

array	pointer to [0][0] element of 2D array
m	logical number of rows
n	logical number of columns
np	physical number of columns

Definition at line 292 of file MemoryIArchive.h.
References UTIL_THROW.

12.158.3.13 recv() `void Util::MemoryIArchive::recv ( MPI::Intracomm & comm, int source )`

Receive packed data via MPI.

Parameters

comm	MPI communicator
source	rank of processor from which data is sent.

Definition at line 133 of file MemoryIArchive.cpp.
References UTIL_THROW.

12.158.3.14 begin() `Byte * Util::MemoryIArchive::begin ( ) const [inline]`

Return pointer to beginning of block.
Definition at line 207 of file MemoryIArchive.h.
Referenced by clear(), and reset().

12.158.3.15 cursor() `Byte * Util::MemoryIArchive::cursor ( ) const [inline]`

Return pointer to current position (cursor).
Definition at line 213 of file MemoryIArchive.h.

12.158.3.16 end() `Byte * Util::MemoryIArchive::end ( ) const [inline]`

Return pointer to end of packed block (one Byte past the last).
Definition at line 219 of file MemoryIArchive.h.

12.158.3.17 capacity() `size_t Util::MemoryIArchive::capacity ( ) const [inline]`

Return capacity in Bytes.
Definition at line 225 of file MemoryIArchive.h.
Referenced by allocate().

12.158.3.18 isAllocated() `bool Util::MemoryIArchive::isAllocated ( ) const [inline]`

Has memory been allocated?
Definition at line 231 of file MemoryIArchive.h.
Referenced by clear(), operator=(), and reset().
The documentation for this class was generated from the following files:

- MemoryIArchive.h
- MemoryIArchive.cpp

12.159 Util::MemoryOArchive Class Reference

Save archive for packed heterogeneous binary data.

```
#include <MemoryOArchive.h>
```

Public Member Functions

- MemoryOArchive ()
  Constructor.
- virtual ∼MemoryOArchive ()
  Destructor.
- virtual void allocate (size_t capacity)
  Allocate memory.
- void clear ()
  Resets the cursor to the beginning.
- template<typename T >
  void operator& (T &data)
  Save one object.
- template<typename T >
  MemoryOArchive & operator<< (T &data)
  Save one object.
- template<typename T >
  void pack (const T &data)
  Pack a T object.
- template<typename T >
  void pack (const T ∗array, int n)
  Pack a C array.
- template<typename T >
  void pack (const T ∗array, int m, int n, int np)
  Pack a 2D C array.
- void send (MPI::Intracomm &comm, int dest)
  Send packed data via MPI.
- void iSend (MPI::Intracomm &comm, MPI::Request &req, int dest)
  Send packed data via MPI (non-blocking)
- Byte ∗ begin () const
  Return pointer to beginning of block.
- Byte ∗ cursor () const
  Return pointer to current position (cursor).
- size_t capacity () const
  Return capacity in Bytes.
- bool isAllocated () const
  Has memory been allocated?

Static Public Member Functions

- static bool is_saving ()
  Returns true;.
- static bool is_loading ()
  Returns false;.

12.159.1 Detailed Description

Save archive for packed heterogeneous binary data.
Definition at line 31 of file MemoryOArchive.h.

12.159.2 Constructor & Destructor Documentation

12.159.2.1 MemoryOArchive() `Util::MemoryOArchive::MemoryOArchive ( )`

Constructor.
Definition at line 19 of file MemoryOArchive.cpp.

12.159.2.2 ∼MemoryOArchive() `Util::MemoryOArchive::∼MemoryOArchive ( ) [virtual]`

Destructor.
Definition at line 33 of file MemoryOArchive.cpp.

12.159.3 Member Function Documentation

12.159.3.1 is_saving() `bool Util::MemoryOArchive::is_saving ( ) [inline], [static]`

Returns true;.
Definition at line 188 of file MemoryOArchive.h.

12.159.3.2 is_loading() `bool Util::MemoryOArchive::is_loading ( ) [inline], [static]`

Returns false;.
Definition at line 191 of file MemoryOArchive.h.

12.159.3.3 allocate() `void Util::MemoryOArchive::allocate ( size_t capacity ) [virtual]`

Allocate memory.

Parameters

capacity	size of memory block in bytes

Definition at line 44 of file MemoryOArchive.cpp.
References begin(), capacity(), and UTIL_THROW.

12.159.3.4 clear() `void Util::MemoryOArchive::clear ( )`

Resets the cursor to the beginning.
Definition at line 62 of file MemoryOArchive.cpp.
References begin(), isAllocated(), and UTIL_THROW.

12.159.3.5 operator&() `template<typename T > void Util::MemoryOArchive::operator& ( T & data ) [inline]`

Save one object.
Definition at line 224 of file MemoryOArchive.h.
References Util::serialize().

12.159.3.6 operator<<() `template<typename T >`

```
MemoryOArchive & Util::MemoryOArchive::operator<< (
            T & data ) [inline]
```

Save one object.

Definition at line 231 of file MemoryOArchive.h.

References Util::serialize().

12.159.3.7 pack() [1/3] `template<typename T >`

```
void Util::MemoryOArchive::pack (
            const T & data ) [inline]
```

Pack a T object.

Definition at line 243 of file MemoryOArchive.h.

References UTIL_THROW.

12.159.3.8 pack() [2/3] `template<typename T >`

```
void Util::MemoryOArchive::pack (
            const T * array,
            int n ) [inline]
```

Pack a C array.

Parameters

array	C array
n	number of elements

Definition at line 261 of file MemoryOArchive.h.

References UTIL_THROW.

12.159.3.9 pack() [3/3] `template<typename T >`

```
void Util::MemoryOArchive::pack (
            const T * array,
            int m,
            int n,
            int np ) [inline]
```

Pack a 2D C array.

Pack m rows of n elements from array of type T array[mp][np], with n <= np and m <= mp.

Parameters

array	poiner to [0][0] element of 2D array
m	logical number of rows
n	logical number of columns
np	physical number of columns

Definition at line 281 of file MemoryOArchive.h.

References UTIL_THROW.

12.159.3.10 send() `void Util::MemoryOArchive::send (`

```
MPI::Intracomm & comm,
int dest )
```

Send packed data via MPI.

Parameters

comm	MPI communicator
dest	rank of processor to which data is sent

Definition at line 74 of file MemoryOArchive.cpp.
References UTIL_THROW.

12.159.3.11 iSend() `void Util::MemoryOArchive::iSend (`

```
MPI::Intracomm & comm,
MPI::Request & req,
int dest )
```

Send packed data via MPI (non-blocking)

Parameters

comm	MPI communicator
req	MPI request
dest	rank of processor to which data is sent

Definition at line 97 of file MemoryOArchive.cpp.
References UTIL_THROW.

12.159.3.12 begin() `Byte * Util::MemoryOArchive::begin ( ) const [inline]`

Return pointer to beginning of block.
Definition at line 211 of file MemoryOArchive.h.
Referenced by allocate(), and clear().

12.159.3.13 cursor() `Byte * Util::MemoryOArchive::cursor ( ) const [inline]`

Return pointer to current position (cursor).
Definition at line 217 of file MemoryOArchive.h.

12.159.3.14 capacity() `size_t Util::MemoryOArchive::capacity ( ) const [inline]`

Return capacity in Bytes.
Definition at line 205 of file MemoryOArchive.h.
Referenced by allocate().

12.159.3.15 isAllocated() `bool Util::MemoryOArchive::isAllocated ( ) const [inline]`

Has memory been allocated?
Definition at line 199 of file MemoryOArchive.h.
Referenced by clear().
The documentation for this class was generated from the following files:

- MemoryOArchive.h
- MemoryOArchive.cpp

12.160 Util::MethodFunctor< Object, T > Class Template Reference

Functor that wraps a one-argument class member function.

`#include <MethodFunctor.h>`

Inheritance diagram for Util::MethodFunctor< Object, T >:

classUtil_1_1MethodFunctor-eps-converted-to.pdf

Public Member Functions

- MethodFunctor (Object &object, Method1Ptr methodPtr)

  Constructor.
- virtual ~MethodFunctor ()

  Destructor.
- virtual void operator() (const T &t)

  Operator ().

12.160.1 Detailed Description

template<class Object, typename T = void>
class Util::MethodFunctor< Object, T >

Functor that wraps a one-argument class member function.

The constructor to MethodFunctor<T> takes pointers to an invoking instance of class Object and a member function that takes one T argument. The operator () (const T&) invokes that method on that object.

Definition at line 26 of file MethodFunctor.h.

12.160.2 Constructor & Destructor Documentation

12.160.2.1 MethodFunctor() template<class Object , typename T = void>

```
Util::MethodFunctor< Object, T >::MethodFunctor (
            Object & object,
            Method1Ptr methodPtr )  [inline]
```

Constructor.

Parameters

object	invoking object
methodPtr	pointer to member function

Definition at line 38 of file MethodFunctor.h.

12.160.2.2 ∼MethodFunctor()

```
template<class Object , typename T = void>
virtual Util::MethodFunctor< Object, T >::~MethodFunctor ( ) [inline], [virtual]
```

Destructor.
Definition at line 46 of file MethodFunctor.h.

12.160.3 Member Function Documentation

12.160.3.1 operator()()

```
template<class Object , typename T = void>
virtual void Util::MethodFunctor< Object, T >::operator() (
            const T & t ) [inline], [virtual]
```

Operator ().

Parameters

t	Parameter passed to method of associated T object.

Definition at line 53 of file MethodFunctor.h.
The documentation for this class was generated from the following file:

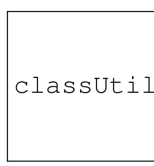
- MethodFunctor.h

12.161 Util::MethodFunctor< Object, void > Class Template Reference

Functor that wraps a class member function with no arguments.

```
#include <MethodFunctor.h>
```

Inheritance diagram for Util::MethodFunctor< Object, void >:



Public Member Functions

- MethodFunctor (Object &object, Method0Ptr methodPtr)

    Constructor.

- virtual ∼MethodFunctor ()

    Destructor.

- virtual void operator() ()

    Call a specific member function with one parameter.

12.161.1 Detailed Description

template<class Object>
class Util::MethodFunctor< Object, void >

Functor that wraps a class member function with no arguments.
Definition at line 67 of file MethodFunctor.h.

12.161.2 Constructor & Destructor Documentation

12.161.2.1 MethodFunctor() `template<class Object >`

`Util::MethodFunctor< Object, void >::MethodFunctor ( Object & object, Method0Ptr methodPtr ) [inline]`

Constructor.

Parameters

object	invoking object
methodPtr	pointer to member function

Definition at line 79 of file MethodFunctor.h.

12.161.2.2 ∼MethodFunctor() `template<class Object >`

`virtual Util::MethodFunctor< Object, void >::∼MethodFunctor ( ) [inline], [virtual]`

Destructor.

Definition at line 87 of file MethodFunctor.h.

12.161.3 Member Function Documentation

12.161.3.1 operator()() `template<class Object >`

`virtual void Util::MethodFunctor< Object, void >::operator() ( ) [inline], [virtual]`

Call a specific member function with one parameter.

Implements Util::IFunctor< void >.

Definition at line 89 of file MethodFunctor.h.

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

- MethodFunctor.h

12.162 Util::MpiFileIo Class Reference

Identifies whether this processor may do file I/O.

`#include <MpiFileIo.h>`

Inheritance diagram for Util::MpiFileIo:

classUtil_1_1MpiFileIo-eps-converted-to.pdf

Public Member Functions

- MpiFileIo ()

    Constructor.
- MpiFileIo (const MpiFileIo &other)

    Copy constructor.
- bool isIoProcessor () const

    Can this processor do file I/O ?
- void setIoCommunicator (MPI::Intracomm &communicator)

    Set the communicator.
- void clearCommunicator ()

    Clear (nullify) the communicator.
- bool hasIoCommunicator () const

    Does this object have an associated MPI communicator?
- MPI::Intracomm & ioCommunicator () const

    Get the MPI communicator by reference.

12.162.1 Detailed Description

Identifies whether this processor may do file I/O.

The return value of isIoProcessor() indicates whether this processor can read and write to file. If the the class is compiled with UTIL_MPI not defined, then isIoProcessor() always returns true. If the class is compiled with UTIL_MPI defined, then this method returns true if either: (1) A communicator has been set and this processor has rank 0 within that communicator, or (2) No communicator has been set.

When compiled with UTIL_MPI defined, an MpiFileIo object has a pointer to an MPI communiciator, and provides methods to set and unset (nullify) the associated communicator.

Definition at line 33 of file MpiFileIo.h.

12.162.2 Constructor & Destructor Documentation

12.162.2.1 MpiFileIo() [1/2] `Util::MpiFileIo::MpiFileIo ( )`

Constructor.

Definition at line 18 of file MpiFileIo.cpp.

12.162.2.2 MpiFileIo() [2/2] `Util::MpiFileIo::MpiFileIo ( const MpiFileIo & other )`

Copy constructor.
Definition at line 28 of file MpiFileIo.cpp.

12.162.3 Member Function Documentation

12.162.3.1 isIoProcessor() `bool Util::MpiFileIo::isIoProcessor ( ) const [inline]`

Can this processor do file I/O ?
Definition at line 92 of file MpiFileIo.h.
Referenced by Util::Parameter::load(), Util::ParamComposite::load(), Util::ParamComposite::loadOptional(), Util::↩Blank::readParam(), Util::End::readParam(), Util::Begin::readParam(), and Util::Parameter::readParam().

12.162.3.2 setIoCommunicator() `void Util::MpiFileIo::setIoCommunicator ( MPI::Intracomm & communicator )`

Set the communicator.
Definition at line 36 of file MpiFileIo.cpp.
Referenced by Util::ParamComposite::setParent().

12.162.3.3 clearCommunicator() `void Util::MpiFileIo::clearCommunicator ( )`

Clear (nullify) the communicator.
Definition at line 46 of file MpiFileIo.cpp.

12.162.3.4 hasIoCommunicator() `bool Util::MpiFileIo::hasIoCommunicator ( ) const [inline]`

Does this object have an associated MPI communicator?
Definition at line 99 of file MpiFileIo.h.
Referenced by Util::Parameter::load(), Util::Factory< Data >::loadObject(), Util::ParamComposite::loadOptional(), Util↩::Factory< Data >::readObject(), Util::Begin::readParam(), Util::Parameter::readParam(), and Util::ParamComposite↩::setParent().

12.162.3.5 ioCommunicator() `MPI::Intracomm & Util::MpiFileIo::ioCommunicator ( ) const [inline]`

Get the MPI communicator by reference.
Definition at line 105 of file MpiFileIo.h.
Referenced by Util::Parameter::load(), Util::Factory< Data >::loadObject(), Util::ParamComposite::loadOptional(), Util↩::Factory< Data >::readObject(), Util::Begin::readParam(), Util::Parameter::readParam(), and Util::ParamComposite↩::setParent().
The documentation for this class was generated from the following files:

- MpiFileIo.h
- MpiFileIo.cpp

12.163 Util::MpiLoader< IArchive > Class Template Reference

Provides methods for MPI-aware loading of data from input archive.
`#include <MpiLoader.h>`

Public Member Functions

- MpiLoader (MpiFileIo &mpiFileIo, IArchive &archive)

    Constructor.
- template<typename Data >
  void load (Data &value)

    Load and broadcast a single Data value.
- template<typename Data >
  void load (Data ∗value, int n)

    Load and broadcast a C array.
- template<typename Data >
  void load (DArray< Data > &array, int n)

    Load and broadcast a DArray < Data > container.
- template<typename Data , int N>
  void load (FArray< Data, N > &array)

    Load and broadcast an FArray <Data , N > object.
- template<typename Data >
  void load (Data ∗value, int m, int n, int np)

    Load and broadcast a 2D CArray of Data objects.
- template<typename Data >
  void load (DMatrix< Data > &matrix, int m, int n)

    Load and broadcast a DMatrix<Data> object.

12.163.1 Detailed Description

template<class IArchive>
class Util::MpiLoader< IArchive >

Provides methods for MPI-aware loading of data from input archive.
Each MpiLoader is associated with an IArchive input archive, and with a MpiFileIo, which are passed as arguments to the constructor. The MpiFileIo argument is often a ParamComposite, which is derived from MpiFileIo.
The "load" function templates all load data from the archive and (if appropriate) broadcast data among processors. If MPI is not enabled (i.e., if UTIL_MPI is not defined), then the data is simply loaded from the archive. If MPI is enabled and a parameter communicator is set, data is loaded from the archive by the ioProcessor and then broadcast to all other processors in the IO communicator. If MPI is enabled but no parameter communicator is set, every processor loads data independently.
Definition at line 43 of file MpiLoader.h.

12.163.2 Constructor & Destructor Documentation

12.163.2.1 MpiLoader() `template<typename IArchive >`

```
Util::MpiLoader< IArchive >::MpiLoader (
            MpiFileIo & mpiFileIo,
            IArchive & archive )
```

Constructor.

Parameters

mpi←FileIo	associated MpiFileIo object
archive	input archive from which data will be loaded

Definition at line 127 of file MpiLoader.h.

12.163.3 Member Function Documentation

12.163.3.1 load() [1/6]

```
template<typename IArchive >
template<typename Data >
void Util::MpiLoader< IArchive >::load (
            Data & value )
```

Load and broadcast a single Data value.

Parameters

value	reference to a Data

Definition at line 137 of file MpiLoader.h.
Referenced by Util::FileMaster::loadParameters().

12.163.3.2 load() [2/6]

```
template<typename IArchive >
template<typename Data >
void Util::MpiLoader< IArchive >::load (
            Data * value,
            int n )
```

Load and broadcast a C array.

Parameters

value	pointer to array
n	number of elements

Definition at line 154 of file MpiLoader.h.

12.163.3.3 load() [3/6]

```
template<typename IArchive >
template<typename Data >
void Util::MpiLoader< IArchive >::load (
            DArray< Data > & array,
            int n )
```

Load and broadcast a DArray < Data > container.

Parameters

array	DArray object
n	number of elements

Definition at line 174 of file MpiLoader.h.
References UTIL_THROW.

12.163.3.4 load() [4/6]

```
template<typename IArchive >
```

```
template<typename Data , int N>
void Util::MpiLoader< IArchive >::load (
            FArray< Data, N > & array )
```

Load and broadcast an FArray <Data , N > object.

Parameters

array	FArray object to be loaded

Definition at line 194 of file MpiLoader.h.

12.163.3.5 load() [5/6]

```
template<typename IArchive >
template<typename Data >
void Util::MpiLoader< IArchive >::load (
            Data * value,
            int m,
            int n,
            int np )
```

Load and broadcast a 2D CArray of Data objects.

Loads m rows of n elements into array declared as Data array[][np].

Parameters

value	pointer to first element or row in array
m	logical number of rows (1st dimension)
n	logical number of columns (2nd dimension)
np	physcial number of columns (elements allocated per row)

Definition at line 214 of file MpiLoader.h.

12.163.3.6 load() [6/6]

```
template<typename IArchive >
template<typename Data >
void Util::MpiLoader< IArchive >::load (
            DMatrix< Data > & matrix,
            int m,
            int n )
```

Load and broadcast a DMatrix<Data> object.

Parameters

matrix	DMatrix object
m	number of rows (1st dimension)
n	number of columns (2nd dimension)

Definition at line 237 of file MpiLoader.h.

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

- MpiLoader.h

12.164 Util::MpiLogger Class Reference

Allows information from every processor in a communicator, to be output in rank sequence.

`#include <MpiLogger.h>`

Public Member Functions

- MpiLogger (MPI::Intracomm &comm=MPI::COMM_WORLD)

  Constructor.
- void begin ()

  Begin logging block.
- void end ()

  End logging block.

12.164.1 Detailed Description

Allows information from every processor in a communicator, to be output in rank sequence.

The begin() method for processor of rank > 0 waits for receipt of a message from processor rank - 1. The end() method sends a message to processor rank + 1.

Usage:

```
MpiLogger logger;
logger.begin();
std::cout « "Print from processor " « MPI::COMM_WORLD.Get_rank() « std::endl;
logger.endl();
```

Definition at line 37 of file MpiLogger.h.

12.164.2 Constructor & Destructor Documentation

12.164.2.1 MpiLogger() `Util::MpiLogger::MpiLogger ( MPI::Intracomm & comm = MPI::COMM_WORLD )`

Constructor.

Definition at line 18 of file MpiLogger.cpp.

12.164.3 Member Function Documentation

12.164.3.1 begin() `void Util::MpiLogger::begin ( )`

Begin logging block.

Definition at line 26 of file MpiLogger.cpp.

12.164.3.2 end() `void Util::MpiLogger::end ( )`

End logging block.

Definition at line 42 of file MpiLogger.cpp.

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

- MpiLogger.h
- MpiLogger.cpp

12.165 Util::MpiStructBuilder Class Reference

A MpiStructBuilder objects is used to create an MPI Struct datatype.

`#include <MpiStructBuilder.h>`

Public Member Functions

- MpiStructBuilder ()

  Default constructor.
- void setBase (void ∗objectAddress)

  Set address of an class instance.
- void addMember (void ∗memberAddress, MPI::Datatype type, int count=1)

  Add a new member variable to the type map.
- void commit (MPI::Datatype &newType)

  Build and commit a user-defined MPI Struct datatype.

12.165.1 Detailed Description

A MpiStructBuilder objects is used to create an MPI Struct datatype.

This class provides methods to simplify construction of an MPI data type that can stores instances of a C struct or C++ class.

As an example, consider the creation of an MPI datatype MyClassMpi for class MyClass, with a class definition:

```
class MyClass
{
   double x[3];
   int    i, j;
}
```

The code required to build and commit the MPI datatype MyClassMpi is:

```
MyClass          object;
MPI::Datatype  MyClassMpi;
MpiStructBuilder builder;
builder.setBase(&object)
builder.addMember(&object.x, MPI::DOUBLE, 3);
builder.addMember(&object.i, MPI::INT, 1);
builder.addMember(&object.j, MPI::INT, 1);
builder.commit(&MyClassMpi);
```

The setBase and addMember classes require addresses of an instance of the class and of its members, respectively. These addresses must all refer to same instance. The commit method calculates the offset of each member by subtracting the address of the object from the address of each of its members.

Definition at line 54 of file MpiStructBuilder.h.

12.165.2 Constructor & Destructor Documentation

12.165.2.1 MpiStructBuilder() `Util::MpiStructBuilder::MpiStructBuilder ( )`

Default constructor.

Definition at line 10 of file MpiStructBuilder.cpp.

12.165.3 Member Function Documentation

12.165.3.1 setBase() `void Util::MpiStructBuilder::setBase ( void * objectAddress )`

Set address of an class instance.

Definition at line 18 of file MpiStructBuilder.cpp.

Referenced by Util::Pair< DPropagator >::commitMpiType(), Util::FArray< DPropagator, 2 >::commitMpiType(), Util↩ ::Tensor::commitMpiType(), Util::Vector::commitMpiType(), and Util::IntVector::commitMpiType().

12.165.3.2 addMember() `void Util::MpiStructBuilder::addMember ( void * memberAddress, MPI::Datatype type, int count = 1 )`

Add a new member variable to the type map.

This method must be called once for each member. The address parameter must be a pointer to a member variable of the object whose base address is passed to setBase().

The count parameter is required only for array members: the default value of count=1 may be used for scalar members.

Parameters

memberAddress	displacement of variable, in bytes.
type	data type (MPI::INT, MPI::DOUBLE, etc.)
count	number of contiguous variables (array count)

Definition at line 26 of file MpiStructBuilder.cpp.

Referenced by Util::Pair< DPropagator >::commitMpiType(), Util::FArray< DPropagator, 2 >::commitMpiType(), Util::Tensor::commitMpiType(), Util::Vector::commitMpiType(), and Util::IntVector::commitMpiType().

12.165.3.3 commit() `void Util::MpiStructBuilder::commit ( MPI::Datatype & newType )`

Build and commit a user-defined MPI Struct datatype.

The setBase() method must be called once and the addMember() method must be called once per member before calling this method.

Parameters

newType	new MPI datatype (on output).

Definition at line 39 of file MpiStructBuilder.cpp.

Referenced by Util::Pair< DPropagator >::commitMpiType(), Util::FArray< DPropagator, 2 >::commitMpiType(), Util::Tensor::commitMpiType(), Util::Vector::commitMpiType(), and Util::IntVector::commitMpiType().

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

- MpiStructBuilder.h
- MpiStructBuilder.cpp

12.166 Util::MpiTraits< T > Class Template Reference

Default MpiTraits class.

`#include <MpiTraits.h>`

Inheritance diagram for Util::MpiTraits< T >:

classUtil_1_1MpiTraits-eps-converted-to.pdf

Static Public Attributes

- static const bool hasType

Is the MPI type initialized?

- static const MPI::Datatype type

  MPI Datatype (dummy - unused)

Additional Inherited Members

12.166.1 Detailed Description

template<typename T>
class Util::MpiTraits< T >

Default MpiTraits class.
Each explicit specialization of MpiTraits has a public static const member named type. This is the MPI data type associated with the C++ template type parameter.
Definition at line 39 of file MpiTraits.h.

12.166.2 Member Data Documentation

12.166.2.1 hasType `template<typename T >`

`const bool Util::MpiTraitsNoType::hasType [static]`
Is the MPI type initialized?
Definition at line 28 of file MpiTraits.h.

12.166.2.2 type `template<typename T >`

`const MPI::Datatype Util::MpiTraitsNoType::type [static]`
MPI Datatype (dummy - unused)
Definition at line 26 of file MpiTraits.h.
The documentation for this class was generated from the following file:

- MpiTraits.h

12.167 Util::MpiTraits< bool > Class Reference

MpiTraits<bool> explicit specialization.
`#include <MpiTraits.h>`

Static Public Attributes

- static const MPI::Datatype type = MPI::BYTE

  MPI Datatype.
- static const bool hasType = false

  Is the MPI type initialized?

12.167.1 Detailed Description

MpiTraits<bool> explicit specialization.
Definition at line 171 of file MpiTraits.h.

12.167.2 Member Data Documentation

12.167.2.1 type `const MPI::Datatype Util::MpiTraits< bool >::type = MPI::BYTE [static]`

MPI Datatype.
Definition at line 174 of file MpiTraits.h.

12.167.2.2 hasType `const bool Util::MpiTraits< bool >::hasType = false [static]`

Is the MPI type initialized?
Definition at line 175 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.168 Util::MpiTraits< char > Class Reference

MpiTraits<char> explicit specialization.

```
#include <MpiTraits.h>
```

Static Public Attributes

- static const MPI::Datatype type = MPI::CHAR

    MPI Datatype.
- static const bool hasType = true

    Is the MPI type initialized?

12.168.1 Detailed Description

MpiTraits<char> explicit specialization.
Definition at line 50 of file MpiTraits.h.

12.168.2 Member Data Documentation

12.168.2.1 type `const MPI::Datatype Util::MpiTraits< char >::type = MPI::CHAR [static]`

MPI Datatype.
Definition at line 53 of file MpiTraits.h.

12.168.2.2 hasType `const bool Util::MpiTraits< char >::hasType = true [static]`

Is the MPI type initialized?
Definition at line 54 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.169 Util::MpiTraits< double > Class Reference

MpiTraits<double> explicit specialization.

```
#include <MpiTraits.h>
```

Static Public Attributes

- static const MPI::Datatype type = MPI::DOUBLE

  MPI Datatype.
- static const bool hasType = true

  Is the MPI type initialized?

12.169.1 Detailed Description

MpiTraits<double> explicit specialization.
Definition at line 149 of file MpiTraits.h.

12.169.2 Member Data Documentation

12.169.2.1 type `const MPI::Datatype Util::MpiTraits< double >::type = MPI::DOUBLE [static]`

MPI Datatype.
Definition at line 152 of file MpiTraits.h.

12.169.2.2 hasType `const bool Util::MpiTraits< double >::hasType = true [static]`

Is the MPI type initialized?
Definition at line 153 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.170 Util::MpiTraits< float > Class Reference

MpiTraits<float> explicit specialization.
`#include <MpiTraits.h>`

Static Public Attributes

- static const MPI::Datatype type = MPI::FLOAT

  MPI Datatype.
- static const bool hasType = true

  Is the MPI type initialized?

12.170.1 Detailed Description

MpiTraits<float> explicit specialization.
Definition at line 138 of file MpiTraits.h.

12.170.2 Member Data Documentation

12.170.2.1 type `const MPI::Datatype Util::MpiTraits< float >::type = MPI::FLOAT [static]`

MPI Datatype.
Definition at line 141 of file MpiTraits.h.

12.170.2.2 hasType `const bool Util::MpiTraits< float >::hasType = true [static]`

Is the MPI type initialized?
Definition at line 142 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.171 Util::MpiTraits< int > Class Reference

MpiTraits<int> explicit specialization.
`#include <MpiTraits.h>`

Static Public Attributes

- static const MPI::Datatype type = MPI::INT

  MPI Datatype.
- static const bool hasType = true

  Is the MPI type initialized?

12.171.1 Detailed Description

MpiTraits<int> explicit specialization.
Definition at line 83 of file MpiTraits.h.

12.171.2 Member Data Documentation

12.171.2.1 type `const MPI::Datatype Util::MpiTraits< int >::type = MPI::INT [static]`

MPI Datatype.
Definition at line 86 of file MpiTraits.h.

12.171.2.2 hasType `const bool Util::MpiTraits< int >::hasType = true [static]`

Is the MPI type initialized?
Definition at line 87 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.172 Util::MpiTraits< IntVector > Class Reference

Explicit specialization MpiTraits<IntVector>.
`#include <IntVector.h>`

Static Public Attributes

- static MPI::Datatype type = MPI::BYTE

  MPI Datatype.
- static bool hasType = false

  Is the MPI type initialized?

12.172.1 Detailed Description

Explicit specialization MpiTraits<IntVector>.
Definition at line 443 of file IntVector.h.

12.172.2 Member Data Documentation

12.172.2.1 type `MPI::Datatype Util::MpiTraits< IntVector >::type = MPI::BYTE [static]`

MPI Datatype.
Definition at line 446 of file IntVector.h.

12.172.2.2 hasType `bool Util::MpiTraits< IntVector >::hasType = false [static]`

Is the MPI type initialized?
Definition at line 447 of file IntVector.h.
The documentation for this class was generated from the following files:

- IntVector.h
- IntVector.cpp

12.173 Util::MpiTraits< long > Class Reference

MpiTraits<long> explicit specialization.
`#include <MpiTraits.h>`

Static Public Attributes

- static const MPI::Datatype type = MPI::LONG

  MPI Datatype.
- static const bool hasType = true

  Is the MPI type initialized?

12.173.1 Detailed Description

MpiTraits<long> explicit specialization.
Definition at line 94 of file MpiTraits.h.

12.173.2 Member Data Documentation

12.173.2.1 type `const MPI::Datatype Util::MpiTraits< long >::type = MPI::LONG [static]`

MPI Datatype.
Definition at line 97 of file MpiTraits.h.

12.173.2.2 hasType `const bool Util::MpiTraits< long >::hasType = true [static]`

Is the MPI type initialized?
Definition at line 98 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.174 Util::MpiTraits< long double > Class Reference

MpiTraits< long double > explicit specialization.

`#include <MpiTraits.h>`

Static Public Attributes

- static const MPI::Datatype type = MPI::LONG_DOUBLE

  MPI Datatype.
- static const bool hasType = true

  Is the MPI type initialized?

12.174.1 Detailed Description

MpiTraits< long double > explicit specialization.
Definition at line 160 of file MpiTraits.h.

12.174.2 Member Data Documentation

12.174.2.1 type `const MPI::Datatype Util::MpiTraits< long double >::type = MPI::LONG_DOUBLE [static]`

MPI Datatype.
Definition at line 163 of file MpiTraits.h.

12.174.2.2 hasType `const bool Util::MpiTraits< long double >::hasType = true [static]`

Is the MPI type initialized?
Definition at line 164 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.175 Util::MpiTraits< Rational > Class Reference

Explicit specialization MpiTraits< Rational >.

`#include <Rational.h>`

Static Public Attributes

- static MPI::Datatype type

  MPI Datatype.
- static bool hasType

  Is the MPI type initialized?

12.175.1 Detailed Description

Explicit specialization MpiTraits< Rational >.
Definition at line 759 of file Rational.h.

12.175.2 Member Data Documentation

12.175.2.1 type `MPI::Datatype Util::MpiTraits< Rational >::type [static]`

MPI Datatype.

Definition at line 762 of file Rational.h.

12.175.2.2 hasType `bool Util::MpiTraits< Rational >::hasType [static]`

Is the MPI type initialized?

Definition at line 763 of file Rational.h.

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

- Rational.h

12.176 Util::MpiTraits< short > Class Reference

MpiTraits<short> explicit specialization.

`#include <MpiTraits.h>`

Static Public Attributes

- static const MPI::Datatype type = MPI::SHORT_INT

  MPI Datatype.
- static const bool hasType = true

  Is the MPI type initialized?

12.176.1 Detailed Description

MpiTraits<short> explicit specialization.

Definition at line 72 of file MpiTraits.h.

12.176.2 Member Data Documentation

12.176.2.1 type `const MPI::Datatype Util::MpiTraits< short >::type = MPI::SHORT_INT [static]`

MPI Datatype.

Definition at line 75 of file MpiTraits.h.

12.176.2.2 hasType `const bool Util::MpiTraits< short >::hasType = true [static]`

Is the MPI type initialized?

Definition at line 76 of file MpiTraits.h.

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

- MpiTraits.h
- MpiTraits.cpp

12.177 Util::MpiTraits< Tensor > Class Reference

Explicit specialization MpiTraits<Tensor>.

`#include <Tensor.h>`

Static Public Attributes

- static MPI::Datatype type = MPI::BYTE
    MPI Datatype.
- static bool hasType = false
    Is the MPI type initialized?

12.177.1 Detailed Description

Explicit specialization MpiTraits<Tensor>.
Definition at line 385 of file Tensor.h.

12.177.2 Member Data Documentation

12.177.2.1 type `MPI::Datatype Util::MpiTraits< Tensor >::type = MPI::BYTE [static]`

MPI Datatype.
Definition at line 388 of file Tensor.h.

12.177.2.2 hasType `bool Util::MpiTraits< Tensor >::hasType = false [static]`

Is the MPI type initialized?
Definition at line 389 of file Tensor.h.
The documentation for this class was generated from the following files:

- Tensor.h
- Tensor.cpp

12.178 Util::MpiTraits< unsigned char > Class Reference

MpiTraits<unsigned char> explicit specialization.
`#include <MpiTraits.h>`

Static Public Attributes

- static const MPI::Datatype type = MPI::UNSIGNED_CHAR
    MPI Datatype.
- static const bool hasType = true
    Is the MPI type initialized?

12.178.1 Detailed Description

MpiTraits<unsigned char> explicit specialization.
Definition at line 61 of file MpiTraits.h.

12.178.2 Member Data Documentation

12.178.2.1 type `const MPI::Datatype Util::MpiTraits< unsigned char >::type = MPI::UNSIGNED_CHAR [static]`

MPI Datatype.
Definition at line 64 of file MpiTraits.h.

12.178.2.2 hasType `const bool Util::MpiTraits< unsigned char >::hasType = true [static]`

Is the MPI type initialized?
Definition at line 65 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.179 Util::MpiTraits< unsigned int > Class Reference

MpiTraits<unsigned int> explicit specialization.
`#include <MpiTraits.h>`

Static Public Attributes

- static const MPI::Datatype type = MPI::UNSIGNED

  MPI Datatype.
- static const bool hasType = true

  Is the MPI type initialized?

12.179.1 Detailed Description

MpiTraits<unsigned int> explicit specialization.
Definition at line 116 of file MpiTraits.h.

12.179.2 Member Data Documentation

12.179.2.1 type `const MPI::Datatype Util::MpiTraits< unsigned int >::type = MPI::UNSIGNED [static]`

MPI Datatype.
Definition at line 119 of file MpiTraits.h.

12.179.2.2 hasType `const bool Util::MpiTraits< unsigned int >::hasType = true [static]`

Is the MPI type initialized?
Definition at line 120 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.180 Util::MpiTraits< unsigned long > Class Reference

MpiTraits<unsigned long> explicit specialization.
`#include <MpiTraits.h>`

Static Public Attributes

- static const MPI::Datatype type = MPI::UNSIGNED_LONG

  MPI Datatype.
- static const bool hasType = true

  Is the MPI type initialized?

12.180.1 Detailed Description

MpiTraits<unsigned long> explicit specialization.
Definition at line 127 of file MpiTraits.h.

12.180.2 Member Data Documentation

12.180.2.1 type `const MPI::Datatype Util::MpiTraits< unsigned long >::type = MPI::UNSIGNED_LONG [static]`

MPI Datatype.
Definition at line 130 of file MpiTraits.h.

12.180.2.2 hasType `const bool Util::MpiTraits< unsigned long >::hasType = true [static]`

Is the MPI type initialized?
Definition at line 131 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.181 Util::MpiTraits< unsigned short > Class Reference

MpiTraits<unsigned short> explicit specialization.

```
#include <MpiTraits.h>
```

Static Public Attributes

- static const MPI::Datatype type = MPI::UNSIGNED_SHORT

  MPI Datatype.
- static const bool hasType = true

  Is the MPI type initialized?

12.181.1 Detailed Description

MpiTraits<unsigned short> explicit specialization.
Definition at line 105 of file MpiTraits.h.

12.181.2 Member Data Documentation

12.181.2.1 type `const MPI::Datatype Util::MpiTraits< unsigned short >::type = MPI::UNSIGNED_SHORT [static]`

MPI Datatype.
Definition at line 108 of file MpiTraits.h.

12.181.2.2 hasType `const bool Util::MpiTraits< unsigned short >::hasType = true [static]`

Is the MPI type initialized?
Definition at line 109 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.182 Util::MpiTraits< Vector > Class Reference

Explicit specialization MpiTraits<Vector>.
`#include <Vector.h>`

Static Public Attributes

- static MPI::Datatype type = MPI::BYTE

  MPI Datatype.
- static bool hasType = false

  Is the MPI type initialized?

12.182.1 Detailed Description

Explicit specialization MpiTraits<Vector>.
Definition at line 449 of file Vector.h.

12.182.2 Member Data Documentation

12.182.2.1 type `MPI::Datatype Util::MpiTraits< Vector >::type = MPI::BYTE [static]`

MPI Datatype.
Initialize MPI Datatype.
Definition at line 452 of file Vector.h.

12.182.2.2 hasType `bool Util::MpiTraits< Vector >::hasType = false [static]`

Is the MPI type initialized?
Definition at line 453 of file Vector.h.
The documentation for this class was generated from the following files:

- Vector.h
- Vector.cpp

12.183 Util::MpiTraitsNoType Class Reference

Base class for MpiTraits with no type.
`#include <MpiTraits.h>`
Inheritance diagram for Util::MpiTraitsNoType:

classUtil_1_1MpiTraitsNoType-eps-converted-to.pdf

Static Protected Attributes

- static const MPI::Datatype type = MPI::CHAR

    MPI Datatype (dummy - unused)
- static const bool hasType = false

    Is the MPI type initialized?

12.183.1 Detailed Description

Base class for MpiTraits with no type.
Definition at line 22 of file MpiTraits.h.

12.183.2 Member Data Documentation

12.183.2.1 type `const MPI::Datatype Util::MpiTraitsNoType::type = MPI::CHAR [static], [protected]`

MPI Datatype (dummy - unused)
Definition at line 26 of file MpiTraits.h.

12.183.2.2 hasType `const bool Util::MpiTraitsNoType::hasType = false [static], [protected]`

Is the MPI type initialized?
Definition at line 28 of file MpiTraits.h.
The documentation for this class was generated from the following files:

- MpiTraits.h
- MpiTraits.cpp

12.184 Util::MTRand Class Reference

Generates double floating point numbers in the half-open interval [0, 1)
`#include <mtrand.h>`
Inheritance diagram for Util::MTRand:

classUtil_1_1MTRand-eps-converted-to.pdf

Additional Inherited Members

12.184.1 Detailed Description

Generates double floating point numbers in the half-open interval [0, 1)
Definition at line 202 of file mtrand.h.
The documentation for this class was generated from the following file:

- mtrand.h

12.185 Util::MTRand53 Class Reference

generates 53 bit resolution doubles in the half-open interval [0, 1)

`#include <mtrand.h>`

Inheritance diagram for Util::MTRand53:

classUtil_1_1MTRand53-eps-converted-to.pdf

Additional Inherited Members

12.185.1 Detailed Description

generates 53 bit resolution doubles in the half-open interval [0, 1)

Definition at line 296 of file mtrand.h.

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

- mtrand.h

12.186 Util::MTRand_closed Class Reference

Generates double floating point numbers in the closed interval [0, 1].

`#include <mtrand.h>`

Inheritance diagram for Util::MTRand_closed:

classUtil_1_1MTRand__closed-eps-converted-to.pdf

Additional Inherited Members

12.186.1 Detailed Description

Generates double floating point numbers in the closed interval [0, 1].

Definition at line 230 of file mtrand.h.

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

- mtrand.h

12.187 Util::MTRand_int32 Class Reference

Mersenne Twister random number generator engine.

`#include <mtrand.h>`

Inheritance diagram for Util::MTRand_int32:

classUtil_1_1MTRand__int32-eps-converted-to.pdf

Public Member Functions

- MTRand_int32 ()

  Default constructor.
- MTRand_int32 (unsigned long s)

  Constructor with 32 bit int as seed.
- MTRand_int32 (const unsigned long ∗array, int size)

  Constructor with array of size 32 bit ints as seed.
- MTRand_int32 (const MTRand_int32 &)

  Copy constructor.
- void operator= (const MTRand_int32 &)

  Assignment.
- virtual ∼MTRand_int32 ()

  Destructor.
- void seed (unsigned long)

  Seed with 32 bit integer.
- void seed (const unsigned long ∗, int size)

  Seed with array.
- unsigned long operator() ()

  Overload operator() to make this a generator (functor)
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize to/from an archive.

12.187.1 Detailed Description

Mersenne Twister random number generator engine.
Generates unsigned long integers randomly distributed over the range 0 < i < 2^{32} - 1.
Definition at line 70 of file mtrand.h.

12.187.2 Constructor & Destructor Documentation

12.187.2.1 MTRand_int32() [1/4] `Util::MTRand_int32::MTRand_int32 ( )`

Default constructor.
Definition at line 13 of file mtrand.cpp.
References seed().

12.187.2.2 MTRand_int32() [2/4] `Util::MTRand_int32::MTRand_int32 ( unsigned long s )`

Constructor with 32 bit int as seed.
Definition at line 25 of file mtrand.cpp.
References seed().

12.187.2.3 MTRand_int32() [3/4]

Util::MTRand_int32::MTRand_int32 (
const unsigned long * *array*,
int *size*)

Constructor with array of size 32 bit ints as seed.

Definition at line 37 of file mtrand.cpp.

References seed().

12.187.2.4 MTRand_int32() [4/4]

Util::MTRand_int32::MTRand_int32 (
const MTRand_int32 & *other*)

Copy constructor.

Definition at line 49 of file mtrand.cpp.

12.187.2.5 ∼MTRand_int32()

virtual Util::MTRand_int32::∼MTRand_int32 ( ) [inline], [virtual]

Destructor.

2007-02-11: made the destructor virtual; Thanks "double more" for pointing this out

Definition at line 105 of file mtrand.h.

12.187.3 Member Function Documentation

12.187.3.1 operator=()

void Util::MTRand_int32::operator= (
const MTRand_int32 & *other*)

Assignment.

Definition at line 61 of file mtrand.cpp.

12.187.3.2 seed() [1/2]

void Util::MTRand_int32::seed (
unsigned long *s*)

Seed with 32 bit integer.

Definition at line 86 of file mtrand.cpp.

Referenced by MTRand_int32(), and seed().

12.187.3.3 seed() [2/2]

void Util::MTRand_int32::seed (
const unsigned long * *array*,
int *size*)

Seed with array.

Definition at line 98 of file mtrand.cpp.

References seed().

12.187.3.4 operator()()

unsigned long Util::MTRand_int32::operator() ( ) [inline]

Overload operator() to make this a generator (functor)

Generate 32 bit random int, public method.

Definition at line 197 of file mtrand.h.

12.187.3.5 serialize()

```
template<class Archive >
void Util::MTRand_int32::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize to/from an archive.
Definition at line 185 of file mtrand.h.
The documentation for this class was generated from the following files:

- mtrand.h
- mtrand.cpp

12.188 Util::MTRand_open Class Reference

Generates double floating point numbers in the open interval (0, 1).

```
#include <mtrand.h>
```

Inheritance diagram for Util::MTRand_open:

classUtil_1_1MTRand__open-eps-converted-to.pdf

Additional Inherited Members

12.188.1 Detailed Description

Generates double floating point numbers in the open interval (0, 1).
Definition at line 263 of file mtrand.h.
The documentation for this class was generated from the following file:

- mtrand.h

12.189 Util::Node< Data > Class Template Reference

Linked List Node, class template.

```
#include <Node.h>
```

Public Member Functions

- Node ()

    Default constructor.
- Node (const Node< Data > &other)

    Copy constructor.
- Node< Data > ∗ next () const

    Get the next pointer.
- Node< Data > ∗ prev () const

    Get the previous pointer.
- const Data & data () const

    Get a const reference to the associated Data.
- Data & data ()

    Get a reference to the associated Data object.
- List< Data > & list () const

Get a reference to the List.

- void setNext (Node< Data > *next)

  Set pointer to the next Node.

- void setPrev (Node< Data > *prev)

  Set pointer to the previous Node.

- void setList (List< Data > &list)

  Set the list.

- void setList (List< Data > *list)

  Set the list.

- void attachNext (Node< Data > &other)

  Set pointers connecting the other node after this node.

- void attachPrev (Node< Data > &other)

  Set pointers connecting the other node before this node.

- void clear ()

  Nullify previous and next pointers, and nullify the list pointer.

12.189.1 Detailed Description

template<typename Data>
class Util::Node< Data >

Linked List Node, class template.
Definition at line 22 of file Node.h.

12.189.2 Constructor & Destructor Documentation

12.189.2.1 Node() [1/2] `template<typename Data >`

`Util::Node< Data >::Node ( ) [inline]`

Default constructor.
Definition at line 30 of file Node.h.

12.189.2.2 Node() [2/2] `template<typename Data >`

```
Util::Node< Data >::Node (
            const Node< Data > & other ) [inline]
```

Copy constructor.
Definition at line 40 of file Node.h.

12.189.3 Member Function Documentation

12.189.3.1 next() `template<typename Data >`

`Node<Data>* Util::Node< Data >::next ( ) const [inline]`

Get the next pointer.

Returns

pointer to next Node in List.

Definition at line 52 of file Node.h.
Referenced by Util::List< Data >::insert(), Util::List< Data >::insertNext(), Util::List< Data >::isValid(), Util::List< Data >::popFront(), Util::List< Data >::remove(), and Util::Node< Data >::setNext().

12.189.3.2 prev() `template<typename Data >`

`Node<Data>* Util::Node< Data >::prev ( ) const [inline]`

Get the previous pointer.

Returns

pointer to previous Node in List.

Definition at line 60 of file Node.h.
Referenced by Util::List< Data >::insert(), Util::List< Data >::insertPrev(), Util::List< Data >::isValid(), Util::List< Data >::popBack(), Util::List< Data >::remove(), and Util::Node< Data >::setPrev().

12.189.3.3 data() [1/2] `template<typename Data >`

`const Data& Util::Node< Data >::data ( ) const [inline]`

Get a const reference to the associated Data.

Returns

Data object associated with this Node.

Definition at line 68 of file Node.h.

12.189.3.4 data() [2/2] `template<typename Data >`

`Data& Util::Node< Data >::data ( ) [inline]`

Get a reference to the associated Data object.

Returns

Data object associated with this Node.

Definition at line 76 of file Node.h.

12.189.3.5 list() `template<typename Data >`

`List<Data>& Util::Node< Data >::list ( ) const [inline]`

Get a reference to the List.

Returns

Reference to the list to which this Node belongs.

Definition at line 84 of file Node.h.
Referenced by Util::List< Data >::insert(), Util::List< Data >::isValid(), Util::List< Data >::remove(), and Util::Node< Data >::setList().

12.189.3.6 setNext() `template<typename Data >`

```
void Util::Node< Data >::setNext (
            Node< Data > * next ) [inline]
```

Set pointer to the next Node.

Parameters

next	pointer to next Node

Definition at line 92 of file Node.h.
References Util::Node< Data >::next().
Referenced by Util::List< Data >::insert(), Util::List< Data >::insertNext(), and Util::List< Data >::pushBack().

12.189.3.7 setPrev() `template<typename Data >`

```
void Util::Node< Data >::setPrev (
            Node< Data > * prev ) [inline]
```

Set pointer to the previous Node.

Parameters

prev	pointer to previous Node

Definition at line 100 of file Node.h.
References Util::Node< Data >::prev().
Referenced by Util::List< Data >::insert(), Util::List< Data >::insertPrev(), and Util::List< Data >::pushFront().

12.189.3.8 setList() [1/2] `template<typename Data >`

```
void Util::Node< Data >::setList (
            List< Data > & list ) [inline]
```

Set the list.

Parameters

list	associated List object

Definition at line 108 of file Node.h.
References Util::Node< Data >::list().
Referenced by Util::List< Data >::insert(), Util::List< Data >::pushBack(), and Util::List< Data >::pushFront().

12.189.3.9 setList() [2/2] `template<typename Data >`

```
void Util::Node< Data >::setList (
            List< Data > * list ) [inline]
```

Set the list.

Parameters

list	pointer to an associated List object

Definition at line 116 of file Node.h.
References Util::Node< Data >::list().

12.189.3.10 attachNext() `template<typename Data >`

```
void Util::Node< Data >::attachNext (
```

```
            Node< Data > & other )   [inline]
```

Set pointers connecting the other node after this node.

This method sets the next pointer of this node to other and the previous node of other to this. It also also sets the list of the other node to this list.

It does not reset the next pointer of the other node, and so does not finish splicing the other node into the middle of this list. The next pointer of the other node is left unchanged.

Parameters

other	a Node to connect to this one.

Definition at line 132 of file Node.h.

Referenced by Util::List< Data >::insert(), Util::List< Data >::insertNext(), and Util::List< Data >::pushBack().

12.189.3.11 attachPrev() `template<typename Data >`

```
void Util::Node< Data >::attachPrev (
            Node< Data > & other )   [inline]
```

Set pointers connecting the other node before this node.

This method sets the previous pointer of this node to other and the next node of other to this. It also also sets the list of the other node to this list.

It does not reset the previous pointer of the other node, and so does not finish splicing the other node into the middle of this list. The previous pointer of the other node is left unchanged.

Parameters

other	a Node to connect to this one.

Definition at line 152 of file Node.h.

Referenced by Util::List< Data >::insertPrev(), and Util::List< Data >::pushFront().

12.189.3.12 clear() `template<typename Data >`

```
void Util::Node< Data >::clear ( )   [inline]
```

Nullify previous and next pointers, and nullify the list pointer.

This method disconnects the Node from any List, but does not modify the datum_.

Definition at line 165 of file Node.h.

Referenced by Util::List< Data >::popBack(), Util::List< Data >::popFront(), and Util::List< Data >::remove().

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

- Node.h

12.190 Util::Notifier< Event > Class Template Reference

Abstract template for a notifier (or subject) in the Observer design pattern.

```
#include <Notifier.h>
```

Public Member Functions

- void registerObserver (Observer< Event > &observer)

    Register an observer.

- void removeObserver (Observer< Event > &observer)

    Remove an analyzer observer from the container list.

- void notifyObservers (const Event &event)

  Notify the list of observers about an Event.

12.190.1 Detailed Description

template<typename Event>
class Util::Notifier< Event >

Abstract template for a notifier (or subject) in the Observer design pattern.
In the observer design pattern, a Notifier manages a list of registered Observer objects, and provides a method to notify all observers when some event occurs. A list of observer objects is maintained as a list of Observer pointers. The method Notifier::notifyObservers(Event&) method calls the the update(Event&) method of every Observer in the list.
The typename parameter Event is the type of the object that must be passed to the update() method of each observer. This type can name either a primitive C data type or a class, but must encode whatever information is required for any Observer to respond appropriately when notified.
Definition at line 41 of file Notifier.h.

12.190.2 Member Function Documentation

12.190.2.1 registerObserver() `template<typename Event >`

```
void Util::Notifier< Event >::registerObserver (
            Observer< Event > & observer )
```

Register an observer.

Parameters

observer	observer object

Definition at line 80 of file Notifier.h.

12.190.2.2 removeObserver() `template<typename Event >`

```
void Util::Notifier< Event >::removeObserver (
            Observer< Event > & observer )
```

Remove an analyzer observer from the container list.

Parameters

observer	observer object

Definition at line 89 of file Notifier.h.

12.190.2.3 notifyObservers() `template<typename Event >`

```
void Util::Notifier< Event >::notifyObservers (
            const Event & event )
```

Notify the list of observers about an Event.
Definition at line 98 of file Notifier.h.
The documentation for this class was generated from the following file:

- Notifier.h

12.191 Util::Observer< Event > Class Template Reference

Abstract class template for observer in the observer design pattern.

`#include <Observer.h>`

Public Member Functions

- virtual ~Observer ()

  Destructor.
- virtual void update (const Event &event)=0

  Respond to news of an event.

12.191.1 Detailed Description

template<typename Event>
class Util::Observer< Event >

Abstract class template for observer in the observer design pattern.

An Observer is notified of an event by calling its update method. The template class parameter Event is the type of object that is passed to the update() method as a message about an event.

Definition at line 19 of file Notifier.h.

12.191.2 Constructor & Destructor Documentation

12.191.2.1 ~Observer() `template<typename Event >`

`Util::Observer< Event >::~Observer [virtual]`

Destructor.

Definition at line 47 of file Observer.h.

12.191.3 Member Function Documentation

12.191.3.1 update() `template<typename Event >`

```
virtual void Util::Observer< Event >::update (
            const Event & event ) [pure virtual]
```

Respond to news of an event.

Parameters

event	Object containing information about the event.

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

- Notifier.h
- Observer.h

12.192 Util::OptionalLabel Class Reference

An optional Label string in a file format.

`#include <OptionalLabel.h>`

Inheritance diagram for Util::OptionalLabel:

classUtil_1_1OptionalLabel-eps-converted-to.pdf

Public Member Functions

- OptionalLabel ()

  Default constructor.
- OptionalLabel (std::string string)

  Constructor.
- OptionalLabel (const char ∗string)

  Constructor.
- OptionalLabel (const OptionalLabel &other)

  Copy constructor.
- virtual ∼OptionalLabel ()

  Destructor.

Additional Inherited Members

12.192.1 Detailed Description

An optional Label string in a file format.
A subclass of Label that is always optional.
Definition at line 23 of file OptionalLabel.h.

12.192.2 Constructor & Destructor Documentation

12.192.2.1 OptionalLabel() [1/4] `Util::OptionalLabel::OptionalLabel ( )`

Default constructor.
Definition at line 16 of file OptionalLabel.cpp.

12.192.2.2 OptionalLabel() [2/4] `Util::OptionalLabel::OptionalLabel ( std::string string ) [explicit]`

Constructor.

Parameters

string	label string that precedes value in file format

Definition at line 23 of file OptionalLabel.cpp.

12.192.2.3 OptionalLabel() [3/4] `Util::OptionalLabel::OptionalLabel ( const char * string ) [explicit]`

Constructor.

Parameters

string	label string that precedes value in file format

Definition at line 30 of file OptionalLabel.cpp.

12.192.2.4 OptionalLabel() [4/4] `Util::OptionalLabel::OptionalLabel ( const OptionalLabel & other )`

Copy constructor.

Parameters

other	OptionalLabel being cloned.

Definition at line 37 of file OptionalLabel.cpp.

12.192.2.5 ∼OptionalLabel() `Util::OptionalLabel::∼OptionalLabel ( ) [virtual]`

Destructor.
Definition at line 44 of file OptionalLabel.cpp.
The documentation for this class was generated from the following files:

- OptionalLabel.h
- OptionalLabel.cpp

12.193 Util::Pair< Data > Class Template Reference

An array of exactly 2 objects.
`#include <Pair.h>`
Inheritance diagram for Util::Pair< Data >:

classUtil_1_1Pair-eps-converted-to.pdf

Static Public Member Functions

- static void commitMpiType ()

  Commit associated MPI DataType.

Additional Inherited Members

12.193.1 Detailed Description

template<typename Data>
class Util::Pair< Data >

An array of exactly 2 objects.
Definition at line 23 of file Pair.h.

12.193.2 Member Function Documentation

12.193.2.1 commitMpiType() template<typename Data >

void Util::Pair< Data >::commitMpiType [static]

Commit associated MPI DataType.

Definition at line 69 of file Pair.h.

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

- Pair.h

12.194 Util::ParamComponent Class Reference

Abstract base class for classes that input and output parameters to file.

#include <ParamComponent.h>

Inheritance diagram for Util::ParamComponent:

classUtil_1_1ParamComponent-eps-converted-to.pdf

Public Member Functions

- virtual ~ParamComponent ()

  Destructor.
- virtual void readParam (std::istream &in)=0

  Read parameter(s) from file.
- virtual void writeParam (std::ostream &out)=0

  Read parameter(s) to file.
- virtual void load (Serializable::IArchive &ar)

  Load internal state from an archive.
- virtual void save (Serializable::OArchive &ar)

  Save internal state to an archive.
- virtual void resetParam ()

  Nontrivial implementation provided by ParamComposite subclass.
- void setIndent (const ParamComponent &parent, bool next=true)

  Set indent level.
- std::string indent () const

  Return indent string for this object (string of spaces).

- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize this ParamComponent as a string.

Static Public Member Functions

- static void initStatic ()

  Initialize static echo member to false.
- static void setEcho (bool echo=true)

  Enable or disable echoing for all subclasses of ParamComponent.
- static bool echo ()

  Get echo parameter.

Protected Member Functions

- ParamComponent ()

  Constructor.
- ParamComponent (const ParamComponent &other)

  Copy constructor.

Additional Inherited Members

12.194.1 Detailed Description

Abstract base class for classes that input and output parameters to file.
The readParam method reads a parameter or parameter list from iostream. The writeParam method writes a parameter or parameter list to an ostream. The same io format should be used by write and read methods.
Definition at line 31 of file ParamComponent.h.

12.194.2 Constructor & Destructor Documentation

12.194.2.1 ∼ParamComponent() `Util::ParamComponent::∼ParamComponent ( ) [virtual]`

Destructor.
Definition at line 36 of file ParamComponent.cpp.

12.194.2.2 ParamComponent() `[1/2]` `Util::ParamComponent::ParamComponent ( ) [protected]`

Constructor.
Protected to prevent instantiation of a conceptually abstract class.
On return the indent string is empty. If UTIL_MPI is defined, no communicator is set upon construction.
Definition at line 20 of file ParamComponent.cpp.

12.194.2.3 ParamComponent() `[2/2]` `Util::ParamComponent::ParamComponent ( const ParamComponent & other ) [protected]`

Copy constructor.
Definition at line 28 of file ParamComponent.cpp.

12.194.3 Member Function Documentation

12.194.3.1 readParam() `virtual void Util::ParamComponent::readParam ( std::istream & in ) [pure virtual]`

Read parameter(s) from file.

Parameters

in	input stream

Implemented in Util::ParamComposite, Util::Parameter, Util::Manager< Data >, Util::Begin, Util::Blank, and Util::End.
Referenced by serialize().

12.194.3.2 writeParam() `virtual void Util::ParamComponent::writeParam ( std::ostream & out ) [pure virtual]`

Read parameter(s) to file.

Parameters

out	output stream

Implemented in Util::ParamComposite, Util::CArray2DParam< Type >, Util::ScalarParam< Type >, Util::Begin, Util::DMatrixParam< Type >, Util::DSymmMatrixParam< Type >, Util::Blank, Util::End, Util::FArrayParam< Type, N >, Util::DArrayParam< Type >, and Util::CArrayParam< Type >.
Referenced by Util::Parameter::load(), Util::Parameter::readParam(), and serialize().

12.194.3.3 load() `virtual void Util::ParamComponent::load ( Serializable::IArchive & ar ) [inline], [virtual]`

Load internal state from an archive.
The default implementation is empty. This default is used by the Begin, End, and Blank subclasses.
Implements Util::Serializable.
Reimplemented in Util::ParamComposite, Util::Parameter, and Util::AutoCorrelation< Data, Product >.
Definition at line 61 of file ParamComponent.h.

12.194.3.4 save() `virtual void Util::ParamComponent::save ( Serializable::OArchive & ar ) [inline], [virtual]`

Save internal state to an archive.
The default implementation is empty. This default is used by the Begin, End, and Blank subclasses.
Implements Util::Serializable.
Reimplemented in Util::ParamComposite, Util::FileMaster, Util::Parameter, Util::Manager< Data >, Util::AutoCorrArray< Data, Product >, Util::AutoCorr< Data, Product >, Util::Average, Util::MeanSqDispArray< Data >, Util::Distribution, Util::TensorAverage, Util::SymmTensorAverage, Util::Random, Util::IntDistribution, Util::RadialDistribution, and Util::AutoCorrelation< Data, Product >.
Definition at line 70 of file ParamComponent.h.

12.194.3.5 resetParam() `virtual void Util::ParamComponent::resetParam ( ) [inline], [virtual]`

Nontrivial implementation provided by ParamComposite subclass.
The default implementation is empty. This default is used by all leaf nodes (all other than ParamComposite and subclasses).
Reimplemented in Util::ParamComposite, Util::Begin, and Util::End.
Definition at line 79 of file ParamComponent.h.

12.194.3.6 setIndent()

```
void Util::ParamComponent::setIndent (
            const ParamComponent & parent,
            bool next = true )
```

Set indent level.

If next=true (default) set indent level one higher than that of parent. If next=false, set indent level the same as parent.

Parameters

parent	parent ParamComponent object
next	If true, set level one higher than for parent.

Definition at line 48 of file ParamComponent.cpp.

References indent().

Referenced by Util::Factory< Data >::readObject(), and Util::ParamComposite::setParent().

12.194.3.7 indent()

```
std::string Util::ParamComponent::indent ( ) const
```

Return indent string for this object (string of spaces).

Definition at line 42 of file ParamComponent.cpp.

Referenced by Util::Parameter::load(), Util::ParamComposite::loadOptional(), Util::Begin::readParam(), Util::↩ Parameter::readParam(), setIndent(), Util::End::writeParam(), and Util::Begin::writeParam().

12.194.3.8 serialize()

```
template<class Archive >
void Util::ParamComponent::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize this ParamComponent as a string.

Parameters

ar	saving or loading archive
version	version id for archive

Definition at line 177 of file ParamComponent.h.

References readParam(), and writeParam().

12.194.3.9 initStatic()

```
void Util::ParamComponent::initStatic ( ) [static]
```

Initialize static echo member to false.

Definition at line 77 of file ParamComponent.cpp.

Referenced by Util::initStatic().

12.194.3.10 setEcho()

```
void Util::ParamComponent::setEcho (
            bool echo = true ) [static]
```

Enable or disable echoing for all subclasses of ParamComponent.

When echoing is enabled, all parameters are echoed to a log file immediately after being read. This is useful as an aid to debugging the parameter file, by showing where the error occurred.

Parameters

echo	set true to enable echoing, false to disable.

Definition at line 62 of file ParamComponent.cpp.
References echo().
Referenced by Pscf::Pspg::Continuous::System< D >::setOptions().

12.194.3.11 echo() `bool Util::ParamComponent::echo ( ) [static]`

Get echo parameter.

Returns

true if echoing is enabled, false if disabled.

Definition at line 68 of file ParamComponent.cpp.
Referenced by Util::Manager< Data >::endReadManager(), Util::Parameter::load(), Util::ParamComposite::load(), Util::ParamComposite::loadOptional(), Util::Factory< Data >::readObject(), Util::Blank::readParam(), Util::End::readParam(), Util::Begin::readParam(), Util::Parameter::readParam(), and setEcho().
The documentation for this class was generated from the following files:

- ParamComponent.h
- ParamComponent.cpp

12.195 Util::ParamComposite Class Reference

An object that can read multiple parameters from file.
#include <ParamComposite.h>
Inheritance diagram for Util::ParamComposite:

classUtil_1_1ParamComposite-eps-converted-to.pdf

Public Member Functions

- ParamComposite ()

  Constructor.
- ParamComposite (const ParamComposite &other)

  Copy constructor.
- ParamComposite (int capacity)

  Constructor.
- virtual ∼ParamComposite ()

  Virtual destructor.
- void resetParam ()

  Resets ParamComposite to its empty state.

Read and write functions for the composite

- virtual void readParam (std::istream &in)

  Read the parameter file block.
- virtual void readParamOptional (std::istream &in)

  Read optional parameter file block.
- virtual void readParameters (std::istream &in)

  Read the body of parameter block, without begin and end lines.
- virtual void writeParam (std::ostream &out)

  Write all parameters to an output stream.

Serialization: Load and save functions for this composite

- virtual void load (Serializable::IArchive &ar)

  Load all parameters from an input archive.
- virtual void loadOptional (Serializable::IArchive &ar)

  Load an optional ParamComposite.
- virtual void loadParameters (Serializable::IArchive &ar)

  Load state from archive, without adding Begin and End lines.
- virtual void save (Serializable::OArchive &ar)

  Saves all parameters to an archive.
- void saveOptional (Serializable::OArchive &ar)

  Saves isActive flag, and then calls save() iff isActive is true.

read∗ functions for child components

Each of these functions creates a new instance of a particular subclass of ParamComponent, invokes the readParam() function of the new object to read the associated line or block of a parameter file, and adds the object to the format.

- void readParamComposite (std::istream &in, ParamComposite &child, bool next=true)

  Add and read a required child ParamComposite.
- void readParamCompositeOptional (std::istream &in, ParamComposite &child, bool next=true)

  Add and attempt to read an optional child ParamComposite.
- template< typename Type >
  ScalarParam< Type > & read (std::istream &in, const char ∗label, Type &value)

  Add and read a new required ScalarParam < Type > object.
- template< typename Type >
  ScalarParam< Type > & readOptional (std::istream &in, const char ∗label, Type &value)

  Add and read a new optional ScalarParam < Type > object.
- template< typename Type >
  CArrayParam< Type > & readCArray (std::istream &in, const char ∗label, Type ∗value, int n)

Add and read a required C array parameter.

- template< typename Type >
  CArrayParam< Type > & readOptionalCArray (std::istream &in, const char *label, Type *value, int n)
  Add and read an optional C array parameter.
- template< typename Type >
  DArrayParam< Type > & readDArray (std::istream &in, const char *label, DArray< Type > &array, int n)
  Add and read a required DArray < Type > parameter.
- template< typename Type >
  DArrayParam< Type > & readOptionalDArray (std::istream &in, const char *label, DArray< Type > &array, int n)
  Add and read an optional DArray < Type > parameter.
- template< typename Type , int N>
  FArrayParam< Type, N > & readFArray (std::istream &in, const char *label, FArray< Type, N > &array)
  Add and read a required FArray < Type, N > array parameter.
- template< typename Type , int N>
  FArrayParam< Type, N > & readOptionalFArray (std::istream &in, const char *label, FArray< Type, N > &array)
  Add and read an optional FArray < Type, N > array parameter.
- template< typename Type >
  CArray2DParam< Type > & readCArray2D (std::istream &in, const char *label, Type *value, int m, int n, int np)
  Add and read a required CArray2DParam < Type > 2D C-array.
- template< typename Type >
  CArray2DParam< Type > & readOptionalCArray2D (std::istream &in, const char *label, Type *value, int m, int n, int np)
  Add and read an optional CArray2DParam < Type > 2D C-array parameter.
- template< typename Type >
  DMatrixParam< Type > & readDMatrix (std::istream &in, const char *label, DMatrix< Type > &matrix, int m, int n)
  Add and read a required DMatrix < Type > matrix parameter.
- template< typename Type >
  DMatrixParam< Type > & readOptionalDMatrix (std::istream &in, const char *label, DMatrix< Type > &matrix, int m, int n)
  Add and read an optional DMatrix < Type > matrix parameter.
- template< typename Type >
  DSymmMatrixParam< Type > & readDSymmMatrix (std::istream &in, const char *label, DMatrix< Type > &matrix, int n)
  Add and read a required symmetrix DMatrix.
- template< typename Type >
  DSymmMatrixParam< Type > & readOptionalDSymmMatrix (std::istream &in, const char *label, DMatrix< Type > &matrix, int n)
  Add and read an optional DMatrix matrix parameter.
- Begin & readBegin (std::istream &in, const char *label, bool isRequired=true)
  Add and read a class label and opening bracket.
- End & readEnd (std::istream &in)
  Add and read the closing bracket.
- Blank & readBlank (std::istream &in)
  Add and read a new Blank object, representing a blank line.

load* functions for child components

Load parameters from an Archive, for restarting.

Each of these functions creates a new instance of a subclass of ParamComponent, and invokes the load() function of that new object to load the associated parameter value, and adds the ParamComponent to the format list. These functions are used to load parameters when a program is restarted from a checkpoint file.

- void loadParamComposite (Serializable::IArchive &ar, ParamComposite &child, bool next=true)

  Add and load a required child ParamComposite.
- void loadParamCompositeOptional (Serializable::IArchive &ar, ParamComposite &child, bool next=true)

  Add and load an optional child ParamComposite if isActive.
- template<typename Type >
  ScalarParam< Type > & loadParameter (Serializable::IArchive &ar, const char ∗label, Type &value, bool isRequired)

  Add and load a new ScalarParam < Type > object.
- template<typename Type >
  ScalarParam< Type > & loadParameter (Serializable::IArchive &ar, const char ∗label, Type &value)

  Add and load new required ScalarParam < Type > object.
- template<typename Type >
  CArrayParam< Type > & loadCArray (Serializable::IArchive &ar, const char ∗label, Type ∗value, int n, bool isRequired)

  Add a C array parameter and load its elements.
- template<typename Type >
  CArrayParam< Type > & loadCArray (Serializable::IArchive &ar, const char ∗label, Type ∗value, int n)

  Add and load a required CArrayParam< Type > array parameter.
- template<typename Type >
  DArrayParam< Type > & loadDArray (Serializable::IArchive &ar, const char ∗label, DArray< Type > &array, int n, bool isRequired)

  Add an load a DArray < Type > array parameter.
- template<typename Type >
  DArrayParam< Type > & loadDArray (Serializable::IArchive &ar, const char ∗label, DArray< Type > &array, int n)

  Add and load a required DArray< Type > array parameter.
- template<typename Type , int N>
  FArrayParam< Type, N > & loadFArray (Serializable::IArchive &ar, const char ∗label, FArray< Type, N > &array, bool isRequired)

  Add and load an FArray < Type, N > fixed-size array parameter.
- template<typename Type , int N>
  FArrayParam< Type, N > & loadFArray (Serializable::IArchive &ar, const char ∗label, FArray< Type, N > &array)

  Add and load a required FArray < Type > array parameter.
- template<typename Type >
  CArray2DParam< Type > & loadCArray2D (Serializable::IArchive &ar, const char ∗label, Type ∗value, int m, int n, int np, bool isRequired)

  Add and load a CArray2DParam < Type > C 2D array parameter.
- template<typename Type >
  CArray2DParam< Type > & loadCArray2D (Serializable::IArchive &ar, const char ∗label, Type ∗value, int m, int n, int np)

  Add and load a required < Type > matrix parameter.
- template<typename Type >
  DMatrixParam< Type > & loadDMatrix (Serializable::IArchive &ar, const char ∗label, DMatrix< Type > &matrix, int m, int n, bool isRequired)

  Add and load a DMatrixParam < Type > matrix parameter.
- template<typename Type >
  DMatrixParam< Type > & loadDMatrix (Serializable::IArchive &ar, const char ∗label, DMatrix< Type > &matrix, int m, int n)

  Add and load a required DMatrixParam < Type > matrix parameter.
- template<typename Type >
  DSymmMatrixParam< Type > & loadDSymmMatrix (Serializable::IArchive &ar, const char ∗label, DMatrix< Type > &matrix, int n, bool isRequired)

  Add and load a symmetric DSymmMatrixParam < Type > matrix parameter.

- template< typename Type >
  DSymmMatrixParam< Type > & loadDSymmMatrix (Serializable::IArchive &ar, const char *label, DMatrix< Type > &matrix, int n)

  Add and load a required DSymmMatrixParam < Type > matrix parameter.

add* functions for child components

These functions add a ParamComponent to the format array, but do not read data.

- void addParamComposite (ParamComposite &child, bool next=true)

  Add a child ParamComposite object to the format array.
- Begin & addBegin (const char *label)

  Add a Begin object representing a class name and bracket.
- End & addEnd ()

  Add a closing bracket.
- Blank & addBlank ()

  Create and add a new Blank object, representing a blank line.

Accessors

- std::string className () const

  Get class name string.
- bool isRequired () const

  Is this ParamComposite required in the input file?
- bool isActive () const

  Is this parameter active?
- void setClassName (const char *className)

  Set class name string.
- void setIsRequired (bool isRequired)

  Set or unset the isActive flag.
- void setIsActive (bool isActive)

  Set or unset the isActive flag.
- void setParent (ParamComponent &param, bool next=true)

  Set this to the parent of a child component.
- void addComponent (ParamComponent &param, bool isLeaf=true)

  Add a new ParamComponent object to the format array.
- template< typename Type >
  ScalarParam< Type > & add (std::istream &in, const char *label, Type &value, bool isRequired=true)

  Add a new required ScalarParam < Type > object.
- template< typename Type >
  CArrayParam< Type > & addCArray (std::istream &in, const char *label, Type *value, int n, bool isRequired=true)

  Add (but do not read) a required C array parameter.
- template< typename Type >
  DArrayParam< Type > & addDArray (std::istream &in, const char *label, DArray< Type > &array, int n, bool isRequired=true)

  Add (but do not read) a DArray < Type > parameter.
- template< typename Type , int N>
  FArrayParam< Type, N > & addFArray (std::istream &in, const char *label, FArray< Type, N > &array, bool isRequired=true)

  Add (but do not read) a FArray < Type, N > array parameter.

- template< typename Type >
  CArray2DParam< Type > & addCArray2D (std::istream &in, const char ∗label, Type ∗value, int m, int n, int np, bool isRequired=true)

  Add (but do not read) a CArray2DParam < Type > 2D C-array.
- template< typename Type >
  DMatrixParam< Type > & addDMatrix (std::istream &in, const char ∗label, DMatrix< Type > &matrix, int m, int n, bool isRequired=true)

  Add and read a required DMatrix < Type > matrix parameter.

Additional Inherited Members

12.195.1 Detailed Description

An object that can read multiple parameters from file.
Any class that reads a block of parameters from a parameter file must be derived from ParamComposite. Each such class must implement either the readParameters() function or the readParam() function, but not both. The readParameters(), if reimplemented, should read the body of the associated parameter file block, without opening or closing lines. The readParam() function reads the the entire block, including opening line and closing lines. The default implementation of readParam() reads the opening line of the block, calls readParameters() to read the body of the block, and then reads the closing line. Most subclasses of ParamComposite re-implement the readParameters() function, and rely on the default implementation of readParam() to add the Begin and End lines.
The writeParam() function, if called after readParam(), writes the associated parameter block using the same file format as that used to read the data in the earlier call to readParam().

12.195.1.1 Implementation details:

After parameter file block is read from file, the file format is stored as a private array of ParaComponent∗ pointers. We will refer to this in what follows as the format_ array. Each pointer in this array may point to a Parameter, ParamComposite, Begin, End, or Blank object. Pointers to these objects are added to the format array as the associated objects are read from file, and are stored in the same order as they appear in the parameter file. The default implementation of the writeParam() function simply calls the writeParam() function of each child ParamComponent.

12.195.1.2 Subclass implementation details:

The readParameters() function of each subclass of ParamComposite should be implemented using protected member functions of ParamComposite with names that begin with "read". The read<T>() function template can be used to read an individual parameter, while readParamComposite reads the nested subblock associated with a child ParamComposite. There are also more specialized methods (e.g., readDArray<T>to read different types of arrays and matrices of parameters, and to read optional parameters. See the users manual for further details.
The setClassName() and className() functions may be used to set and get a std::string containing the subclass name. The setClassName() function should be called in the constructor of each subclass of ParamComposite. The class name set in the constructor of a subclass will replace any name set by a base class, because of the order in which constructors are called. The default implementation of ParamComposite::readParam() checks if the class name that appears in the opening line of a parameter block agrees with the class name returned by the className() function, and throws an exception if it does not.
Definition at line 89 of file ParamComposite.h.

12.195.2 Constructor & Destructor Documentation

12.195.2.1 ParamComposite() [1/3]

`Util::ParamComposite::ParamComposite ( )`

Constructor.
Definition at line 23 of file ParamComposite.cpp.

12.195.2.2 ParamComposite() [2/3] `Util::ParamComposite::ParamComposite ( const ParamComposite & other )`

Copy constructor.
Definition at line 55 of file ParamComposite.cpp.

12.195.2.3 ParamComposite() [3/3] `Util::ParamComposite::ParamComposite ( int capacity )`

Constructor.
Reserve space for capacity elements in the format array.

Parameters

capacity	maximum length of parameter list

Definition at line 36 of file ParamComposite.cpp.
References UTIL_THROW.

12.195.2.4 ∼ParamComposite() `Util::ParamComposite::∼ParamComposite ( ) [virtual]`

Virtual destructor.
Definition at line 68 of file ParamComposite.cpp.

12.195.3 Member Function Documentation

12.195.3.1 resetParam() `void Util::ParamComposite::resetParam ( ) [virtual]`

Resets ParamComposite to its empty state.
This function deletes Parameter, Begin, End, and Blank objects in the format array (i.e., all "leaf" objects in the format tree), invokes the resetParam() function of any child ParamComposite in the format array, and clears the format array.
Reimplemented from Util::ParamComponent.
Definition at line 209 of file ParamComposite.cpp.

12.195.3.2 readParam() `void Util::ParamComposite::readParam ( std::istream & in ) [virtual]`

Read the parameter file block.
Inherited from ParamComponent. This function reads the entire parameter block for this ParamComposite, including an opening line, which is of the form "ClassName{", and the closing line, which contains only a closing bracket, "}". The default implementation reads the opening line (a Begin object), calls the virtual readParameters function to read the body of the block, and reads the closing line (an End object).

Exceptions

Throws	if the string in the opening line does not match the string returned by the classname() function.

Parameters

in	input stream for reading

Implements Util::ParamComponent.
Reimplemented in Util::Manager< Data >.
Definition at line 88 of file ParamComposite.cpp.
References readBegin(), readEnd(), and readParameters().
Referenced by readParamComposite().

12.195.3.3 readParamOptional() `void Util::ParamComposite::readParamOptional ( std::istream & in ) [virtual]`

Read optional parameter file block.
Read an optional ParamComposite. This function must use a Label() object to read the opening "ClassName{" line, and then continues to read the rest of the block if and only if the class name in the opening line matches the string returned by the classname() function.
If the first line matches, the default implementation calls the readParameters() member function to read the body of the block, and then reads the ending line.

Parameters

in	input stream for reading

Reimplemented in Util::Manager< Data >.
Definition at line 101 of file ParamComposite.cpp.
References Util::Begin::isActive(), readBegin(), readEnd(), and readParameters().
Referenced by readParamCompositeOptional().

12.195.3.4 readParameters() `virtual void Util::ParamComposite::readParameters ( std::istream & in ) [inline], [virtual]`

Read the body of parameter block, without begin and end lines.
Most subclasses of ParamComposite should re-implement this function, which has an empty default implementation. Every subclass of Paramcomposite must either: (1) Re-implement this function and rely on the default implementation of readParam(), which calls this function. (2) Re-implement readParam() itself. Option (1) is far more common. Option (2) is required only for classes that require a non-standard treatment of the beginning and ending lines (e.g., the Manager class template).

Parameters

in	input stream for reading

Reimplemented in Util::FileMaster, Util::Manager< Data >, Pscf::Pspg::Continuous::Mixture< D >, Util::AutoCorrArray< Data, Product >, Util::AutoCorr< Data, Product >, Pscf::PolymerTmpl< Block >, Pscf::PolymerTmpl< Block< D > >, Util::TensorAverage, Util::SymmTensorAverage, Util::Random, Util::Average, Util::MeanSqDispArray< Data >, Pscf::Pspg::Continuous::AmIterator< D >, Pscf::Pspg::Discrete::DMixtureTmpl< TP, TS >, Pscf::Pspg::Discrete::DMixtureTmpl< DPolymer< D >, Solvent< D > >, Util::Distribution, Pscf::Homogeneous::Molecule, Util::IntDistribution, Pscf::MixtureTmpl< TP, TS >, Pscf::MixtureTmpl< Polymer< D >, S Pscf::Homogeneous::Mixture, Pscf::Pspg::Discrete::DMixture< D >, Util::RadialDistribution, Pscf::Pspg::Discrete::AmIterator< D >, Util::AutoCorrelation< Data, Product >, Pscf::ChiInteraction, Pscf::DPolymerTmpl< Bond >, and Pscf::DPolymerTmpl< Bond< D > >.
Definition at line 180 of file ParamComposite.h.
Referenced by readParam(), and readParamOptional().

12.195.3.5 writeParam() `void Util::ParamComposite::writeParam ( std::ostream & out ) [virtual]`

Write all parameters to an output stream.

The default implementation iterates through the format array, and calls the readParam() member function of each ParamComponent in the array. This is sufficient for most subclasses.

Parameters

out	output stream for reading

Implements Util::ParamComponent.
Definition at line 119 of file ParamComposite.cpp.

12.195.3.6 load() `void Util::ParamComposite::load ( Serializable::IArchive & ar ) [virtual]`

Load all parameters from an input archive.
This function is inherited from Serializable. The default implementation of ParamComposite::load() calls loadParameters, and adds Begin and End lines to the format array.. All subclasses of ParamComposite should overload the virtual load←Parameters member function.

Parameters

ar	input/loading archive.

Reimplemented from Util::ParamComponent.
Reimplemented in Util::AutoCorrelation< Data, Product >.
Definition at line 131 of file ParamComposite.cpp.
References addBegin(), addEnd(), Util::ParamComponent::echo(), Util::Log::file(), Util::MpiFileIo::isIoProcessor(), loadParameters(), Util::End::writeParam(), and Util::Begin::writeParam().
Referenced by loadOptional(), and loadParamComposite().

12.195.3.7 loadOptional() `void Util::ParamComposite::loadOptional ( Serializable::IArchive & ar ) [virtual]`

Load an optional ParamComposite.
Loads isActive, and calls load(ar) if active.

Parameters

ar	input/loading archive.

Definition at line 152 of file ParamComposite.cpp.
References Util::bcast< bool >(), className(), Util::ParamComponent::echo(), Util::Log::file(), Util::MpiFileIo::has←IoCommunicator(), Util::ParamComponent::indent(), Util::MpiFileIo::ioCommunicator(), Util::MpiFileIo::isIoProcessor(), load(), and UTIL_THROW.
Referenced by loadParamCompositeOptional().

12.195.3.8 loadParameters() `virtual void Util::ParamComposite::loadParameters ( Serializable::IArchive & ar ) [inline], [virtual]`

Load state from archive, without adding Begin and End lines.
This function should be re-implemented by all subclasses that have an internal state that should be saved in a restart file. The default implementation is empty. Subclass implementations should load the entire internal state from the archive, including parameters that appear in the parameter file and any persistent private member variables that do not appear in the parameter file.

Parameters

ar	input/loading archive.

Reimplemented in Util::FileMaster, Util::Manager< Data >, Util::AutoCorrArray< Data, Product >, Util::AutoCorr< Data, Product >, Util::Average, Util::MeanSqDispArray< Data >, Util::Distribution, Util::TensorAverage, Util::SymmTensorAverage, Util::Random, Util::IntDistribution, and Util::RadialDistribution.
Definition at line 233 of file ParamComposite.h.
Referenced by load().

12.195.3.9 save() `void Util::ParamComposite::save ( Serializable::OArchive & ar ) [virtual]`

Saves all parameters to an archive.
The default implementation simply calls the save function for all items in the parameter file format array. This is often not sufficient. Specifically, it is not sufficient for classes that contain any persistent member variables that do not appear in the parameter file format.
If a class also defines a serialize function template, which allows instances to be serialized to any type of archive, then the save function can often be implemented as follows:

```
void save(Serializable::OArchive& ar)
{ ar & *this; }
```

Parameters

ar	output/saving archive.

Reimplemented from Util::ParamComponent.
Reimplemented in Util::FileMaster, Util::Manager< Data >, Util::AutoCorrArray< Data, Product >, Util::AutoCorr< Data, Product >, Util::Average, Util::MeanSqDispArray< Data >, Util::Distribution, Util::TensorAverage, Util::SymmTensorAverage, Util::Random, Util::IntDistribution, Util::RadialDistribution, and Util::AutoCorrelation< Data, Product >.
Definition at line 188 of file ParamComposite.cpp.
Referenced by saveOptional().

12.195.3.10 saveOptional() `void Util::ParamComposite::saveOptional ( Serializable::OArchive & ar )`

Saves isActive flag, and then calls save() iff isActive is true.

Parameters

ar	output/saving archive.

Definition at line 198 of file ParamComposite.cpp.
References save().

12.195.3.11 readParamComposite() `void Util::ParamComposite::readParamComposite ( std::istream & in, ParamComposite & child, bool next = true )`

Add and read a required child ParamComposite.

Parameters

in	input stream for reading
child	child ParamComposite object
next	true if the indent level is one higher than parent.

Definition at line 260 of file ParamComposite.cpp.
References addParamComposite(), and readParam().
Referenced by Pscf::Homogeneous::Mixture::readParameters().

12.195.3.12 readParamCompositeOptional() `void Util::ParamComposite::readParamCompositeOptional ( std::istream & in, ParamComposite & child, bool next = true )`

Add and attempt to read an optional child ParamComposite.

Parameters

in	input stream for reading
child	child ParamComposite object
next	true if the indent level is one higher than parent.

Definition at line 271 of file ParamComposite.cpp.
References addParamComposite(), and readParamOptional().

12.195.3.13 read() `template<typename Type > ScalarParam< Type > & Util::ParamComposite::read ( std::istream & in, const char * label, Type & value )`

Add and read a new required ScalarParam < Type > object.
This is equivalent to ScalarParam<Type>(in, label, value, true).

Parameters

in	input stream for reading
label	Label string
value	reference to new ScalarParam< Type >

Definition at line 1156 of file ParamComposite.h.

12.195.3.14 readOptional() `template<typename Type > ScalarParam< Type > & Util::ParamComposite::readOptional ( std::istream & in, const char * label, Type & value ) [inline]`

Add and read a new optional ScalarParam < Type > object.
This is equivalent to ScalarParam<Type>(in, label, value, false).

Parameters

in	input stream for reading
label	Label string
value	reference to new ScalarParam< Type >

Definition at line 1164 of file ParamComposite.h.

12.195.3.15 readCArray() `template<typename Type >`

```
CArrayParam< Type > & Util::ParamComposite::readCArray (
              std::istream & in,
              const char * label,
              Type * value,
              int n ) [inline]
```

Add and read a required C array parameter.

Parameters

in	input stream for reading
label	Label string for new array
value	pointer to array
n	number of elements

Returns

reference to the new CArrayParam<Type> object

Definition at line 1231 of file ParamComposite.h.

12.195.3.16 readOptionalCArray() `template<typename Type >`

```
CArrayParam< Type > & Util::ParamComposite::readOptionalCArray (
              std::istream & in,
              const char * label,
              Type * value,
              int n ) [inline]
```

Add and read an optional C array parameter.

Parameters

in	input stream for reading
label	Label string for new array
value	pointer to array
n	number of elements

Returns

reference to the new CArrayParam<Type> object

Definition at line 1240 of file ParamComposite.h.

12.195.3.17 readDArray() `template<typename Type >`

```
DArrayParam< Type > & Util::ParamComposite::readDArray (
            std::istream & in,
            const char * label,
            DArray< Type > & array,
            int n ) [inline]
```

Add and read a required DArray < Type > parameter.

Parameters

in	input stream for reading
label	Label string for new array
array	DArray object
n	number of elements

Returns

reference to the new DArrayParam<Type> object

Definition at line 1308 of file ParamComposite.h.

12.195.3.18 readOptionalDArray() `template<typename Type >`

```
DArrayParam< Type > & Util::ParamComposite::readOptionalDArray (
            std::istream & in,
            const char * label,
            DArray< Type > & array,
            int n ) [inline]
```

Add and read an optional DArray < Type > parameter.

Parameters

in	input stream for reading
label	Label string for new array
array	DArray object
n	number of elements

Returns

reference to the new DArrayParam<Type> object

Definition at line 1318 of file ParamComposite.h.

12.195.3.19 readFArray() `template<typename Type , int N>`

```
FArrayParam< Type, N > & Util::ParamComposite::readFArray (
            std::istream & in,
            const char * label,
            FArray< Type, N > & array ) [inline]
```

Add and read a required FArray < Type, N > array parameter.

Parameters

in	input stream for reading
label	Label string for new array
array	FArray object

Returns

reference to the new FArrayParam<Type, N> object

Definition at line 1386 of file ParamComposite.h.

12.195.3.20 readOptionalFArray() `template<typename Type , int N>`

```
FArrayParam< Type, N > & Util::ParamComposite::readOptionalFArray (
            std::istream & in,
            const char * label,
            FArray< Type, N > & array )  [inline]
```

Add and read an optional FArray < Type, N > array parameter.

Parameters

in	input stream for reading
label	Label string for new array
array	FArray object

Returns

reference to the new FArrayParam<Type, N> object

Definition at line 1396 of file ParamComposite.h.

12.195.3.21 readCArray2D() `template<typename Type >`

```
CArray2DParam< Type > & Util::ParamComposite::readCArray2D (
            std::istream & in,
            const char * label,
            Type * value,
            int m,
            int n,
            int np )  [inline]
```

Add and read a required CArray2DParam < Type > 2D C-array.

Parameters

in	input stream for reading
label	Label string for new array
value	pointer to array
m	number of rows (1st dimension)
n	logical number of columns (2nd dimension)
np	physical number of columns (elements allocated per row)

Returns

reference to the CArray2DParam<Type> object

Definition at line 1456 of file ParamComposite.h.

12.195.3.22 readOptionalCArray2D() `template<typename Type >`

```
CArray2DParam< Type > & Util::ParamComposite::readOptionalCArray2D (
            std::istream & in,
            const char * label,
            Type * value,
            int m,
            int n,
            int np ) [inline]
```

Add and read an optional CArray2DParam < Type > 2D C-array parameter.

Parameters

in	input stream for reading
label	Label string for new array
value	pointer to array
m	number of rows (1st dimension)
n	logical number of columns (2nd dimension)
np	physical number of columns (elements allocated per row)

Returns

reference to the CArray2DParam<Type> object

Definition at line 1465 of file ParamComposite.h.

12.195.3.23 readDMatrix() `template<typename Type >`

```
DMatrixParam< Type > & Util::ParamComposite::readDMatrix (
            std::istream & in,
            const char * label,
            DMatrix< Type > & matrix,
            int m,
            int n ) [inline]
```

Add and read a required DMatrix < Type > matrix parameter.

Parameters

in	input stream for reading
label	Label string for new array
matrix	DMatrix object
m	number of rows (1st dimension)
n	number of columns (2nd dimension)

Returns

reference to the DMatrixParam<Type> object

Definition at line 1538 of file ParamComposite.h.

12.195.3.24 readOptionalDMatrix() template<typename Type >

```
DMatrixParam< Type > & Util::ParamComposite::readOptionalDMatrix (
            std::istream & in,
            const char * label,
            DMatrix< Type > & matrix,
            int m,
            int n ) [inline]
```

Add and read an optional DMatrix < Type > matrix parameter.

Parameters

in	input stream for reading
label	Label string for new array
matrix	DMatrix object
m	number of rows (1st dimension)
n	number of columns (2nd dimension)

Returns

reference to the DMatrixParam<Type> object

Definition at line 1547 of file ParamComposite.h.

12.195.3.25 readDSymmMatrix() template<typename Type >

```
DSymmMatrixParam< Type > & Util::ParamComposite::readDSymmMatrix (
            std::istream & in,
            const char * label,
            DMatrix< Type > & matrix,
            int n ) [inline]
```

Add and read a required symmetrix DMatrix.

Parameters

in	input stream for reading
label	Label string for new array
matrix	DMatrix object
n	number of rows or columns

Returns

reference to the DMatrixParam<Type> object

Definition at line 1603 of file ParamComposite.h.
Referenced by Pscf::ChiInteraction::readParameters().

12.195.3.26 readOptionalDSymmMatrix()

```
template<typename Type >
DSymmMatrixParam< Type > & Util::ParamComposite::readOptionalDSymmMatrix (
            std::istream & in,
            const char * label,
            DMatrix< Type > & matrix,
            int n )  [inline]
```

Add and read an optional DMatrix matrix parameter.

Parameters

in	input stream for reading
label	Label string for new array
matrix	DMatrix object
n	number of rows or columns

Returns

reference to the DMatrixParam<Type> object

Definition at line 1614 of file ParamComposite.h.

12.195.3.27 readBegin()

```
Begin & Util::ParamComposite::readBegin (
            std::istream & in,
            const char * label,
            bool isRequired = true )
```

Add and read a class label and opening bracket.

Parameters

in	input stream for reading
label	class name string, without trailing bracket
isRequired	Is this the beginning of a required element?

Returns

reference to the new Begin object

Definition at line 316 of file ParamComposite.cpp.
References addComponent(), Util::Begin::isActive(), isRequired(), Util::Begin::readParam(), and setParent().
Referenced by readParam(), and readParamOptional().

12.195.3.28 readEnd()

```
End & Util::ParamComposite::readEnd (
            std::istream & in )
```

Add and read the closing bracket.

Parameters

in	input stream for reading

Returns

reference to the new End object

Definition at line 344 of file ParamComposite.cpp.
References addEnd(), and Util::End::readParam().
Referenced by readParam(), and readParamOptional().

12.195.3.29 readBlank()

```
Blank & Util::ParamComposite::readBlank (
        std::istream & in )
```

Add and read a new Blank object, representing a blank line.

Parameters

in	input stream for reading

Returns

reference to the new Blank object

Definition at line 367 of file ParamComposite.cpp.
References addBlank(), and Util::Blank::readParam().

12.195.3.30 loadParamComposite()

```
void Util::ParamComposite::loadParamComposite (
        Serializable::IArchive & ar,
        ParamComposite & child,
        bool next = true )
```

Add and load a required child ParamComposite.

Parameters

ar	input archive for loading
child	child ParamComposite object
next	true if the indent level is one higher than parent.

Definition at line 282 of file ParamComposite.cpp.
References addParamComposite(), and load().
Referenced by Util::Factory< Data >::loadObject().

12.195.3.31 loadParamCompositeOptional()

```
void Util::ParamComposite::loadParamCompositeOptional (
        Serializable::IArchive & ar,
        ParamComposite & child,
        bool next = true )
```

Add and load an optional child ParamComposite if isActive.
This functional loads the isActive flag, and then calls the load function of the child iff isActive is true.

Parameters

ar	input archive for loading
child	child ParamComposite object
next	true if the indent level is one higher than parent.

Definition at line 293 of file ParamComposite.cpp.
References addParamComposite(), and loadOptional().

12.195.3.32 loadParameter() [1/2] template<typename Type >

```
ScalarParam< Type > & Util::ParamComposite::loadParameter (
            Serializable::IArchive & ar,
            const char * label,
            Type & value,
            bool isRequired )
```

Add and load a new ScalarParam < Type > object.
An optional parameter is indicated by setting isRequired = false. Optional parameters must be saved using the Parameter::saveOptional() static member function.

Parameters

ar	archive for loading
label	Label string
value	reference to the Type variable
isRequired	Is this a required parameter?

Returns

reference to the new ScalarParam < Type > object

Definition at line 1173 of file ParamComposite.h.
References addComponent(), isRequired(), Util::Parameter::load(), and setParent().

12.195.3.33 loadParameter() [2/2] template<typename Type >

```
ScalarParam< Type > & Util::ParamComposite::loadParameter (
            Serializable::IArchive & ar,
            const char * label,
            Type & value ) [inline]
```

Add and load new required ScalarParam < Type > object.
Equivalent to loadParameter < Type > (ar, label, value, true).

Parameters

ar	archive for loading
label	label string
value	reference to the Type variable

Returns

reference to the new ScalarParam < Type > object

Definition at line 1189 of file ParamComposite.h.

12.195.3.34 loadCArray() [1/2] template<typename Type >

```
CArrayParam< Type > & Util::ParamComposite::loadCArray (
```

```
Serializable::IArchive & ar,
const char * label,
Type * value,
int n,
bool isRequired )
```

Add a C array parameter and load its elements.

Parameters

ar	archive for loading
label	label string for new array
value	pointer to array
n	number of elements
isRequired	Is this a required parameter?

Returns

reference to the new CArrayParam<Type> object

Definition at line 1249 of file ParamComposite.h.
References addComponent(), isRequired(), Util::Parameter::load(), and setParent().

12.195.3.35 loadCArray() [2/2]

```
template<typename Type >
CArrayParam< Type > & Util::ParamComposite::loadCArray (
    Serializable::IArchive & ar,
    const char * label,
    Type * value,
    int n ) [inline]
```

Add and load a required CArrayParam< Type > array parameter.
Equivalent to loadCArray < Type > (ar, label, value, n, true).

Parameters

ar	archive for loading
label	label string for new array
value	pointer to array
n	number of elements

Returns

reference to the new CArrayParam<Type> object

Definition at line 1265 of file ParamComposite.h.

12.195.3.36 loadDArray() [1/2]

```
template<typename Type >
DArrayParam< Type > & Util::ParamComposite::loadDArray (
    Serializable::IArchive & ar,
    const char * label,
    DArray< Type > & array,
    int n,
    bool isRequired )
```

Add an load a DArray < Type > array parameter.

Parameters

ar	archive for loading
label	Label string for new array
array	DArray object
n	number of elements (logical size)
isRequired	Is this a required parameter?

Returns

reference to the new DArrayParam<Type> object

Definition at line 1327 of file ParamComposite.h.
References addComponent(), isRequired(), Util::Parameter::load(), and setParent().

12.195.3.37 loadDArray() [2/2] template<typename Type >

```
DArrayParam< Type > & Util::ParamComposite::loadDArray (
            Serializable::IArchive & ar,
            const char * label,
            DArray< Type > & array,
            int n ) [inline]
```

Add and load a required DArray< Type > array parameter.
Equivalent to loadDArrayParam < Type > (ar, label, array, n, true).

Parameters

ar	archive for loading
label	Label string for new array
array	DArray object
n	number of elements (logical size)

Returns

reference to the new DArrayParam<Type> object

Definition at line 1343 of file ParamComposite.h.

12.195.3.38 loadFArray() [1/2] template<typename Type , int N>

```
FArrayParam< Type, N > & Util::ParamComposite::loadFArray (
            Serializable::IArchive & ar,
            const char * label,
            FArray< Type, N > & array,
            bool isRequired )
```

Add and load an FArray < Type, N > fixed-size array parameter.

Parameters

ar	archive for loading

Parameters

label	label string for new array
array	FArray object
isRequired	Is this a required parameter?

Returns

reference to the new FArrayParam<Type, N> object

Definition at line 1405 of file ParamComposite.h.
References addComponent(), isRequired(), Util::Parameter::load(), and setParent().

12.195.3.39 loadFArray() [2/2] `template<typename Type , int N>`

```
FArrayParam<Type, N>& Util::ParamComposite::loadFArray (
        Serializable::IArchive & ar,
        const char * label,
        FArray< Type, N > & array ) [inline]
```

Add and load a required FArray < Type > array parameter.
Equivalent to loadFArrayParam < Type > (ar, label, array, true).

Parameters

ar	archive for loading
label	label string for new array
array	FArray object

Returns

reference to the new FArrayParam<Type, N> object

Definition at line 675 of file ParamComposite.h.

12.195.3.40 loadCArray2D() [1/2] `template<typename Type >`

```
CArray2DParam< Type > & Util::ParamComposite::loadCArray2D (
        Serializable::IArchive & ar,
        const char * label,
        Type * value,
        int m,
        int n,
        int np,
        bool isRequired )
```

Add and load a CArray2DParam < Type > C 2D array parameter.

Parameters

ar	archive for loading
label	Label string for new array
value	pointer to array
m	number of rows (1st dimension)

Parameters

n	logical number of columns (2nd dimension)
np	physical number of columns (elements allocated per row)
isRequired	Is this a required parameter?

Returns

reference to the CArray2DParam<Type> object

Definition at line 1475 of file ParamComposite.h.
References addComponent(), isRequired(), Util::Parameter::load(), and setParent().

12.195.3.41 loadCArray2D() [2/2]

```
template<typename Type >
CArray2DParam< Type > & Util::ParamComposite::loadCArray2D (
        Serializable::IArchive & ar,
        const char * label,
        Type * value,
        int m,
        int n,
        int np )
```

Add and load a required < Type > matrix parameter.
Equivalent to loadCArray2DParam < Type > (ar, label, value, m, n, np, true).

Parameters

ar	archive for loading
label	Label string for new array
value	pointer to array
m	number of rows (1st dimension)
n	logical number of columns (2nd dimension)
np	physical number of columns (elements allocated per row)

Returns

reference to the CArray2DParam<Type> object

Definition at line 1493 of file ParamComposite.h.

12.195.3.42 loadDMatrix() [1/2]

```
template<typename Type >
DMatrixParam< Type > & Util::ParamComposite::loadDMatrix (
        Serializable::IArchive & ar,
        const char * label,
        DMatrix< Type > & matrix,
        int m,
        int n,
        bool isRequired )
```

Add and load a DMatrixParam < Type > matrix parameter.

Parameters

ar	archive for loading
label	Label string for new array
matrix	DMatrix object
m	number of rows (1st dimension)
n	number of columns (2nd dimension)
isRequired	Is this a required parameter?

Returns

reference to the DMatrixParam<Type> object

Definition at line 1556 of file ParamComposite.h.
References addComponent(), isRequired(), Util::Parameter::load(), and setParent().

12.195.3.43 loadDMatrix() [2/2]

```
template<typename Type >
DMatrixParam< Type > & Util::ParamComposite::loadDMatrix (
        Serializable::IArchive & ar,
        const char * label,
        DMatrix< Type > & matrix,
        int m,
        int n ) [inline]
```

Add and load a required DMatrixParam < Type > matrix parameter.

Parameters

ar	archive for loading
label	Label string for new array
matrix	DMatrix object
m	number of rows (1st dimension)
n	number of columns (2nd dimension)

Returns

reference to the DMatrixParam<Type> object

Definition at line 1573 of file ParamComposite.h.

12.195.3.44 loadDSymmMatrix() [1/2]

```
template<typename Type >
DSymmMatrixParam< Type > & Util::ParamComposite::loadDSymmMatrix (
        Serializable::IArchive & ar,
        const char * label,
        DMatrix< Type > & matrix,
        int n,
        bool isRequired )
```

Add and load a symmetric DSymmMatrixParam < Type > matrix parameter.

Parameters

ar	archive for loading
label	Label string for new array
matrix	DMatrix object
n	number of rows or columns
isRequired	Is this a required parameter?

Returns

reference to the DMatrixParam<Type> object

Definition at line 1625 of file ParamComposite.h.
References addComponent(), isRequired(), Util::Parameter::load(), and setParent().

12.195.3.45 loadDSymmMatrix() [2/2]

```
template<typename Type >
DSymmMatrixParam< Type > & Util::ParamComposite::loadDSymmMatrix (
            Serializable::IArchive & ar,
            const char * label,
            DMatrix< Type > & matrix,
            int n ) [inline]
```

Add and load a required DSymmMatrixParam < Type > matrix parameter.

Parameters

ar	archive for loading
label	Label string for new array
matrix	DMatrix object
n	number of rows or columns

Returns

reference to the DMatrixParam<Type> object

Definition at line 1644 of file ParamComposite.h.

12.195.3.46 addParamComposite()

```
void Util::ParamComposite::addParamComposite (
            ParamComposite & child,
            bool next = true )
```

Add a child ParamComposite object to the format array.

Parameters

child	child ParamComposite object
next	true if the indent level is one higher than parent.

Definition at line 249 of file ParamComposite.cpp.
References addComponent(), and setParent().
Referenced by loadParamComposite(), loadParamCompositeOptional(), Util::Factory< Data >::readObject(), readParamComposite(), and readParamCompositeOptional().

12.195.3.47 addBegin() `Begin & Util::ParamComposite::addBegin ( const char * label )`

Add a Begin object representing a class name and bracket.

Parameters

label	class name string, without trailing bracket

Returns

reference to the new begin object.

Definition at line 305 of file ParamComposite.cpp.
References addComponent(), and setParent().
Referenced by load().

12.195.3.48 addEnd() `End & Util::ParamComposite::addEnd ( )`

Add a closing bracket.

Returns

reference to the new End object.

Definition at line 333 of file ParamComposite.cpp.
References addComponent(), and setParent().
Referenced by load(), and readEnd().

12.195.3.49 addBlank() `Blank & Util::ParamComposite::addBlank ( )`

Create and add a new Blank object, representing a blank line.

Returns

reference to the new Blank object

Definition at line 356 of file ParamComposite.cpp.
References addComponent(), and setParent().
Referenced by readBlank().

12.195.3.50 className() `std::string Util::ParamComposite::className ( ) const [inline]`

Get class name string.
Definition at line 1103 of file ParamComposite.h.
Referenced by Util::Manager< Data >::beginReadManager(), loadOptional(), and setClassName().

12.195.3.51 isRequired() `bool Util::ParamComposite::isRequired ( ) const [inline]`

Is this ParamComposite required in the input file?
Definition at line 1109 of file ParamComposite.h.
Referenced by add(), addCArray(), addCArray2D(), addDArray(), addDMatrix(), addFArray(), loadCArray(), loadC←Array2D(), loadDArray(), loadDMatrix(), loadDSymmMatrix(), loadFArray(), loadParameter(), readBegin(), and setIs←Required().

12.195.3.52 isActive() `bool Util::ParamComposite::isActive ( ) const [inline]`

Is this parameter active?

Definition at line 1115 of file ParamComposite.h.

Referenced by setIsActive().

12.195.3.53 setClassName() `void Util::ParamComposite::setClassName ( const char * className ) [protected]`

Set class name string.

Should be set in subclass constructor.

Definition at line 377 of file ParamComposite.cpp.

References className().

Referenced by Pscf::Pspg::Discrete::AmIterator< D >::AmIterator(), Pscf::Pspg::Continuous::AmIterator< D >::AmIterator(), Util::AutoCorr< Data, Product >::AutoCorr(), Util::AutoCorrArray< Data, Product >::AutoCorrArray(), Util::Average::Average(), Pscf::ChiInteraction::ChiInteraction(), Util::Distribution::Distribution(), Util::FileMaster::FileMaster(), Util::IntDistribution::IntDistribution(), Pscf::Interaction::Interaction(), Pscf::Pspg::Continuous::Iterator< D >::Iterator(), Pscf::Homogeneous::Mixture::Mixture(), Pscf::Pspg::Continuous::Mixture< D >::Mixture(), Pscf::Homogeneous::Molecule::Molecule(), Util::RadialDistribution::RadialDistribution(), Util::Random::Random(), Util::SymmTensorAverage::SymmTensorAverage(), and Util::TensorAverage::TensorAverage().

12.195.3.54 setIsRequired() `void Util::ParamComposite::setIsRequired ( bool isRequired ) [protected]`

Set or unset the isActive flag.

Required to re-implement readParam[Optional].

Parameters

isRequired	flag to set true or false.

Definition at line 383 of file ParamComposite.cpp.

References isRequired().

12.195.3.55 setIsActive() `void Util::ParamComposite::setIsActive ( bool isActive ) [protected]`

Set or unset the isActive flag.

Required to re-implement readParam[Optional].

Parameters

isActive	flag to set true or false.

Definition at line 394 of file ParamComposite.cpp.

References isActive(), and UTIL_THROW.

12.195.3.56 setParent() `void Util::ParamComposite::setParent ( ParamComponent & param, bool next = true ) [protected]`

Set this to the parent of a child component.

This function sets the indent and (ifdef UTIL_MPI) the ioCommunicator of the child component.

Parameters

param	child ParamComponent
next	if true, set indent level one higher than for parent.

Definition at line 224 of file ParamComposite.cpp.

References Util::MpiFileIo::hasIoCommunicator(), Util::MpiFileIo::ioCommunicator(), Util::ParamComponent::set←Indent(), and Util::MpiFileIo::setIoCommunicator().

Referenced by add(), addBegin(), addBlank(), addCArray(), addCArray2D(), addDArray(), addDMatrix(), addEnd(), addFArray(), addParamComposite(), loadCArray(), loadCArray2D(), loadDArray(), loadDMatrix(), loadDSymmMatrix(), loadFArray(), loadParameter(), and readBegin().

12.195.3.57 addComponent()

```
void Util::ParamComposite::addComponent (
        ParamComponent & param,
        bool isLeaf = true )  [protected]
```

Add a new ParamComponent object to the format array.

Parameters

param	Parameter object
isLeaf	Is this a leaf or a ParamComposite node?

Definition at line 237 of file ParamComposite.cpp.

Referenced by add(), addBegin(), addBlank(), addCArray(), addCArray2D(), addDArray(), addDMatrix(), addEnd(), addFArray(), addParamComposite(), loadCArray(), loadCArray2D(), loadDArray(), loadDMatrix(), loadDSymmMatrix(), loadFArray(), loadParameter(), and readBegin().

12.195.3.58 add()

```
template<typename Type >
ScalarParam< Type > & Util::ParamComposite::add (
        std::istream & in,
        const char * label,
        Type & value,
        bool isRequired = true )  [protected]
```

Add a new required ScalarParam < Type > object.

Parameters

in	input stream for reading
label	Label string
value	reference to new ScalarParam< Type >
isRequired	Is this a required parameter?

Returns

reference to the new ScalarParam<Type> object

Definition at line 1141 of file ParamComposite.h.

References addComponent(), isRequired(), and setParent().

12.195.3.59 addCArray()

```
template<typename Type >
CArrayParam< Type > & Util::ParamComposite::addCArray (
            std::istream & in,
            const char * label,
            Type * value,
            int n,
            bool isRequired = true ) [protected]
```

Add (but do not read) a required C array parameter.

Parameters

in	input stream for reading
label	Label string for new array
value	pointer to array
n	number of elements
isRequired	Is this a required parameter?

Returns

reference to the new CArrayParam<Type> object

Definition at line 1216 of file ParamComposite.h.
References addComponent(), isRequired(), and setParent().

12.195.3.60 addDArray()

```
template<typename Type >
DArrayParam< Type > & Util::ParamComposite::addDArray (
            std::istream & in,
            const char * label,
            DArray< Type > & array,
            int n,
            bool isRequired = true ) [protected]
```

Add (but do not read) a DArray < Type > parameter.

Parameters

in	input stream for reading
label	Label string for new array
array	DArray object
n	number of elements
isRequired	Is this a required parameter?

Returns

reference to the new DArrayParam<Type> object

Definition at line 1292 of file ParamComposite.h.
References addComponent(), isRequired(), and setParent().

12.195.3.61 addFArray()

```
template<typename Type , int N>
FArrayParam< Type, N > & Util::ParamComposite::addFArray (
```

```
std::istream & in,
const char * label,
FArray< Type, N > & array,
bool isRequired = true ) [protected]
```

Add (but do not read) a FArray < Type, N > array parameter.

Parameters

in	input stream for reading
label	Label string for new array
array	FArray object
isRequired	Is this a required parameter?

Returns

reference to the new FArrayParam<Type, N> object

Definition at line 1370 of file ParamComposite.h.
References addComponent(), isRequired(), and setParent().

12.195.3.62 addCArray2D()

```
template<typename Type >
CArray2DParam< Type > & Util::ParamComposite::addCArray2D (
std::istream & in,
const char * label,
Type * value,
int m,
int n,
int np,
bool isRequired = true ) [protected]
```

Add (but do not read) a CArray2DParam < Type > 2D C-array.

Parameters

in	input stream for reading
label	Label string for new array
value	pointer to array
m	number of rows (1st dimension)
n	logical number of columns (2nd dimension)
np	physical number of columns (elements allocated per row)
isRequired	Is this a required parameter?

Returns

reference to the CArray2DParam<Type> object

Definition at line 1440 of file ParamComposite.h.
References addComponent(), isRequired(), and setParent().

12.195.3.63 addDMatrix()

```
template<typename Type >
DMatrixParam< Type > & Util::ParamComposite::addDMatrix (
```

```
std::istream & in,
const char * label,
DMatrix< Type > & matrix,
int m,
int n,
bool isRequired = true )   [protected]
```

Add and read a required DMatrix < Type > matrix parameter.

Parameters

in	input stream for reading
label	Label string for new array
matrix	DMatrix object
m	number of rows (1st dimension)
n	number of columns (2nd dimension)
isRequired	Is this a required parameter?

Returns

reference to the DMatrixParam<Type> object

Definition at line 1522 of file ParamComposite.h.
References addComponent(), isRequired(), and setParent().
The documentation for this class was generated from the following files:

- ParamComposite.h
- ParamComposite.cpp

12.196 Util::Parameter Class Reference

A single variable in a parameter file.
`#include <Parameter.h>`
Inheritance diagram for Util::Parameter:

classUtil_1_1Parameter-eps-converted-to.pdf

Public Member Functions

- Parameter (const char *label, bool isRequired=true)

    Constructor.
- virtual ~Parameter ()

    Destructor.
- virtual void readParam (std::istream &in)

    Read a label and (if the label matches) a parameter value.
- virtual void load (Serializable::IArchive &ar)

    Load from an archive.
- virtual void save (Serializable::OArchive &ar)

    Save to an archive.

- std::string label () const

    Return label string.

- bool isRequired () const

    Is this an optional parameter?

- bool isActive () const

    Is this parameter active?

Static Public Member Functions

- template<class Type >
  static void saveOptional (Serializable::OArchive &ar, Type &value, bool isActive)

    Save an optional parameter value to an output archive.

- template<class Type >
  static void saveOptionalCArray (Serializable::OArchive &ar, Type *ptr, int n, bool isActive)

    Save an optional C-array of n values to an output archive.

- template<class Type >
  static void saveOptionalCArray2D (Serializable::OArchive &ar, Type *ptr, int m, int n, int np, bool isActive)

    Save an optional two-dimensional C array to an output archive.

Static Public Attributes

- static const int Width = 20

    Width of output field for a scalar variable.

- static const int Precision = 12

    Precision for io of floating point data field.

Protected Member Functions

- virtual void readValue (std::istream &in)

    Read parameter value from an input stream.

- virtual void loadValue (Serializable::IArchive &ar)

    Load bare parameter value from an archive.

- virtual void saveValue (Serializable::OArchive &ar)

    Save parameter value to an archive.

- virtual void bcastValue ()

    Broadcast parameter value within the ioCommunicator.

Protected Attributes

- Label label_

    Label object that contains parameter label string.

- bool isActive_

    Is this parameter active (always true if isRequired).

Additional Inherited Members

12.196.1 Detailed Description

A single variable in a parameter file.

Parameter is a base class for objects that read and write the value of a single C++ variable from or to a parameter file. The parameter file format for a parameter contains a string label followed by a value for the variable. Different subclasses of parameter are specialized for different variable types, which can include primitive C/C++ variables, user defined types that overload the << and >> operators, or any of several different types of container.

A Parameter may be required or optional element in a parameter file, depending on the value of the bool isRequired parameter of the constructor. An optional element becomes "active" when an entry with the correct label is read from a parameter file, or when an active value is loaded from an archive. By convention, a required Parameter is always active, even before its value is read or loaded. The bool functions isRequired() and isActive() can be used to query the state of a Parameter.

The overloaded saveOptional() static member functions can be used to save optional parameters to an archive in a form that records whether or not they are active.

Definition at line 45 of file Parameter.h.

12.196.2 Constructor & Destructor Documentation

12.196.2.1 Parameter()

```
Util::Parameter::Parameter (
        const char * label,
        bool isRequired = true )
```

Constructor.

Parameters

label	label string preceding value in file format
isRequired	Is this a required parameter?

Definition at line 22 of file Parameter.cpp.

12.196.2.2 ∼Parameter()

```
Util::Parameter::~Parameter ( ) [virtual]
```

Destructor.

Definition at line 30 of file Parameter.cpp.

12.196.3 Member Function Documentation

12.196.3.1 saveOptional()

```
template<class Type >
void Util::Parameter::saveOptional (
        Serializable::OArchive & ar,
        Type & value,
        bool isActive ) [static]
```

Save an optional parameter value to an output archive.

Parameters

ar	output archive to which to save
value	reference to value of optional parameter

Parameters

isActive	Is this parameter present in the parameter file?

Definition at line 224 of file Parameter.h.
References isActive().

12.196.3.2 saveOptionalCArray() `template<class Type >`

```
void Util::Parameter::saveOptionalCArray (
            Serializable::OArchive & ar,
            Type * ptr,
            int n,
            bool isActive ) [static]
```

Save an optional C-array of n values to an output archive.

Parameters

ar	output archive to which to save
ptr	pointer to first element of optional C-array parameter
n	number of elements in array
isActive	Is this parameter present in the parameter file?

Definition at line 237 of file Parameter.h.
References isActive(), and Util::BinaryFileOArchive::pack().

12.196.3.3 saveOptionalCArray2D() `template<class Type >`

```
void Util::Parameter::saveOptionalCArray2D (
            Serializable::OArchive & ar,
            Type * ptr,
            int m,
            int n,
            int np,
            bool isActive ) [static]
```

Save an optional two-dimensional C array to an output archive.

Parameters

ar	output archive to which to save
ptr	pointer to first element optional 2D C-array parameter
m	logical number of rows in array
n	logical number of columns in array
np	logical number of columns in array
isActive	Is this parameter present in the parameter file?

Definition at line 250 of file Parameter.h.
References isActive(), and Util::BinaryFileOArchive::pack().

12.196.3.4 readParam() `void Util::Parameter::readParam (`

std::istream & *in*) [virtual]

Read a label and (if the label matches) a parameter value.

The parameter file format for a Parameter consists of a label string followed by value. The value is read if and only if the label matches the expected value for this Parameter. If this Parameter is required and the input label not match, an error message is printed to the log file and Exception is thrown. If the Parameter is not required and the input label does not match, the label string is retained in an buffer for later processing by the readParam method of other ParamComponent objects.

Upon entry to this function, a label string is read into a label buffer if and only if the buffer is empty. This buffer is a static member of the Label class, which can retain a label between invocations of the readParameter method of different ParamComponent objects. Once a label string is read from file, it remains in the label buffer until until it is matched, at which point the buffer is cleared to allow processing of the next label.

Parameters

in	input stream from which to read

Implements Util::ParamComponent.

Definition at line 36 of file Parameter.cpp.

References Util::bcast< bool >(), bcastValue(), Util::ParamComponent::echo(), Util::Log::file(), Util::MpiFileIo::hasIoCommunicator(), Util::ParamComponent::indent(), Util::MpiFileIo::ioCommunicator(), isActive_, Util::MpiFileIo::isIoProcessor(), Util::Label::isMatched(), isRequired(), label_, readValue(), UTIL_THROW, Width, and Util::ParamComponent::writeParam().

12.196.3.5 load() void Util::Parameter::load (Serializable::IArchive & *ar*) [virtual]

Load from an archive.

An optional Parameter loads the value of an isActive flag, and then loads the parameter value only if the isActive is true. A required Parameter simply loads the parameter value. The variable associated with an optional Parameter must be set to its default value before attempting to load the parameter. Optional parameters should be saved either using the save() method of an associated Parameter object or using the appropriate overloaded Parameter::saveOptional() static member function, which both use the required format.

Parameters

ar	input archive from which to load

Reimplemented from Util::ParamComponent.

Definition at line 98 of file Parameter.cpp.

References Util::bcast< bool >(), bcastValue(), Util::ParamComponent::echo(), Util::Log::file(), Util::MpiFileIo::hasIoCommunicator(), Util::ParamComponent::indent(), Util::MpiFileIo::ioCommunicator(), isActive_, Util::MpiFileIo::isIoProcessor(), isRequired(), label_, loadValue(), UTIL_THROW, Width, and Util::ParamComponent::writeParam().

Referenced by Util::ParamComposite::loadCArray(), Util::ParamComposite::loadCArray2D(), Util::ParamComposite::loadDArray(), Util::ParamComposite::loadDMatrix(), Util::ParamComposite::loadDSymmMatrix(), Util::ParamComposite::loadFArray(), and Util::ParamComposite::loadParameter().

12.196.3.6 save() void Util::Parameter::save (Serializable::OArchive & *ar*) [virtual]

Save to an archive.

An optional Parameter saves the value of the isActive flag, and then saves a parameter value only if the isActive is true. A required Parameter simply saves its value. The label string is not saved to the archive.

The overloaded static saveOptional functions can also be used to save optional parameter values in this format.

Parameters

ar	output archive to which to save

Reimplemented from Util::ParamComponent.
Definition at line 145 of file Parameter.cpp.
References isActive_, isRequired(), and saveValue().

12.196.3.7 label() `std::string Util::Parameter::label ( ) const`

Return label string.
Definition at line 158 of file Parameter.cpp.
References label_, and Util::Label::string().

12.196.3.8 isRequired() `bool Util::Parameter::isRequired ( ) const`

Is this an optional parameter?
Definition at line 164 of file Parameter.cpp.
References Util::Label::isRequired(), and label_.
Referenced by load(), readParam(), and save().

12.196.3.9 isActive() `bool Util::Parameter::isActive ( ) const`

Is this parameter active?
Definition at line 170 of file Parameter.cpp.
References isActive_.
Referenced by saveOptional(), saveOptionalCArray(), and saveOptionalCArray2D().

12.196.3.10 readValue() `virtual void Util::Parameter::readValue ( std::istream & in ) [inline], [protected], [virtual]`

Read parameter value from an input stream.

Parameters

in	input stream from which to read

Reimplemented in Util::ScalarParam< Type >, Util::CArray2DParam< Type >, Util::DMatrixParam< Type >, Util::DSymmMatrixParam< Type >, Util::FArrayParam< Type, N >, Util::DArrayParam< Type >, and Util::CArrayParam< Type >.
Definition at line 195 of file Parameter.h.
Referenced by readParam().

12.196.3.11 loadValue() `virtual void Util::Parameter::loadValue ( Serializable::IArchive & ar ) [inline], [protected], [virtual]`

Load bare parameter value from an archive.

Parameters

ar	input archive from which to load

Reimplemented in Util::ScalarParam< Type >, Util::CArray2DParam< Type >, Util::DMatrixParam< Type >,

Util::DSymmMatrixParam< Type >, Util::FArrayParam< Type, N >, Util::DArrayParam< Type >, and Util::CArrayParam< Type >.
Definition at line 202 of file Parameter.h.
Referenced by load().

12.196.3.12 saveValue() `virtual void Util::Parameter::saveValue ( Serializable::OArchive & ar ) [inline], [protected], [virtual]`

Save parameter value to an archive.

Parameters

ar	output archive to which to save

Reimplemented in Util::ScalarParam< Type >, Util::CArray2DParam< Type >, Util::DMatrixParam< Type >, Util::DSymmMatrixParam< Type >, Util::FArrayParam< Type, N >, Util::DArrayParam< Type >, and Util::CArrayParam< Type >.
Definition at line 209 of file Parameter.h.
Referenced by save().

12.196.3.13 bcastValue() `virtual void Util::Parameter::bcastValue ( ) [inline], [protected], [virtual]`

Broadcast parameter value within the ioCommunicator.
Reimplemented in Util::ScalarParam< Type >, Util::CArray2DParam< Type >, Util::DMatrixParam< Type >, Util::DSymmMatrixParam< Type >, Util::FArrayParam< Type, N >, Util::DArrayParam< Type >, and Util::CArrayParam< Type >.
Definition at line 215 of file Parameter.h.
Referenced by load(), and readParam().

12.196.4 Member Data Documentation

12.196.4.1 Width `const int Util::Parameter::Width = 20 [static]`

Width of output field for a scalar variable.
Definition at line 53 of file Parameter.h.
Referenced by load(), readParam(), Util::CArrayParam< Type >::writeParam(), Util::DArrayParam< Type >::writeParam(), Util::FArrayParam< Type, N >::writeParam(), Util::DSymmMatrixParam< Type >::writeParam(), Util::DMatrixParam< Type >::writeParam(), Util::ScalarParam< Type >::writeParam(), and Util::CArray2DParam< Type >::writeParam().

12.196.4.2 Precision `const int Util::Parameter::Precision = 12 [static]`

Precision for io of floating point data field.
Definition at line 56 of file Parameter.h.
Referenced by Util::CArrayParam< Type >::writeParam(), Util::DArrayParam< Type >::writeParam(), Util::FArrayParam< Type, N >::writeParam(), Util::DSymmMatrixParam< Type >::writeParam(), Util::DMatrixParam< Type >::writeParam(), Util::ScalarParam< Type >::writeParam(), and Util::CArray2DParam< Type >::writeParam().

12.196.4.3 label_ `Label Util::Parameter::label_ [protected]`

Label object that contains parameter label string.
Definition at line 185 of file Parameter.h.
Referenced by isRequired(), label(), load(), pscfpp.ParamComposite.Parameter::read(), readParam(), and pscfpp.ParamComposite.Parameter::setValue().

12.196.4.4 isActive_ `bool Util::Parameter::isActive_ [protected]`

Is this parameter active (always true if isRequired).

Definition at line 188 of file Parameter.h.

Referenced by isActive(), load(), readParam(), and save().

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

- Parameter.h
- Parameter.cpp

12.197 Util::PArray< Data > Class Template Reference

An array that only holds pointers to its elements.

`#include <PArray.h>`

Inheritance diagram for Util::PArray< Data >:

classUtil_1_1PArray-eps-converted-to.pdf

Public Member Functions

- virtual ~PArray ()

  Destructor.
- int capacity () const

  Return allocated size.
- int size () const

  Return logical size.
- void begin (PArrayIterator< Data > &iterator) const

  Set a PArrayIterator to the beginning of this PArray.
- void begin (ConstPArrayIterator< Data > &iterator) const

  Set a ConstPArrayIterator to the beginning of this PArray.
- Data & operator[ ] (int i) const

  Mimic C array subscripting.

Protected Member Functions

- PArray ()

  Constructor (protected to prevent instantiation).

Protected Attributes

- Data ∗∗ ptrs_

  PArray of of pointers to Data objects.
- int capacity_

  Allocated size of ptrs_ array.
- int size_

  Logical size (number of elements with initialized data).

12.197.1 Detailed Description

template<typename Data>
class Util::PArray< Data >

An array that only holds pointers to its elements.

A PArray<Data> is an array that is implemented by storing pointers to Data objects, rather than actual Data objects. The array suscript operator [] returns a reference to an associated Data object, as for Array<Data>. A PArray<Data> is not responsible for destroying the associated Data objects.

A PArray cannot be instantiated, because its constructor is protected. PArray is a base class for DPArray and for ArraySet.

Definition at line 33 of file PArray.h.

12.197.2 Constructor & Destructor Documentation

12.197.2.1 ∼PArray() `template<typename Data >`
`Util::PArray< Data >::∼PArray [virtual]`

Destructor.

Definition at line 123 of file PArray.h.

12.197.2.2 PArray() `template<typename Data >`
`Util::PArray< Data >::PArray [inline], [protected]`

Constructor (protected to prevent instantiation).

Definition at line 113 of file PArray.h.

12.197.3 Member Function Documentation

12.197.3.1 capacity() `template<typename Data >`
`int Util::PArray< Data >::capacity [inline]`

Return allocated size.

Returns

Number of elements allocated in array.

Definition at line 130 of file PArray.h.

12.197.3.2 size() `template<typename Data >`
`int Util::PArray< Data >::size [inline]`

Return logical size.

Returns

logical size of this array.

Definition at line 137 of file PArray.h.

12.197.3.3 begin() [1/2]

```
template<typename Data >
void Util::PArray< Data >::begin (
        PArrayIterator< Data > & iterator ) const [inline]
```

Set a PArrayIterator to the beginning of this PArray.
Set an PArrayIterator to the beginning of this PArray.

Parameters

iterator	PArrayIterator, initialized on output.

Definition at line 146 of file PArray.h.
References Util::PArrayIterator< Data >::setCurrent(), Util::PArrayIterator< Data >::setEnd(), and Util::PArrayIterator< Data >::setNull().

12.197.3.4 begin() [2/2]

```
template<typename Data >
void Util::PArray< Data >::begin (
        ConstPArrayIterator< Data > & iterator ) const [inline]
```

Set a ConstPArrayIterator to the beginning of this PArray.
Set an ConstPArrayIterator to the beginning of this PArray.

Parameters

iterator	PArrayIterator, initialized on output.
iterator	ConstPArrayIterator, initialized on output.

Definition at line 162 of file PArray.h.
References Util::ConstPArrayIterator< Data >::setCurrent(), Util::ConstPArrayIterator< Data >::setEnd(), and Util::↩ ConstPArrayIterator< Data >::setNull().

12.197.3.5 operator[]()

```
template<typename Data >
Data & Util::PArray< Data >::operator[] (
        int i ) const [inline]
```

Mimic C array subscripting.
Subscript - return a reference.

Parameters

i	array index

Returns

reference to element i

Definition at line 179 of file PArray.h.

12.197.4 Member Data Documentation

12.197.4.1 ptrs_

```
template<typename Data >
Data** Util::PArray< Data >::ptrs_ [protected]
```

PArray of of pointers to Data objects.
Definition at line 87 of file PArray.h.
Referenced by Util::DPArray< Data >::DPArray(), Util::GPArray< Data >::GPArray(), and Util::DPArray< Data >↩
::operator=().

12.197.4.2 capacity_ `template<typename Data >`

`int Util::PArray< Data >::capacity_ [protected]`

Allocated size of ptrs_ array.
Definition at line 90 of file PArray.h.
Referenced by Util::DPArray< Data >::DPArray(), and Util::GPArray< Data >::GPArray().

12.197.4.3 size_ `template<typename Data >`

`int Util::PArray< Data >::size_ [protected]`

Logical size (number of elements with initialized data).
Definition at line 93 of file PArray.h.
Referenced by Util::DPArray< Data >::DPArray(), Util::GPArray< Data >::GPArray(), Util::GPArray< Data >↩
::operator=(), and Util::DPArray< Data >::operator=().
The documentation for this class was generated from the following file:

- PArray.h

12.198 Util::PArrayIterator< Data > Class Template Reference

Forward iterator for a PArray.
`#include <PArrayIterator.h>`

Public Member Functions

- PArrayIterator ()
  Default constructor.
- void setCurrent (Data ∗∗ptr)
  Set the current pointer value.
- void setEnd (Data ∗∗ptr)
  Set the value of the end pointer.
- void setNull ()
  Nullify the iterator.
- bool isEnd () const
  Is the current pointer at the end of the PArray?
- bool notEnd () const
  Is the current pointer not at the end of the PArray?
- Data ∗ get () const
  Return a pointer to the current data.

Operators

- Data & operator∗ () const
  Return a reference to the current Data.
- Data ∗ operator-> () const
  Provide a pointer to the current Data object.
- PArrayIterator< Data > & operator++ ()
  Increment the current pointer.

12.198.1 Detailed Description

template<typename Data>
class Util::PArrayIterator< Data >

Forward iterator for a PArray.

An PArrayIterator is an abstraction of a pointer, similar to an STL forward iterator. The * operator returns a reference to an associated Data object, the -> operator returns a pointer to that object. The ++ operator increments the current pointer by one array element.

Unlike an STL forward iterator, an PArrayIterator contains the address of the end of the array. The isEnd() method can be used to test for termination of a for or while loop. When isEnd() is true, the iterator has no current value, and cannot be incremented further. The isEnd() method returns true either if the iterator: i) has already been incremented one past the end of an associated PArray, or ii) is in a null state that is produced by the constructor and the clear() method.

Definition at line 19 of file ArraySet.h.

12.198.2 Constructor & Destructor Documentation

12.198.2.1 PArrayIterator()

```
template<typename Data >
Util::PArrayIterator< Data >::PArrayIterator ( )  [inline]
```

Default constructor.

Constructs a null iterator.

Definition at line 44 of file PArrayIterator.h.

12.198.3 Member Function Documentation

12.198.3.1 setCurrent()

```
template<typename Data >
void Util::PArrayIterator< Data >::setCurrent (
             Data ** ptr )  [inline]
```

Set the current pointer value.

Parameters

ptr	Pointer to current element of array of Data* pointers.

Definition at line 55 of file PArrayIterator.h.

Referenced by Util::PArray< Data >::begin(), Util::FPArray< Data, Capacity >::begin(), and Util::SSet< Data, Capacity >::begin().

12.198.3.2 setEnd()

```
template<typename Data >
void Util::PArrayIterator< Data >::setEnd (
             Data ** ptr )  [inline]
```

Set the value of the end pointer.

Parameters

ptr	Pointer to one element past end of array of Data* pointers.

Definition at line 66 of file PArrayIterator.h.
Referenced by Util::PArray< Data >::begin(), Util::FPArray< Data, Capacity >::begin(), and Util::SSet< Data, Capacity >::begin().

12.198.3.3 setNull() `template<typename Data >`

`void Util::PArrayIterator< Data >::setNull ( ) [inline]`
Nullify the iterator.
Definition at line 72 of file PArrayIterator.h.
Referenced by Util::PArray< Data >::begin().

12.198.3.4 isEnd() `template<typename Data >`

`bool Util::PArrayIterator< Data >::isEnd ( ) const [inline]`
Is the current pointer at the end of the PArray?

Returns

true if at end, false otherwise.

Definition at line 84 of file PArrayIterator.h.

12.198.3.5 notEnd() `template<typename Data >`

`bool Util::PArrayIterator< Data >::notEnd ( ) const [inline]`
Is the current pointer not at the end of the PArray?

Returns

true if not at end, false otherwise.

Definition at line 92 of file PArrayIterator.h.

12.198.3.6 get() `template<typename Data >`

`Data* Util::PArrayIterator< Data >::get ( ) const [inline]`
Return a pointer to the current data.

Returns

true if at end, false otherwise.

Definition at line 100 of file PArrayIterator.h.

12.198.3.7 operator*() `template<typename Data >`

`Data& Util::PArrayIterator< Data >::operator* ( ) const [inline]`
Return a reference to the current Data.

Returns

reference to associated Data object

Definition at line 111 of file PArrayIterator.h.

12.198.3.8 operator->() template<typename Data >

Data* Util::PArrayIterator< Data >::operator-> ( ) const [inline]

Provide a pointer to the current Data object.

Returns

pointer to the Data object

Definition at line 122 of file PArrayIterator.h.

12.198.3.9 operator++() template<typename Data >

PArrayIterator<Data>& Util::PArrayIterator< Data >::operator++ ( ) [inline]

Increment the current pointer.

Returns

this PArrayIterator, after modification.

Definition at line 133 of file PArrayIterator.h.

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

- ArraySet.h
- PArrayIterator.h

12.199 Util::Polynomial< T > Class Template Reference

A Polynomial (i.e,.

#include <Polynomial.h>

Inheritance diagram for Util::Polynomial< T >:

classUtil_1_1Polynomial-eps-converted-to.pdf

Public Member Functions

Constructors, Destructors, and Assignment

- Polynomial (int capacity=10)
  Construct a zero polynomial.
- Polynomial (T c)
  Construct a constant polynomial.
- Polynomial (Array< T > const &coeffs)
  Construct a polynomial from array of coefficients.
- Polynomial (Polynomial< T > const &other)
  Copy constructor.
- template<typename U >
  Polynomial< T > & operator= (Polynomial< U > const &other)
  Assignment from another polynomial.
- void setToZero ()
  Assign this polynomial a value of zero.

Simple Accessors

- int degree () const

  Return degree of polynomial.

Arithmetic Assignment Operators

- Polynomial< T > & operator+= (const Polynomial< T > &a)

  Add another polynomial to this one.
- Polynomial< T > & operator+= (T a)

  Add a constant to this polynomial.
- Polynomial< T > & operator-= (const Polynomial< T > &a)

  Subtract another polynomial from this one.
- Polynomial< T > & operator-= (T a)

  Subtract a constant from this polynomial.
- Polynomial< T > & operator*= (T a)

  Multiply this polynomial by a scalar.
- Polynomial< T > & operator/= (T a)

  Divide this polynomial by a scalar.
- Polynomial< T > & operator*= (const Polynomial< T > &a)

  Multiply this polynomial by another.

Mathematical Functions (return polynomials)

- Polynomial< T > integrate () const

  Compute and return indefinite integral of this polynomial.
- Polynomial< T > differentiate () const

  Compute and return derivative of this polynomial.
- Polynomial< T > reflect () const

  Compute and return reflected polynomial f(-x).
- Polynomial< T > shift (T a) const

  Compute and return shifted polynomial f(x+a).

Polynomial Evaluation Functions

- T operator() (T x) const

  Evaluate polynomial at specific argument of type T.
- double evaluate (double x) const

  Evaluate polynomial at specific floating point argument.
- static Polynomial< T > monomial (int n)

  Return a monomial f(x) = x^{n}.

12.199.1 Detailed Description

template< typename T = Rational >
class Util::Polynomial< T >

A Polynomial (i.e,.
a list of coefficents).
Definition at line 30 of file Polynomial.h.

12.199.2 Constructor & Destructor Documentation

12.199.2.1 Polynomial() [1/4]

template<typename T >

Util::Polynomial< T >::Polynomial (
    int *capacity* = *10*) [inline]

Construct a zero polynomial.

Creates a zero polynomial f(x) = 0, with no stored coefficients. The capacity parameter specifies how much physical space to allocate for subsqequent growth in the array of coefficients.

Parameters

capacity	initial capacity of coefficient array.

Definition at line 264 of file Polynomial.h.

12.199.2.2 Polynomial() [2/4]

template<typename T >

Util::Polynomial< T >::Polynomial (
    T *c*) [inline], [explicit]

Construct a constant polynomial.

Creates a polynomial f(x) = c, with degree() = 0.

Parameters

c	constant coefficient value

Definition at line 272 of file Polynomial.h.

12.199.2.3 Polynomial() [3/4]

template<typename T >

Util::Polynomial< T >::Polynomial (
    Array< T > const & *coeffs*) [inline]

Construct a polynomial from array of coefficients.

Constructs a polynomial in which the coefficient of x^{i} is given by coeffs[i]. The logical and physical size of the coefficient array are both set to the capacity of coeffs.

Parameters

coeffs	array of coefficients.

Definition at line 283 of file Polynomial.h.

12.199.2.4 Polynomial() [4/4]

template<typename T >

Util::Polynomial< T >::Polynomial (
    Polynomial< T > const & *other*) [inline]

Copy constructor.

Parameters

other	Polynomial to be copied

Definition at line 298 of file Polynomial.h.

12.199.3 Member Function Documentation

12.199.3.1 operator=()

```
template<typename T >
template<typename U >
Polynomial< T > & Util::Polynomial< T >::operator= (
            Polynomial< U > const & other )  [inline]
```

Assignment from another polynomial.

Parameters

other	Polynomial to assign.

Definition at line 314 of file Polynomial.h.

12.199.3.2 setToZero()

```
template<typename T >
void Util::Polynomial< T >::setToZero  [inline]
```

Assign this polynomial a value of zero.
Equivalent to GArray::clear(): Clears all coefficients, setting size = 0 and degree = -1.
Definition at line 335 of file Polynomial.h.
Referenced by Util::Polynomial< double >::differentiate(), and Util::Polynomial< double >::integrate().

12.199.3.3 degree()

```
template<typename T >
int Util::Polynomial< T >::degree  [inline]
```

Return degree of polynomial.
Returns size() - 1, number of coefficients - 1. By convention, a zero polynomial has degree = -1.
Definition at line 343 of file Polynomial.h.

12.199.3.4 operator+=() [1/2]

```
template<typename T >
Polynomial< T > & Util::Polynomial< T >::operator+= (
            const Polynomial< T > & a )
```

Add another polynomial to this one.
Upon return, $*$this = this + a.

Parameters

a	increment (input)

Definition at line 350 of file Polynomial.h.

12.199.3.5 operator+=() [2/2]

```
template<typename T >
Polynomial< T > & Util::Polynomial< T >::operator+= (
            T a )
```

Add a constant to this polynomial.
Upon return, $*$this = this + a.

Parameters

a	increment (input)

Definition at line 373 of file Polynomial.h.

12.199.3.6 operator-=() [1/2]

```
template<typename T >
Polynomial< T > & Util::Polynomial< T >::operator-= (
            const Polynomial< T > & a )
```

Subtract another polynomial from this one.
Upon return, ∗this = this + a.

Parameters

a	decrement (input)

Definition at line 387 of file Polynomial.h.

12.199.3.7 operator-=() [2/2]

```
template<typename T >
Polynomial< T > & Util::Polynomial< T >::operator-= (
            T a )
```

Subtract a constant from this polynomial.
Upon return, ∗this = this + a.

Parameters

a	increment (input)

Definition at line 410 of file Polynomial.h.

12.199.3.8 operator∗=() [1/2]

```
template<typename T >
Polynomial< T > & Util::Polynomial< T >::operator*= (
            T a ) [inline]
```

Multiply this polynomial by a scalar.
Upon return, ∗this = this∗a.

Parameters

a	scalar factor

Definition at line 425 of file Polynomial.h.

12.199.3.9 operator/=()

```
template<typename T >
Polynomial< T > & Util::Polynomial< T >::operator/= (
            T a ) [inline]
```

Divide this polynomial by a scalar.
Upon return, ∗this = this∗a.

Parameters

a	scalar factor (input)

Definition at line 440 of file Polynomial.h.

12.199.3.10 operator*=() [2/2] `template<typename T >`

```
Polynomial< T > & Util::Polynomial< T >::operator*= (
            const Polynomial< T > & a )
```

Multiply this polynomial by another.
Upon return, *this = this*a.

Parameters

a	increment (input)

Definition at line 454 of file Polynomial.h.

12.199.3.11 integrate() `template<typename T >`

```
Polynomial< T > Util::Polynomial< T >::integrate
```

Compute and return indefinite integral of this polynomial.
Returns an indefinite integral with zero constant term.

Returns

indefinite integral polynomial.

Definition at line 502 of file Polynomial.h.

12.199.3.12 differentiate() `template<typename T >`

```
Polynomial< T > Util::Polynomial< T >::differentiate
```

Compute and return derivative of this polynomial.
Returns a polynomial of one smaller degree.

Returns

derivative polynomial

Definition at line 528 of file Polynomial.h.

12.199.3.13 reflect() `template<typename T >`

```
Polynomial< T > Util::Polynomial< T >::reflect
```

Compute and return reflected polynomial f(-x).
If this polynomial is f(x), this returns a polynomial g(x) = f(-x) created by the reflection operation x-> -x. This yields a polynomial in which the sign is reversed for all coefficients of odd powers of x.

Returns

polynomial created by reflection x -> -x.

Definition at line 557 of file Polynomial.h.

12.199.3.14 shift() `template<typename T >`

`Polynomial< T > Util::Polynomial< T >::shift (`
`T a ) const`

Compute and return shifted polynomial f(x+a).

If this polynomial is f(x), this returns a polynomial g(x) = f(x+a) created by the shift operation x-> x + a.

Returns

polynomial created by shift operation x -> x + a.

Definition at line 576 of file Polynomial.h.

12.199.3.15 operator()() `template<typename T >`

`T Util::Polynomial< T >::operator() (`
`T x ) const [inline]`

Evaluate polynomial at specific argument of type T.

Parameters

x	value of argument

Returns

Value f(x) of this polynomial at specified x

Definition at line 602 of file Polynomial.h.

12.199.3.16 evaluate() `template<typename T >`

`double Util::Polynomial< T >::evaluate (`
`double x ) const [inline]`

Evaluate polynomial at specific floating point argument.

Parameters

x	value of argument x

Returns

Value f(x) of polynomial at specified x

Definition at line 623 of file Polynomial.h.

12.199.3.17 monomial() `template<typename T >`

`Polynomial< T > Util::Polynomial< T >::monomial (`
`int n ) [static]`

Return a monomial f(x) = x^{n}.

Return a monomial.

Parameters

n	power of x in monomial.

Definition at line 646 of file Polynomial.h.
The documentation for this class was generated from the following file:

- Polynomial.h

12.200 Util::RadialDistribution Class Reference

Distribution (or histogram) of values for particle separations.
`#include <RadialDistribution.h>`
Inheritance diagram for Util::RadialDistribution:

classUtil_1_1RadialDistribution-eps-converted-to.pdf

Public Member Functions

- RadialDistribution ()

  Default constructor.
- RadialDistribution (const RadialDistribution &other)

  Copy constructor.
- RadialDistribution & operator= (const RadialDistribution &other)

  Assignment.
- virtual void readParameters (std::istream &in)

  Read values of min, max, and nBin from file.
- void setParam (double max, int nBin)

  Set parameters and initialize.
- virtual void loadParameters (Serializable::IArchive &ar)

  Load internal state from an archive.
- virtual void save (Serializable::OArchive &ar)

  Save internal state to an archive.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize this RadialDistribution to/from an archive.
- virtual void clear ()

  Clear all accumulators.
- void beginSnapshot ()

  Mark the beginning of a "snapshot" (i.e., a sampled time step).
- void setNorm (double norm)

  Set the factor used to normalize the RDF before output.
- void setOutputIntegral (bool outputIntegral)

Set true to enable output of spatial integral of g(r).

- void output (std::ostream &out)

  Output the distribution to file.

- long nSnapshot ()

  Get number of snapshots.

Additional Inherited Members

12.200.1 Detailed Description

Distribution (or histogram) of values for particle separations.
Definition at line 21 of file RadialDistribution.h.

12.200.2 Constructor & Destructor Documentation

12.200.2.1 RadialDistribution() [1/2] `Util::RadialDistribution::RadialDistribution ( )`

Default constructor.
Definition at line 19 of file RadialDistribution.cpp.
References Util::ParamComposite::setClassName().

12.200.2.2 RadialDistribution() [2/2] `Util::RadialDistribution::RadialDistribution ( const RadialDistribution & other )`

Copy constructor.

Parameters

other	object to be copied.

Definition at line 29 of file RadialDistribution.cpp.

12.200.3 Member Function Documentation

12.200.3.1 operator=() `RadialDistribution & Util::RadialDistribution::operator= ( const RadialDistribution & other )`

Assignment.

Parameters

other	object to be assigned.

Definition at line 40 of file RadialDistribution.cpp.
References Util::Distribution::operator=().

12.200.3.2 readParameters() `void Util::RadialDistribution::readParameters ( std::istream & in ) [virtual]`

Read values of min, max, and nBin from file.

Parameters

in	input parameter file stream.

Reimplemented from Util::Distribution.
Definition at line 55 of file RadialDistribution.cpp.
References Util::DArray< Data >::allocate(), Util::Distribution::binWidth_, clear(), Util::Distribution::histogram_, Util::Distribution::max_, Util::Distribution::min_, and Util::Distribution::nBin_.

12.200.3.3 setParam()

```
void Util::RadialDistribution::setParam (
    double max,
    int nBin )
```

Set parameters and initialize.

Parameters

max	upper bound of range
nBin	number of bins in range [min, max]

Definition at line 68 of file RadialDistribution.cpp.
References Util::DArray< Data >::allocate(), Util::Distribution::binWidth_, clear(), Util::Distribution::histogram_, Util::Distribution::max(), Util::Distribution::max_, Util::Distribution::min_, Util::Distribution::nBin(), and Util::Distribution::nBin_.

12.200.3.4 loadParameters()

```
void Util::RadialDistribution::loadParameters (
    Serializable::IArchive & ar ) [virtual]
```

Load internal state from an archive.

Parameters

ar	input/loading archive

Reimplemented from Util::Distribution.
Definition at line 81 of file RadialDistribution.cpp.
References Util::Distribution::binWidth_, Util::Array< Data >::capacity(), Util::feq(), Util::Distribution::histogram_, Util::Distribution::max_, Util::Distribution::min_, Util::Distribution::nBin_, Util::Distribution::nReject_, Util::Distribution::nSample_, and UTIL_THROW.

12.200.3.5 save()

```
void Util::RadialDistribution::save (
    Serializable::OArchive & ar ) [virtual]
```

Save internal state to an archive.

Parameters

ar	output/saving archive

Reimplemented from Util::Distribution.
Definition at line 104 of file RadialDistribution.cpp.

12.200.3.6 serialize() `template<class Archive >`

```
void Util::RadialDistribution::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize this RadialDistribution to/from an archive.

Parameters

ar	input or output archive
version	file version id

Definition at line 160 of file RadialDistribution.h.
References Util::Distribution::serialize().

12.200.3.7 clear() `void Util::RadialDistribution::clear ( ) [virtual]`

Clear all accumulators.
Reimplemented from Util::Distribution.
Definition at line 110 of file RadialDistribution.cpp.
References Util::Distribution::clear().
Referenced by readParameters(), and setParam().

12.200.3.8 beginSnapshot() `void Util::RadialDistribution::beginSnapshot ( )`

Mark the beginning of a "snapshot" (i.e., a sampled time step).
Definition at line 125 of file RadialDistribution.cpp.

12.200.3.9 setNorm() `void Util::RadialDistribution::setNorm (`

```
            double norm )
```

Set the factor used to normalize the RDF before output.

Parameters

norm	normalizing factor

Definition at line 119 of file RadialDistribution.cpp.

12.200.3.10 setOutputIntegral() `void Util::RadialDistribution::setOutputIntegral (`

```
            bool outputIntegral )
```

Set true to enable output of spatial integral of g(r).

Parameters

outputIntegral	true to enable output of integral.

Definition at line 131 of file RadialDistribution.cpp.

12.200.3.11 output() `void Util::RadialDistribution::output (`

```
            std::ostream & out )
```

Output the distribution to file.

Parameters

out	pointer to output file

Definition at line 137 of file RadialDistribution.cpp.
References Util::Distribution::binWidth_, Util::Distribution::histogram_, and Util::Distribution::nBin_.

12.200.3.12 nSnapshot() `long Util::RadialDistribution::nSnapshot ( ) [inline]`

Get number of snapshots.
Definition at line 153 of file RadialDistribution.h.
The documentation for this class was generated from the following files:

- RadialDistribution.h
- RadialDistribution.cpp

12.201 Util::RaggedMatrix< Data > Class Template Reference

A 2D array in which different rows can have different lengths.
`#include <RaggedMatrix.h>`
Inheritance diagram for Util::RaggedMatrix< Data >:

classUtil_1_1RaggedMatrix-eps-converted-to.pdf

Public Member Functions

- virtual ~RaggedMatrix ()

  Destructor.
- int capacity1 ()

  Get number of rows.
- int capacity2 (int i)

  Get number of elements in row number i.
- const Data & operator() (int i, int j) const

  Return element (i,j) of matrix by const reference.
- Data & operator() (int i, int j)

  Return element (i,j) of matrix by reference.

Protected Member Functions

- RaggedMatrix ()

  Default constructor.

Protected Attributes

- Data * data_

    One-dimensional C array of all elements.
- Data ** rows_

    Array of pointers to rows.
- int * capacity2_

    Array containing number of elements in each row.
- int capacity1_

    Number of rows (range of first index).
- int capacity_

    Total number of elements.

12.201.1 Detailed Description

template<typename Data>
class Util::RaggedMatrix< Data >

A 2D array in which different rows can have different lengths.
A RaggedMatrix object A is a two-dimensional array in which the operator A(i,j) returns a reference to element j of row i, and in which different rows have different lengths. Class RaggedMatrix cannot be instantiated, and functions like an abstract base class.
The memory for a RaggedMatrix is stored in a one-dimensional C array.
Definition at line 29 of file RaggedMatrix.h.

12.201.2 Constructor & Destructor Documentation

12.201.2.1 ~RaggedMatrix() `template<typename Data >`
`Util::RaggedMatrix< Data >::~RaggedMatrix [virtual]`
Destructor.
Definition at line 129 of file RaggedMatrix.h.

12.201.2.2 RaggedMatrix() `template<typename Data >`
`Util::RaggedMatrix< Data >::RaggedMatrix [inline], [protected]`
Default constructor.
Constructor (protected).
Protected to prevent direct instantiation.
Definition at line 117 of file RaggedMatrix.h.

12.201.3 Member Function Documentation

12.201.3.1 capacity1() `template<typename Data >`
`int Util::RaggedMatrix< Data >::capacity1 [inline]`
Get number of rows.

Returns

Number of rows (i.e., range of first array index)

Definition at line 136 of file RaggedMatrix.h.

12.201.3.2 capacity2() `template<typename Data >`

```
int Util::RaggedMatrix< Data >::capacity2 (
            int i ) [inline]
```

Get number of elements in row number i.

Parameters

i	row index

Returns

Number of elements in row i.

Definition at line 143 of file RaggedMatrix.h.

12.201.3.3 operator()() [1/2] `template<typename Data >`

```
const Data & Util::RaggedMatrix< Data >::operator() (
            int i,
            int j ) const [inline]
```

Return element (i,j) of matrix by const reference.

Parameters

i	row index.
j	column index.

Returns

element (i, j)

Definition at line 150 of file RaggedMatrix.h.

12.201.3.4 operator()() [2/2] `template<typename Data >`

```
Data & Util::RaggedMatrix< Data >::operator() (
            int i,
            int j ) [inline]
```

Return element (i,j) of matrix by reference.

Parameters

i	row index.
j	column index.

Returns

element (i, j)

Definition at line 164 of file RaggedMatrix.h.

12.201.4 Member Data Documentation

12.201.4.1 data_ template<typename Data >

Data* Util::RaggedMatrix< Data >::data_ [protected]

One-dimensional C array of all elements.

Definition at line 79 of file RaggedMatrix.h.

12.201.4.2 rows_ template<typename Data >

Data** Util::RaggedMatrix< Data >::rows_ [protected]

Array of pointers to rows.

Definition at line 82 of file RaggedMatrix.h.

12.201.4.3 capacity2_ template<typename Data >

int* Util::RaggedMatrix< Data >::capacity2_ [protected]

Array containing number of elements in each row.

Definition at line 85 of file RaggedMatrix.h.

12.201.4.4 capacity1_ template<typename Data >

int Util::RaggedMatrix< Data >::capacity1_ [protected]

Number of rows (range of first index).

Definition at line 88 of file RaggedMatrix.h.

12.201.4.5 capacity_ template<typename Data >

int Util::RaggedMatrix< Data >::capacity_ [protected]

Total number of elements.

Definition at line 91 of file RaggedMatrix.h.

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

- RaggedMatrix.h

12.202 Util::Random Class Reference

Random number generator.

#include <Random.h>

Inheritance diagram for Util::Random:

classUtil_1_1Random-eps-converted-to.pdf

Public Member Functions

- Random ()

  Constructor.

- virtual ~Random ()

  Destructor.
- virtual void readParameters (std::istream &in)

  Read seed from file, initialize RNG.
- virtual void loadParameters (Serializable::IArchive &ar)

  Load internal state from file.
- virtual void save (Serializable::OArchive &ar)

  Save internal state to file.
- void setSeed (SeedType seed)

  Sets of random seed, and initializes random number generator.
- double uniform ()

  Return a random floating point number x, uniformly distributed in the range 0 <= x < 1.
- double uniform (double range1, double range2)

  Return a random floating point number x, uniformly distributed in the range range1 <= x < range2.
- long uniformInt (long range1, long range2)

  Return random long int x uniformly distributed in range1 <= x < range2.
- void getPoint (double minR[ ], double maxR[ ], double r[ ])

  Generate a random point in a box.
- double gaussian (void)

  Return a Gaussian random number with zero average and unit variance.
- void unitVector (Vector &v)

  Generate unit vector with uniform probability over the unit sphere.
- bool metropolis (double ratio)

  Metropolis algorithm for whether to accept a MC move.
- long drawFrom (double probability[ ], long size)

  Choose one of several outcomes with a specified set of probabilities.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize to/from an archive.
- long seed ()

  Returns value of random seed (private member variable seed_).

Additional Inherited Members

12.202.1 Detailed Description

Random number generator.
This class provides functions that return several forms of random numbers, using an internal Mersenne-Twister random number generator.

The generator may be seeded either by reading a seed from file, using the readParam() method, or by using setSeed() to set or reset it explicitly. In either case, inputting a positive integer causes that value to be used as a seed, but inputting a value of 0 causes the use of a seed that is generated from the system clock.
If the program is compiled with MPI, and MPI is initialized, then any automatically generated seed is also offset by a value that depends on the rank of the processor within the MPI world communicator, so that different processor use different seeds.
Definition at line 46 of file Random.h.

12.202.2 Constructor & Destructor Documentation

12.202.2.1 Random() `Util::Random::Random ( )`

Constructor.
Definition at line 13 of file Random.cpp.
References Util::ParamComposite::setClassName().

12.202.2.2 ∼Random() `Util::Random::~Random ( ) [virtual]`

Destructor.
Definition at line 22 of file Random.cpp.

12.202.3 Member Function Documentation

12.202.3.1 readParameters() `void Util::Random::readParameters ( std::istream & in ) [virtual]`

Read seed from file, initialize RNG.

Parameters

in	input stream.

Reimplemented from Util::ParamComposite.
Definition at line 28 of file Random.cpp.
References setSeed().

12.202.3.2 loadParameters() `void Util::Random::loadParameters ( Serializable::IArchive & ar ) [virtual]`

Load internal state from file.

Parameters

ar	input/loading archive

Reimplemented from Util::ParamComposite.
Definition at line 37 of file Random.cpp.

12.202.3.3 save() `void Util::Random::save ( Serializable::OArchive & ar ) [virtual]`

Save internal state to file.

Parameters

ar	output/saving archive

Reimplemented from Util::ParamComposite.
Definition at line 46 of file Random.cpp.

12.202.3.4 setSeed() `void Util::Random::setSeed (`

`Random::SeedType seed )`

Sets of random seed, and initializes random number generator.

Parameters

seed	value for random seed (private member variable idum)

Definition at line 57 of file Random.cpp.
References seed(), and setSeed().
Referenced by readParameters(), and setSeed().

12.202.3.5 uniform() [1/2] `double Util::Random::uniform ( ) [inline]`

Return a random floating point number x, uniformly distributed in the range 0 <= x < 1.

Returns

random double precision number

Definition at line 203 of file Random.h.
Referenced by drawFrom(), gaussian(), getPoint(), metropolis(), uniformInt(), and unitVector().

12.202.3.6 uniform() [2/2] `double Util::Random::uniform (`

```
double range1,
double range2 ) [inline]
```

Return a random floating point number x, uniformly distributed in the range range1 <= x < range2.

Returns

random double precision number

Definition at line 212 of file Random.h.

12.202.3.7 uniformInt() `long Util::Random::uniformInt (`

```
long range1,
long range2 ) [inline]
```

Return random long int x uniformly distributed in range1 <= x < range2.
Parameters range1 and range2 must be within the range of long integers.

Returns

random integer

Definition at line 224 of file Random.h.
References uniform().

12.202.3.8 getPoint() `void Util::Random::getPoint (`

```
double minR[],
double maxR[],
double r[] ) [inline]
```

Generate a random point in a box.

Parameters

minR[ ]	array of minimum coordinate values along three axes
maxR[ ]	array of maximum coordinate values along three axes
r[ ]	random position such that minR[axis] $<$ r[axis] $<$ maxR[axis]

Definition at line 251 of file Random.h.
References uniform().

12.202.3.9 gaussian()

```
double Util::Random::gaussian (
    void )
```

Return a Gaussian random number with zero average and unit variance.

Returns

Gaussian distributed random number.

Definition at line 92 of file Random.cpp.
References uniform().
Referenced by Util::Ar1Process::init(), and Util::Ar1Process::operator()().

12.202.3.10 unitVector()

```
void Util::Random::unitVector (
    Vector & v )
```

Generate unit vector with uniform probability over the unit sphere.

Parameters

v	random unit vector (upon return)

Definition at line 122 of file Random.cpp.
References uniform().

12.202.3.11 metropolis()

```
bool Util::Random::metropolis (
    double ratio ) [inline]
```

Metropolis algorithm for whether to accept a MC move.
If ratio $>$ 1, this function return true. If 0 $<$ ratio $<$ 1, this function returns true with probability ratio, and false with probability 1 - ratio.

Parameters

ratio	ratio of old to new equilibrium weights

Returns

true if accepted, false if rejected

Definition at line 260 of file Random.h.
References uniform().

12.202.3.12 drawFrom()

```
long Util::Random::drawFrom (
```

```
double probability[ ],
long size )  [inline]
```

Choose one of several outcomes with a specified set of probabilities.
Precondition: Elements of probability array must add to 1.0

Parameters

probability[ ]	array of probabilities, for indices 0,...,size-1
size	number of options

Returns

random integer index of element of probability[] array

Definition at line 237 of file Random.h.
References uniform().

12.202.3.13 serialize() `template<class Archive >`

```
void Util::Random::serialize (
Archive & ar,
const unsigned int version )
```

Serialize to/from an archive.
Definition at line 284 of file Random.h.

12.202.3.14 seed() `long Util::Random::seed ( )  [inline]`

Returns value of random seed (private member variable seed_).

Returns

value of random number generator seed.

Definition at line 277 of file Random.h.
Referenced by setSeed().
The documentation for this class was generated from the following files:

- Random.h
- Random.cpp

12.203 Util::RArray< Data > Class Template Reference

An Array that acts as a reference to another Array or C array.
`#include <RArray.h>`
Inheritance diagram for Util::RArray< Data >:

classUtil_1_1RArray-eps-converted-to.pdf

Public Member Functions

- RArray ()
  Constructor.
- RArray (const RArray< Data > &other)
  Copy constructor.
- void associate (Array< Data > &array)
  Associate this RArray with an existing Array object.
- void associate (Data *array, int capacity)
  Associate this RArray with an existing C array.

Additional Inherited Members

12.203.1 Detailed Description

template<typename Data>
class Util::RArray< Data >

An Array that acts as a reference to another Array or C array.
An RArray is associated with a "target" DArray or C array by the associate() method. The RArray and its target array then wrap the same underlying C array, and so access the same data. The associate() method simply copies the address and capacity of a C array. An RArray can be associated only once, after which it can be safely used as an alias for its target.
An RArray can only be associated with a DArray after the target DArray has been allocated. Because a DArray can be allocated only once, this association cannot be corrupted by re-allocation or re-sizing of the target DArray.
An RArray can be created from another RArray only after the target RArray has already been associated with some other Array.
An RArray differs from a C++ reference to an Array because a C++ reference must be initialized when it is instantiated, whereas an RArray is associated after it is instantiated. Because association is implemented by copying the address and capacity of a shared C array, access through an RArray should be exactly as efficient as access through a DArray.
Definition at line 46 of file RArray.h.

12.203.2 Constructor & Destructor Documentation

12.203.2.1 RArray() [1/2]

```
template<typename Data >
Util::RArray< Data >::RArray ( )  [inline]
```

Constructor.
Definition at line 57 of file RArray.h.

12.203.2.2 RArray() [2/2]

```
template<typename Data >
Util::RArray< Data >::RArray (
              const RArray< Data > & other )  [inline]
```

Copy constructor.
Shallow copy of another RArray

Parameters

other	another RArray<Data> for which this is an alias.

Definition at line 68 of file RArray.h.
References Util::Array< Data >::capacity_, and Util::Array< Data >::data_.

12.203.3 Member Function Documentation

12.203.3.1 associate() [1/2] template<typename Data >

```
void Util::RArray< Data >::associate (
            Array< Data > & array )  [inline]
```

Associate this RArray with an existing Array object.

The target (i.e., the parameter array) must be allocated when this method is invoked, as discussed in the RArray class documentation.

Parameters

array	the target Array

Definition at line 83 of file RArray.h.

References Util::Array< Data >::capacity(), Util::Array< Data >::capacity_, Util::Array< Data >::data_, and UTIL_T↩ HROW.

12.203.3.2 associate() [2/2] template<typename Data >

```
void Util::RArray< Data >::associate (
            Data * array,
            int capacity )  [inline]
```

Associate this RArray with an existing C array.

Parameters

array	the target C array
capacity	the number of elements in the target array

Definition at line 101 of file RArray.h.

References Util::Array< Data >::capacity(), Util::Array< Data >::capacity_, Util::Array< Data >::data_, and UTIL_T↩ HROW.

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

- RArray.h

12.204 Util::Rational Class Reference

A Rational number (a ratio of integers).

`#include <Rational.h>`

Public Member Functions

Constructors

- Rational ()
  Default constructor.
- Rational (int num, int den)
  Constructor, explicit numerator and denominator.
- Rational (int number)
  Constructor, construct from integer.
- Rational (Rational const &v)

Copy constructor.
- ∼Rational ()

  Destructor.

Assignment and Conversion.

- Rational & operator= (Rational const &other)

  Copy assignment from another Rational.
- Rational & operator= (int other)

  Assignment from integer.

Arithmetic Assignment Operators

- Rational & operator+= (Rational const &a)

  Add another rational to this one.
- Rational & operator+= (int a)

  Add an integer to this rational.
- Rational & operator-= (Rational const &a)

  Subtract another rational from this one.
- Rational & operator-= (int)

  Subtract an integer from this rational.
- Rational & operator∗= (Rational const &a)

  Multiply this rational by another.
- Rational & operator∗= (int a)

  Multiply this rational by an integer.
- Rational & operator/= (Rational const &a)

  Divide this rational by another.
- Rational & operator/= (int a)

  Divide this rational by an integer.

Accessors

- int num () const

  Return numerator.
- int den () const

  Return denominator.
- operator double () const

  Cast (convert) to double precision floating point.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize to/from an archive.
- static void commitMpiType ()

  Commit MPI datatype MpiTraits<Rational>::type.
- Rational operator+ (Rational const &a, Rational const &b)

  Compute sum of two rationals.
- Rational operator+ (Rational const &a, int b)

  Compute sum of rational and integer.
- Rational operator- (Rational const &a, Rational const &b)

  Compute difference of rationals.
- Rational operator- (Rational const &a, int b)

  Compute difference of rational and integer.
- Rational operator- (int b, Rational const &a)

Compute difference of integer and rational.

- Rational operator* (Rational const &a, Rational const &b)

  Compute product of rationals.
- Rational operator* (Rational const &a, int b)

  Compute product of rational and integer.
- Rational operator/ (Rational const &a, Rational const &b)

  Compute quotient of two rationals.
- Rational operator/ (Rational const &a, int b)

  Compute quotient Rational divided by integer.
- Rational operator/ (int b, Rational const &a)

  Compute quotient integer divided by Rational.
- bool operator== (Rational const &a, Rational const &b)

  Equality operators.
- bool operator== (Rational const &a, int b)

  Equality operator for a Rational and an integer.
- Rational operator- (Rational const &a)

  Unary negation of Rational.
- std::ostream & operator<< (std::ostream &out, Rational const &rational)

  Output stream inserter for a Rational.

12.204.1 Detailed Description

A Rational number (a ratio of integers).
A rational is always stored in a standard reduced form in which the denominator is a positive integer and the numerator and denominator have no common divisors other than unity. All integers, including zero, are stored with a denominator of 1.
Definition at line 33 of file Rational.h.

12.204.2 Constructor & Destructor Documentation

12.204.2.1 Rational() [1/4] `Util::Rational::Rational ( ) [inline]`

Default constructor.
Definition at line 271 of file Rational.h.

12.204.2.2 Rational() [2/4] `Util::Rational::Rational ( int num, int den ) [inline]`

Constructor, explicit numerator and denominator.
Denominator is reduced to greatest common divisor before return.

Parameters

num	numerator
den	denominator

Definition at line 280 of file Rational.h.

12.204.2.3 Rational() [3/4] `Util::Rational::Rational ( int number ) [inline]`

Constructor, construct from integer.
Creates a rational with a denominator == 1.

Parameters

number	integer number.

Definition at line 289 of file Rational.h.

12.204.2.4 Rational() [4/4] `Util::Rational::Rational ( Rational const & v ) [inline]`

Copy constructor.

Parameters

v	Rational to be copied

Definition at line 298 of file Rational.h.

12.204.2.5 ∼Rational() `Util::Rational::∼Rational ( ) [inline]`

Destructor.
Definition at line 75 of file Rational.h.

12.204.3 Member Function Documentation

12.204.3.1 operator=() [1/2] `Rational & Util::Rational::operator= ( Rational const & other ) [inline]`

Copy assignment from another Rational.

Parameters

other	Rational to assign.

Definition at line 307 of file Rational.h.

12.204.3.2 operator=() [2/2] `Rational & Util::Rational::operator= ( int other ) [inline]`

Assignment from integer.
Creates an integer using a denominator == 1.

Parameters

other	integer to assign.

Definition at line 318 of file Rational.h.

12.204.3.3 operator+=() [1/2]

`Rational & Util::Rational::operator+= ( Rational const & a ) [inline]`

Add another rational to this one.

Upon return, *this = this + a.

Parameters

a	increment (input)

Definition at line 331 of file Rational.h.

12.204.3.4 operator+=() [2/2]

`Rational & Util::Rational::operator+= ( int a ) [inline]`

Add an integer to this rational.

Upon return, *this = this + a.

Parameters

a	increment (input)

Definition at line 343 of file Rational.h.

12.204.3.5 operator-=() [1/2]

`Rational & Util::Rational::operator-= ( Rational const & a ) [inline]`

Subtract another rational from this one.

Upon return, *this = this + a.

Parameters

a	rational decrement (input)

Definition at line 354 of file Rational.h.

12.204.3.6 operator-=() [2/2]

`Rational & Util::Rational::operator-= ( int a ) [inline]`

Subtract an integer from this rational.

Upon return, *this = this + a.

Parameters

a	integer decrement (input)

Definition at line 366 of file Rational.h.

12.204.3.7 operator*=() [1/2]

`Rational & Util::Rational::operator*= ( Rational const & a ) [inline]`

Multiply this rational by another.
Upon return, ∗this = this∗a.

Parameters

a	Rational number to multiply this by (input)

Definition at line 377 of file Rational.h.

12.204.3.8 operator∗=() [2/2] `Rational & Util::Rational::operator*= ( int a ) [inline]`

Multiply this rational by an integer.
Upon return, ∗this = this∗a.

Parameters

a	integer to multiply this by (input)

Definition at line 389 of file Rational.h.

12.204.3.9 operator/=() [1/2] `Rational & Util::Rational::operator/= ( Rational const & a ) [inline]`

Divide this rational by another.
Upon return, ∗this = this∗a.

Parameters

a	rational number to divide this by (input)

Definition at line 400 of file Rational.h.
References UTIL_THROW.

12.204.3.10 operator/=() [2/2] `Rational & Util::Rational::operator/= ( int a ) [inline]`

Divide this rational by an integer.
Upon return, ∗this = this∗a.

Parameters

a	integer to divide this by (input)

Definition at line 415 of file Rational.h.
References UTIL_THROW.

12.204.3.11 num() `int Util::Rational::num ( ) const [inline]`

Return numerator.
Definition at line 431 of file Rational.h.

12.204.3.12 den() `int Util::Rational::den ( ) const [inline]`

Return denominator.

Definition at line 438 of file Rational.h.

12.204.3.13 operator double() `Util::Rational::operator double ( ) const [inline]`

Cast (convert) to double precision floating point.

Returns

double precision representation of this.

Definition at line 445 of file Rational.h.

12.204.3.14 serialize() `template<class Archive >`

```
void Util::Rational::serialize (
            Archive & ar,
            const unsigned int version ) [inline]
```

Serialize to/from an archive.

Parameters

ar	archive
version	archive version id

Definition at line 454 of file Rational.h.

12.204.3.15 commitMpiType() `static void Util::Rational::commitMpiType ( ) [static]`

Commit MPI datatype MpiTraits<Rational>::type.

12.204.4 Friends And Related Function Documentation

12.204.4.1 operator+ [1/2] `Rational operator+ (`

```
            Rational const & a,
            Rational const & b ) [friend]
```

Compute sum of two rationals.

Parameters

a	1st argument
b	2st argument

Returns

sum a + b

Definition at line 490 of file Rational.h.

12.204.4.2 operator+ [2/2] `Rational operator+ (`

`Rational const & a,`
`int b ) [friend]`

Compute sum of rational and integer.

Parameters

a	Rational argument
b	integer argument

Returns

sum a + b

Definition at line 505 of file Rational.h.

12.204.4.3 operator- [1/4] `Rational operator- (`

`Rational const & a,`
`Rational const & b ) [friend]`

Compute difference of rationals.

Parameters

a	1st argument
b	2st argument

Returns

difference a - b

Definition at line 530 of file Rational.h.

12.204.4.4 operator- [2/4] `Rational operator- (`

`Rational const & a,`
`int b ) [friend]`

Compute difference of rational and integer.

Parameters

a	Rational argument
b	integer argument

Returns

difference a - b

Definition at line 545 of file Rational.h.

12.204.4.5 operator- [3/4] `Rational operator- (`

`int b,`
`Rational const & a ) [friend]`

Compute difference of integer and rational.

Parameters

b	integer argument
a	Rational argument

Returns

difference b - a

Definition at line 559 of file Rational.h.

12.204.4.6 operator* [1/2] `Rational operator* ( Rational const & a, Rational const & b ) [friend]`

Compute product of rationals.

Parameters

a	1st Rational argument
b	2st Rational argument

Returns

product a*b

Definition at line 573 of file Rational.h.

12.204.4.7 operator* [2/2] `Rational operator* ( Rational const & a, int b ) [friend]`

Compute product of rational and integer.

Parameters

a	Rational argument
b	integer argument

Returns

product a*b

Definition at line 588 of file Rational.h.

12.204.4.8 operator/ [1/3] `Rational operator/ ( Rational const & a, Rational const & b ) [friend]`

Compute quotient of two rationals.

Parameters

a	1st Rational argument (numerator)
b	2st Rational argument (denominator)

Returns

ratio a/b

Definition at line 610 of file Rational.h.

12.204.4.9 operator/ [2/3]

```
Rational operator/ (
    Rational const & a,
    int b ) [friend]
```

Compute quotient Rational divided by integer.

Parameters

a	Rational argument (numerator)
b	integer argument (denominator)

Returns

ratio a/b

Definition at line 628 of file Rational.h.

12.204.4.10 operator/ [3/3]

```
Rational operator/ (
    int b,
    Rational const & a ) [friend]
```

Compute quotient integer divided by Rational.

Parameters

b	integer argument (numerator)
a	Rational argument (denominator)

Returns

ratio b/a

Definition at line 644 of file Rational.h.

12.204.4.11 operator== [1/2]

```
bool operator== (
    Rational const & a,
    Rational const & b ) [friend]
```

Equality operators.
Equality operator for two Rational numbers.

Parameters

a	1st Rational
b	2nd Rational

Returns

true if equal, false otherwise

Definition at line 674 of file Rational.h.

12.204.4.12 operator== [2/2]

```
bool operator== (
    Rational const & a,
    int b ) [friend]
```

Equality operator for a Rational and an integer.

Parameters

a	Rational number
b	integer number

Returns

true if equal, false otherwise

Definition at line 684 of file Rational.h.

12.204.4.13 operator- [4/4]

```
Rational operator- (
    Rational const & a ) [friend]
```

Unary negation of Rational.

Parameters

a	Rational number

Returns

negation -a

Definition at line 661 of file Rational.h.

12.204.4.14 operator<<

```
std::ostream& operator<< (
    std::ostream & out,
    Rational const & rational ) [friend]
```

Output stream inserter for a Rational.

Output elements of a rational to stream, without line breaks.

Parameters

out	output stream
rational	Rational to be written to stream

Returns

modified output stream

Definition at line 16 of file Rational.cpp.
The documentation for this class was generated from the following file:

- Rational.h

12.205 Util::RingBuffer< Data > Class Template Reference

Class for storing history of previous values in an array.
`#include <RingBuffer.h>`

Public Member Functions

- RingBuffer ()
  Contructor.
- RingBuffer (const RingBuffer< Data > &other)
  Copy contructor.
- RingBuffer & operator= (RingBuffer< Data > const &other)
  Assignment.
- virtual ∼RingBuffer ()
  Destructor.
- void allocate (int capacity)
  Allocate a new empty buffer.
- void clear ()
  Set previously allocated buffer to empty state.
- void append (Data const &value)
  Add a new value to the buffer.
- int size () const
  Return number of values currently in the buffer.
- int capacity () const
  Return the capacity of the buffer.
- bool isAllocated () const
  Return true if the RingBuffer has been allocated, false otherwise.
- bool isFull () const
  Return true if full (if size == capacity), false otherwise.
- const Data & operator[ ] (int offset) const
  Retrieve a const value, a specified number of time steps ago.
- Data & operator[ ] (int offset)
  Retrieve a value, a specified number of time steps ago.
- template< class Archive >
  void serialize (Archive &ar, const unsigned int version)
  Serialize a RingBuffer to/from an Archive.

12.205.1 Detailed Description

template< class Data >
class Util::RingBuffer< Data >

Class for storing history of previous values in an array.
Data is stored in a circular buffer, in which, once the array is full, the newest data value overwrites the oldest.
Definition at line 26 of file RingBuffer.h.

12.205.2 Constructor & Destructor Documentation

12.205.2.1 RingBuffer() [1/2] template<class Data >

`Util::RingBuffer< Data >::RingBuffer`

Contructor.

Definition at line 140 of file RingBuffer.h.

12.205.2.2 RingBuffer() [2/2] template<class Data >

```
Util::RingBuffer< Data >::RingBuffer (
            const RingBuffer< Data > & other )
```

Copy contructor.

Parameters

other	object to be copied.

Definition at line 155 of file RingBuffer.h.

12.205.2.3 ∼RingBuffer() template<class Data >

`Util::RingBuffer< Data >::∼RingBuffer [virtual]`

Destructor.

Definition at line 219 of file RingBuffer.h.

12.205.3 Member Function Documentation

12.205.3.1 operator=() template<class Data >

```
RingBuffer< Data > & Util::RingBuffer< Data >::operator= (
            RingBuffer< Data > const & other )
```

Assignment.

Parameters

other	object to be assigned.

Definition at line 182 of file RingBuffer.h.

12.205.3.2 allocate() template<class Data >

```
void Util::RingBuffer< Data >::allocate (
            int capacity )
```

Allocate a new empty buffer.

Allocate a new array containing capacity elements.

Throw an Exception if this RingBuffer has already been allocated - a RingBuffer can only be allocated once.

Parameters

capacity	number of elements to allocate.

Definition at line 228 of file RingBuffer.h.

12.205.3.3 clear() `template<class Data >`

`void Util::RingBuffer< Data >::clear [inline]`

Set previously allocated buffer to empty state.

Definition at line 245 of file RingBuffer.h.

12.205.3.4 append() `template<class Data >`

```
void Util::RingBuffer< Data >::append (
            Data const & value ) [inline]
```

Add a new value to the buffer.

Parameters

value	new value to be added.

Definition at line 256 of file RingBuffer.h.

12.205.3.5 size() `template<class Data >`

`int Util::RingBuffer< Data >::size [inline]`

Return number of values currently in the buffer.

Definition at line 277 of file RingBuffer.h.

12.205.3.6 capacity() `template<class Data >`

`int Util::RingBuffer< Data >::capacity [inline]`

Return the capacity of the buffer.

Definition at line 285 of file RingBuffer.h.

12.205.3.7 isAllocated() `template<class Data >`

`bool Util::RingBuffer< Data >::isAllocated [inline]`

Return true if the RingBuffer has been allocated, false otherwise.

Definition at line 293 of file RingBuffer.h.

Referenced by Util::RingBuffer< Util::FArray< double, 6 > >::operator=().

12.205.3.8 isFull() `template<class Data >`

`bool Util::RingBuffer< Data >::isFull [inline]`

Return true if full (if size == capacity), false otherwise.

Definition at line 301 of file RingBuffer.h.

12.205.3.9 operator[]() [1/2] `template<class Data >`

```
const Data & Util::RingBuffer< Data >::operator[] (
            int offset ) const [inline]
```

Retrieve a const value, a specified number of time steps ago.

Parameters

offset	number of steps back in time (offset=0 is current value).

Definition at line 309 of file RingBuffer.h.

12.205.3.10 operator[]() [2/2]

template<class Data >
Data & Util::RingBuffer< Data >::operator[] (
int *offset*) [inline]

Retrieve a value, a specified number of time steps ago.

Parameters

offset	number of steps back in time (offset=0 is current value).

Definition at line 327 of file RingBuffer.h.

12.205.3.11 serialize()

template<class Data >
template<class Archive >
void Util::RingBuffer< Data >::serialize (
Archive & *ar*,
const unsigned int *version*)

Serialize a RingBuffer to/from an Archive.

Parameters

ar	archive
version	archive version id

Definition at line 345 of file RingBuffer.h.

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

- RingBuffer.h

12.206 Util::ScalarParam< Type > Class Template Reference

Template for a Parameter object associated with a scalar variable.

`#include <ScalarParam.h>`

Inheritance diagram for Util::ScalarParam< Type >:

classUtil_1_1ScalarParam-eps-converted-to.pdf

Public Member Functions

- ScalarParam (const char *label, Type &value, bool isRequired=true)

  Constructor.
- void writeParam (std::ostream &out)

  Write parameter to stream.
- void setValue (Type &value)

  Set the pointer to point a specific variable.

Protected Member Functions

- virtual void readValue (std::istream &in)

  Read parameter value from an input stream.
- virtual void loadValue (Serializable::IArchive &ar)

  Load bare parameter value from an archive.
- virtual void saveValue (Serializable::OArchive &ar)

  Save parameter value to an archive.
- virtual void bcastValue ()

  Broadcast parameter value within the ioCommunicator.

Additional Inherited Members

12.206.1 Detailed Description

template<class Type>
class Util::ScalarParam< Type >

Template for a Parameter object associated with a scalar variable.
This template can be used to define a Parameter subclass for any data type for which there exist inserter (<<) and extractor (>>) operators for stream io.
Definition at line 34 of file ScalarParam.h.

12.206.2 Constructor & Destructor Documentation

12.206.2.1 ScalarParam() `template<class Type >`

```
Util::ScalarParam< Type >::ScalarParam (
            const char * label,
            Type & value,
            bool isRequired = true )
```

Constructor.

Parameters

label	label string const.
value	reference to parameter value.
isRequired	Is this a required parameter?

Definition at line 111 of file ScalarParam.h.

12.206.3 Member Function Documentation

12.206.3.1 writeParam() `template<class Type >`

```
void Util::ScalarParam< Type >::writeParam (
            std::ostream & out )  [virtual]
```

Write parameter to stream.

Parameters

out	output stream

Implements Util::ParamComponent.

Definition at line 139 of file ScalarParam.h.

References Util::Parameter::Precision, and Util::Parameter::Width.

12.206.3.2 setValue() `template<class Type >`

```
void Util::ScalarParam< Type >::setValue (
            Type & value )
```

Set the pointer to point a specific variable.

Parameters

value	variable that holds the parameter value.

Definition at line 155 of file ScalarParam.h.

12.206.3.3 readValue() `template<class Type >`

```
void Util::ScalarParam< Type >::readValue (
            std::istream & in )  [protected], [virtual]
```

Read parameter value from an input stream.

Parameters

in	input stream from which to read

Reimplemented from Util::Parameter.

Definition at line 118 of file ScalarParam.h.

12.206.3.4 loadValue() `template<class Type >`

```
void Util::ScalarParam< Type >::loadValue (
            Serializable::IArchive & ar )  [protected], [virtual]
```

Load bare parameter value from an archive.

Parameters

ar	input archive from which to load

Reimplemented from Util::Parameter.
Definition at line 122 of file ScalarParam.h.

12.206.3.5 saveValue() `template<class Type >`

```
void Util::ScalarParam< Type >::saveValue (
            Serializable::OArchive & ar )  [protected], [virtual]
```

Save parameter value to an archive.

Parameters

ar	output archive to which to save

Reimplemented from Util::Parameter.
Definition at line 126 of file ScalarParam.h.

12.206.3.6 bcastValue() `template<class Type >`

```
void Util::ScalarParam< Type >::bcastValue  [protected], [virtual]
```

Broadcast parameter value within the ioCommunicator.
Reimplemented from Util::Parameter.
Definition at line 131 of file ScalarParam.h.
The documentation for this class was generated from the following file:

- ScalarParam.h

12.207 Util::ScopedPtr< T > Class Template Reference

A very simple RAII pointer.
`#include <ScopedPtr.h>`

Public Types

- typedef T element_type

  Type of object pointed to.

Public Member Functions

- ScopedPtr (T ∗p=0)

  Constructor.
- ∼ScopedPtr ()

  Destructor, destroys object pointed to, if any.
- void reset (T ∗p=0)

  Acquire ownership of a built-in pointer.
- T & operator∗ () const

  Dereference.
- T ∗ operator-> () const

  Member access.
- T ∗ get () const

  Return enclosed built-in pointer.

12.207.1 Detailed Description

template<typename T>
class Util::ScopedPtr< T >

A very simple RAII pointer.

A ScopedPtr mimics a built-in pointer, except that guarantees destruction of the object to which it points when the ScopedPtr goes out of scope. It accepts ownership of a built-in pointer either upon construction or by the reset() method, and deletes the associated object in its destructor. A ScopedPtr cannot be copy constructed or assigned.

Similar to boost::scoped_ptr, with minor differences. It takes the same amount of memory as a built-in pointer, and should be equally fast.

Definition at line 29 of file ScopedPtr.h.

12.207.2 Member Typedef Documentation

12.207.2.1 element_type `template<typename T >`

`typedef T Util::ScopedPtr< T >::element_type`

Type of object pointed to.

Definition at line 35 of file ScopedPtr.h.

12.207.3 Constructor & Destructor Documentation

12.207.3.1 ScopedPtr() `template<typename T >`

```
Util::ScopedPtr< T >::ScopedPtr (
            T * p = 0 )  [inline], [explicit]
```

Constructor.

Definition at line 38 of file ScopedPtr.h.

12.207.3.2 ∼ScopedPtr() `template<typename T >`

```
Util::ScopedPtr< T >::∼ScopedPtr ( )  [inline]
```

Destructor, destroys object pointed to, if any.

Definition at line 42 of file ScopedPtr.h.

12.207.4 Member Function Documentation

12.207.4.1 reset() `template<typename T >`

```
void Util::ScopedPtr< T >::reset (
            T * p = 0 )  [inline]
```

Acquire ownership of a built-in pointer.

Parameters

p	built-in pointer to acquire.

Definition at line 54 of file ScopedPtr.h.

12.207.4.2 operator∗() `template<typename T >`

`T& Util::ScopedPtr< T >::operator* ( ) const [inline]`

Dereference.

Definition at line 63 of file ScopedPtr.h.

12.207.4.3 operator->() `template<typename T >`

`T* Util::ScopedPtr< T >::operator-> ( ) const [inline]`

Member access.

Definition at line 67 of file ScopedPtr.h.

12.207.4.4 get() `template<typename T >`

`T* Util::ScopedPtr< T >::get ( ) const [inline]`

Return enclosed built-in pointer.

Definition at line 71 of file ScopedPtr.h.

Referenced by Util::isNull().

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

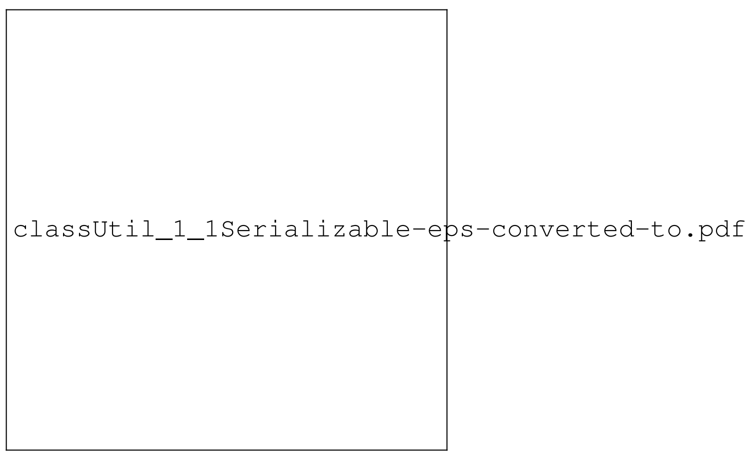
- ScopedPtr.h

12.208 Util::Serializable Class Reference

Abstract class for serializable objects.

`#include <Serializable.h>`

Inheritance diagram for Util::Serializable:



Public Types

- typedef BinaryFileOArchive OArchive

  Type of output archive used by save method.
- typedef BinaryFileIArchive IArchive

  Type of input archive used by load method.

Public Member Functions

- virtual ∼Serializable ()

  Destructor.

- virtual void save (OArchive &ar)=0

  Save to an archive.
- virtual void load (IArchive &ar)=0

  Load from an archive.

12.208.1 Detailed Description

Abstract class for serializable objects.

This class defines an interface for serialization of objects. The save() method saves the internal state of an object to an archive, and the load() method loads the state from an archive.

The type of archive to be used is specified by the OArchive and IArchive typedefs. The two concrete classes that are referred to by these typedefs should be forward declared in this file, and the header files for these two classes must be included in the file Serializable_includes.h. The file Serializable_includes.h should be included in source files that implement that load and save methods for subclasses.

Definition at line 34 of file Serializable.h.

12.208.2 Member Typedef Documentation

12.208.2.1 OArchive `typedef BinaryFileOArchive Util::Serializable::OArchive`

Type of output archive used by save method.

Definition at line 42 of file Serializable.h.

12.208.2.2 IArchive `typedef BinaryFileIArchive Util::Serializable::IArchive`

Type of input archive used by load method.

Definition at line 47 of file Serializable.h.

12.208.3 Constructor & Destructor Documentation

12.208.3.1 ∼Serializable() `virtual Util::Serializable::~Serializable ( ) [inline], [virtual]`

Destructor.

Definition at line 52 of file Serializable.h.

12.208.4 Member Function Documentation

12.208.4.1 save() `virtual void Util::Serializable::save ( OArchive & ar ) [pure virtual]`

Save to an archive.

Parameters

ar	binary saving (output) archive.

Implemented in Util::ParamComposite, Util::FileMaster, Util::Parameter, Util::Manager< Data >, Util::AutoCorrArray< Data, Product >, Util::AutoCorr< Data, Product >, Util::Average, Util::MeanSqDispArray< Data >, Util::Distribution, Util::TensorAverage, Util::SymmTensorAverage, Util::Random, Util::IntDistribution, Util::RadialDistribution, Util::ParamComponent, and Util::AutoCorrelation< Data, Product >.

12.208.4.2 load() `virtual void Util::Serializable::load ( IArchive & ar ) [pure virtual]`

Load from an archive.

Parameters

ar	binary loading (input) archive.

Implemented in Util::ParamComposite, Util::Parameter, Util::ParamComponent, and Util::AutoCorrelation< Data, Product >.

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

- Serializable.h

12.209 Util::Setable< T > Class Template Reference

Template for a value that can be set or declared null (i.e., unknown).

`#include <Setable.h>`

Public Member Functions

- Setable ()

  Default constructor.
- Setable (const Setable< T > &other)

  Copy constructor.
- Setable (const T &value)

  Construct from T value (explicit).
- Setable< T > & operator= (const Setable< T > &other)

  Assignment from another Setable<T> object.
- Setable< T > & operator= (const T &value)

  Assignment from T value.
- void set (const T &value)

  Set the value and mark as set.
- void unset ()

  Unset the value (mark as unknown).
- bool isSet () const

  Is this object set (is the value known)?
- const T & value () const

  Return value (if set).
- bool isValid (MPI::Intracomm &communicator) const

  Test consistency of states on different processors.

12.209.1 Detailed Description

template<class T>
class Util::Setable< T >

Template for a value that can be set or declared null (i.e., unknown).

Type T must be copy-constructable and have an assignment (=) operator.

Convention for MPI programs: In parallel MPI programs in which a value for a variable is calculated by a reduce operation and is set only on a master processor, a default value should be set on all other processors whenever the true value is set on the master. This indicates on all processors that the value is known, though it may only be available on the master processor. Similarly, when a value is unset, the unset() function should be called on all processors. This convention allows the isSet() function to be used on all processors to query whether the value is known, which may be then be used to decide when to initiate a recomputation that may require computation on all processors. This convention is imposed by the isValid() function, which requires that isSet have the same value on all processors within a communicator (i.e., all true or all false).

Definition at line 38 of file Setable.h.

12.209.2 Constructor & Destructor Documentation

12.209.2.1 Setable() [1/3]

```
template<class T >
Util::Setable< T >::Setable ( )  [inline]
```

Default constructor.

Definition at line 46 of file Setable.h.

12.209.2.2 Setable() [2/3]

```
template<class T >
Util::Setable< T >::Setable (
            const Setable< T > & other )  [inline]
```

Copy constructor.

Parameters

other	Setable object being copied.

Definition at line 56 of file Setable.h.

12.209.2.3 Setable() [3/3]

```
template<class T >
Util::Setable< T >::Setable (
            const T & value )  [inline], [explicit]
```

Construct from T value (explicit).

Parameters

value	value of wrapped object

Definition at line 66 of file Setable.h.

12.209.3 Member Function Documentation

12.209.3.1 operator=() [1/2]

```
template<class T >
Setable<T>& Util::Setable< T >::operator= (
            const Setable< T > & other )  [inline]
```

Assignment from another Setable<T> object.

Parameters

other	object on RHS of assignment

Definition at line 76 of file Setable.h.

12.209.3.2 operator=() [2/2]

```
template<class T >
Setable<T>& Util::Setable< T >::operator= (
            const T & value )  [inline]
```

Assignment from T value.

Equivalent to set(value). Sets the value and marks it as set.

Parameters

value	T value on RHS of assignment

Returns

this object

Definition at line 95 of file Setable.h.

References Util::Setable< T >::value().

12.209.3.3 set()

```
template<class T >
void Util::Setable< T >::set (
            const T & value )  [inline]
```

Set the value and mark as set.

Parameters

value	value to be assigned.

Definition at line 107 of file Setable.h.

References Util::Setable< T >::value().

12.209.3.4 unset()

```
template<class T >
void Util::Setable< T >::unset ( )  [inline]
```

Unset the value (mark as unknown).

Definition at line 116 of file Setable.h.

12.209.3.5 isSet()

```
template<class T >
bool Util::Setable< T >::isSet ( ) const  [inline]
```

Is this object set (is the value known)?

Returns

true if set (known), false if null (unknown).

Definition at line 124 of file Setable.h.

12.209.3.6 value()

```
template<class T >
const T& Util::Setable< T >::value ( ) const  [inline]
```

Return value (if set).

Throws an Exception if value is not set.
Definition at line 132 of file Setable.h.
References UTIL_THROW.
Referenced by Util::Setable< T >::operator=(), and Util::Setable< T >::set().

12.209.3.7 isValid() template<typename T >

```
bool Util::Setable< T >::isValid (
            MPI::Intracomm & communicator ) const
```

Test consistency of states on different processors.
If valid, return true, else throws an Exception. The state is valid if the value of isSet is the same on all processors.
Definition at line 163 of file Setable.h.
References UTIL_THROW.
The documentation for this class was generated from the following file:

- Setable.h

12.210 Util::Signal< T > Class Template Reference

Notifier (or subject) in the Observer design pattern.
`#include <Signal.h>`

Public Member Functions

- Signal ()

  Default constructor.
- ∼Signal ()

  Destructor.
- template<class Observer >
  void addObserver (Observer &observer, void(Observer::∗methodPtr)(const T &))

  Register an observer.
- void clear ()

  Clear all observerse from list.
- int nObserver () const

  Get number of registered observers.
- void notify (const T &t)

  Notify all observers.

12.210.1 Detailed Description

template<typename T = void>
class Util::Signal< T >

Notifier (or subject) in the Observer design pattern.
A Signal manages a list of registered functor objects, and provides a void Signal<T>::notify(const T&) method that calls them all with the same argument.
The explicit specialization Signal<void>, or Signal<>, has a notify method void Signal<>::notify() that takes no parameters, which calls a method of each observer that takes no parameters.
Definition at line 38 of file Signal.h.

12.210.2 Constructor & Destructor Documentation

12.210.2.1 Signal() `template<typename T = void>`

`Util::Signal< T >::Signal ( ) [inline]`

Default constructor.

Definition at line 48 of file Signal.h.

12.210.2.2 ∼Signal() `template<typename T >`

`Util::Signal< T >::~Signal`

Destructor.

Definition at line 16 of file Signal.cpp.

12.210.3 Member Function Documentation

12.210.3.1 addObserver() `template<typename T >`

```
template<class Observer >
void Util::Signal< T >::addObserver (
            Observer & observer,
            void(Observer::*)(const T &) methodPtr )
```

Register an observer.

Parameters

observer	observer object (invokes method)
methodPtr	pointer to relevant method

Definition at line 111 of file Signal.h.

12.210.3.2 clear() `template<typename T >`

`void Util::Signal< T >::clear`

Clear all observerse from list.

Definition at line 36 of file Signal.cpp.

12.210.3.3 nObserver() `template<typename T >`

`int Util::Signal< T >::nObserver`

Get number of registered observers.

Definition at line 51 of file Signal.cpp.

12.210.3.4 notify() `template<typename T >`

```
void Util::Signal< T >::notify (
            const T & t )
```

Notify all observers.

This method notifies all registered observers by calling the appropriate method of each observer, passing each the parameter t as argument. The explicit specialization Signal<>, with T=void, is used for notification methods that take

Parameters

t	Argument passed to notification methods of all observers.

Definition at line 22 of file Signal.cpp.
The documentation for this class was generated from the following files:

- Signal.h
- Signal.cpp

12.211 Util::Signal< void > Class Reference

Notifier (or subject) in the Observer design pattern (zero parameters).
`#include <Signal.h>`

Public Member Functions

- Signal ()

  Default constructor.
- ∼Signal ()

  Destructor.
- template<class Observer >
  void addObserver (Observer &observer, void(Observer::∗methodPtr)())

  Register an observer.
- void clear ()

  Clear all observerse from list.
- int nObserver () const

  Get number of registered observers.
- void notify ()

  Notify all observers.

12.211.1 Detailed Description

Notifier (or subject) in the Observer design pattern (zero parameters).
This explicit specialization of Signal<T> provides a notify method that takes no parameters, and that calls methods of each observer object that take no parameters.
Definition at line 168 of file Signal.h.

12.211.2 Constructor & Destructor Documentation

12.211.2.1 Signal() `Util::Signal< void >::Signal ( ) [inline]`

Default constructor.
Definition at line 176 of file Signal.h.

12.211.2.2 ∼Signal() `Util::Signal< void >::∼Signal ( )`

Destructor.

12.211.3 Member Function Documentation

12.211.3.1 addObserver() `template<class Observer >`

`void Util::Signal< void >::addObserver (`
`Observer & observer,`
`void(Observer::*)() methodPtr )`

Register an observer.

Parameters

observer	observer object (invokes method)
methodPtr	pointer to relevant method

12.211.3.2 clear() `void Util::Signal< void >::clear ( )`

Clear all observerse from list.

12.211.3.3 nObserver() `int Util::Signal< void >::nObserver ( ) const`

Get number of registered observers.

12.211.3.4 notify() `void Util::Signal< void >::notify ( )`

Notify all observers.

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

- Signal.h

12.212 Util::SSet< Data, Capacity > Class Template Reference

Statically allocated array of pointers to an unordered set.

`#include <SSet.h>`

Public Member Functions

- SSet ()
  Default constructor.
- SSet (const SSet< Data, Capacity > &other)
  Copy constructor.
- SSet< Data, Capacity > & operator= (const SSet< Data, Capacity > &other)
  Assignment, element by element.
- ~SSet ()
  Destructor.
- void append (Data &data)
  Add an object to the set.
- void remove (const Data &data)
  Remove an object from the set.
- void clear ()
  Set logical size to zero and nullify all elements.
- int capacity () const
  Return physical capacity of array.
- int size () const

Return logical size of this array.

- bool isElement (const Data &data) const

  Is an object an element of the set?
- int index (const Data &data) const

  Return the current index of an object within the set, if any.
- void begin (PArrayIterator< Data > &iterator)

  Set a PArrayIterator to the beginning of this Array.
- void begin (ConstPArrayIterator< Data > &iterator) const

  Set a ConstPArrayIterator to the beginning of this Array.
- Data & operator[ ] (int i)

  Mimic C array subscripting.
- const Data & operator[ ] (int i) const

  Mimic C array subscripting.

Protected Attributes

- Data ∗ ptrs_ [Capacity]

  Array of pointers to Data objects.
- int size_

  Logical size of array (number of elements in array).

12.212.1 Detailed Description

template<typename Data, int Capacity>
class Util::SSet< Data, Capacity >

Statically allocated array of pointers to an unordered set.
An SSet is a statically allocated array that holds pointers to a set of objects. It implements the same interface as PArray and FPArray, plus additional remove() and index() methods. As for any pointer array container, the [] operator returns an associated object by reference .
An SSet holds a set of pointers in a contiguous array. The size is the number of pointers now in the container, and the Capacity is the maximum number it can hold. The class is implemented as a wrapper for a statically allocated C array of Capacity elements.
The append method adds a pointer to the end of the sequence. The remove method removes an object from the set, or throws an exception if the object is not found in the set. As for an ArraySet, the remove method repacks the sequence of pointers by moving the last element to the position of the element that is being removed. Removal of an element thus generally changes the order in which the remaining elements are stored.
Definition at line 43 of file SSet.h.

12.212.2 Constructor & Destructor Documentation

12.212.2.1 SSet() [1/2] `template<typename Data , int Capacity>`

`Util::SSet< Data, Capacity >::SSet [inline]`

Default constructor.
Definition at line 177 of file SSet.h.

12.212.2.2 SSet() [2/2] template<typename Data , int Capacity>

```
Util::SSet< Data, Capacity >::SSet (
            const SSet< Data, Capacity > & other )
```

Copy constructor.

Copies all pointers.

Parameters

other	the SSet to be copied.

Definition at line 185 of file SSet.h.

References Util::SSet< Data, Capacity >::ptrs_, and Util::SSet< Data, Capacity >::size_.

12.212.2.3 ∼SSet() template<typename Data , int Capacity>

```
Util::SSet< Data, Capacity >::∼SSet
```

Destructor.

Definition at line 240 of file SSet.h.

12.212.3 Member Function Documentation

12.212.3.1 operator=() template<typename Data , int Capacity>

```
SSet< Data, Capacity > & Util::SSet< Data, Capacity >::operator= (
            const SSet< Data, Capacity > & other )
```

Assignment, element by element.

Parameters

other	the rhs SSet

Definition at line 209 of file SSet.h.

References Util::SSet< Data, Capacity >::size_, and UTIL_THROW.

12.212.3.2 append() template<typename Data , int Capacity>

```
void Util::SSet< Data, Capacity >::append (
            Data & data ) [inline]
```

Add an object to the set.

Appends a pointer to the object to the end of the sequence.

Parameters

data	Data to add to end of array.

Definition at line 307 of file SSet.h.

References UTIL_THROW.

12.212.3.3 remove() template<typename Data , int Capacity>

```
void Util::SSet< Data, Capacity >::remove (
```

```
const Data & data )
```

Remove an object from the set.

Removal of an object generally changes the storage order of the remaining objects.

Exceptions

Exception	if object data is not in the Set.

Parameters

data	object to be removed.

Definition at line 332 of file SSet.h.

References UTIL_THROW.

12.212.3.4 clear() `template<typename Data , int Capacity>`

```
void Util::SSet< Data, Capacity >::clear  [inline]
```

Set logical size to zero and nullify all elements.

Definition at line 320 of file SSet.h.

12.212.3.5 capacity() `template<typename Data , int Capacity>`

```
int Util::SSet< Data, Capacity >::capacity  [inline]
```

Return physical capacity of array.

Definition at line 247 of file SSet.h.

12.212.3.6 size() `template<typename Data , int Capacity>`

```
int Util::SSet< Data, Capacity >::size  [inline]
```

Return logical size of this array.

Definition at line 254 of file SSet.h.

12.212.3.7 isElement() `template<typename Data , int Capacity>`

```
bool Util::SSet< Data, Capacity >::isElement (
              const Data & data ) const
```

Is an object an element of the set?

Parameters

data	object of interest.

Definition at line 364 of file SSet.h.

12.212.3.8 index() `template<typename Data , int Capacity>`

```
int Util::SSet< Data, Capacity >::index (
              const Data & data ) const
```

Return the current index of an object within the set, if any.

Return the current index of an element within the set, or return -1 if the element is not in the set.

This method returns the current index of the pointer to object data within this SSet, in the range 0 < index < size() - 1. The method returns -1 if data is the object is not in the set.
Throws an exception if data is not in the associated array.

Parameters

data	object of interest.

Returns

current index of pointer to element within this SSet.

Definition at line 385 of file SSet.h.

12.212.3.9 begin() [1/2]

```
template<typename Data , int Capacity>
void Util::SSet< Data, Capacity >::begin (
            PArrayIterator< Data > & iterator )  [inline]
```

Set a PArrayIterator to the beginning of this Array.

Parameters

iterator	PArrayIterator, initialized on output.

Definition at line 263 of file SSet.h.
References Util::PArrayIterator< Data >::setCurrent(), and Util::PArrayIterator< Data >::setEnd().

12.212.3.10 begin() [2/2]

```
template<typename Data , int Capacity>
void Util::SSet< Data, Capacity >::begin (
            ConstPArrayIterator< Data > & iterator ) const  [inline]
```

Set a ConstPArrayIterator to the beginning of this Array.

Parameters

iterator	ConstPArrayIterator, initialized on output.

Definition at line 273 of file SSet.h.
References Util::ConstPArrayIterator< Data >::setCurrent(), and Util::ConstPArrayIterator< Data >::setEnd().

12.212.3.11 operator[]() [1/2]

```
template<typename Data , int Capacity>
Data & Util::SSet< Data, Capacity >::operator[] (
            int i )  [inline]
```

Mimic C array subscripting.

Parameters

i	array index

Returns

reference to element i

Definition at line 283 of file SSet.h.

12.212.3.12 operator[]() [2/2] template<typename Data , int Capacity>

const Data & Util::SSet< Data, Capacity >::operator[] (
int i) const [inline]

Mimic C array subscripting.

Parameters

i	array index

Returns

const reference to element i

Definition at line 294 of file SSet.h.

12.212.4 Member Data Documentation

12.212.4.1 ptrs_ template<typename Data , int Capacity>

Data* Util::SSet< Data, Capacity >::ptrs_[Capacity] [protected]

Array of pointers to Data objects.
Definition at line 164 of file SSet.h.
Referenced by Util::SSet< Data, Capacity >::SSet().

12.212.4.2 size_ template<typename Data , int Capacity>

int Util::SSet< Data, Capacity >::size_ [protected]

Logical size of array (number of elements in array).
Definition at line 167 of file SSet.h.
Referenced by Util::SSet< Data, Capacity >::operator=(), and Util::SSet< Data, Capacity >::SSet().
The documentation for this class was generated from the following file:

- SSet.h

12.213 Util::Str Class Reference

Wrapper for a std::string, for formatted ostream output.
#include <Str.h>

Public Member Functions

Constructors

- Str ()
  Default constructor.
- Str (std::string value)
  Constructor, value only.

- Str (std::string value, int width)

  Constructor, value and width.

Mutators

- void **setValue** (std::string value)
- void **setWidth** (int width)

Accessors

- std::string **value** () const
- int **width** () const
- std::istream & operator>> (std::istream &in, Str &object)

  Input stream extractor for an Str object.
- std::ostream & operator<< (std::ostream &out, const Str &object)

  Output stream inserter for an Str object.

12.213.1 Detailed Description

Wrapper for a std::string, for formatted ostream output.

An Str object has std::string value, and an integer output field width. The << operator for an Str object uses the specified width. The value and width may both be specified as parameters to a constructor. If the width is not specified as a constructor parameter, it is set within the constructor to a default value given by Format::defaultWidth().

An Str object may be passed to an ostream as a temporary object. For example, the expression:

```
std::cout « Str("Hello") « Str("World", 20) « std::endl;
```

outputs "Hello" using the default width, followed by "World" in a field of width 20.

Definition at line 36 of file Str.h.

12.213.2 Constructor & Destructor Documentation

12.213.2.1 Str() [1/3] `Util::Str::Str ( )`

Default constructor.

Definition at line 17 of file Str.cpp.

12.213.2.2 Str() [2/3] `Util::Str::Str ( std::string value ) [explicit]`

Constructor, value only.

Constructor, value only (explicit).

Definition at line 23 of file Str.cpp.

12.213.2.3 Str() [3/3] `Util::Str::Str ( std::string value, int width )`

Constructor, value and width.

Definition at line 29 of file Str.cpp.

12.213.3 Friends And Related Function Documentation

12.213.3.1 operator>> `std::istream& operator>> ( std::istream & in, Str & object ) [friend]`

Input stream extractor for an Str object.

Parameters

in	input stream
object	Str object to be read from stream

Returns

modified input stream

Definition at line 49 of file Str.cpp.

12.213.3.2 operator<< `std::ostream& operator<< ( std::ostream & out, const Str & object ) [friend]`

Output stream inserter for an Str object.

Parameters

out	output stream
object	Str to be written to stream

Returns

modified output stream

Definition at line 58 of file Str.cpp.
The documentation for this class was generated from the following files:

- Str.h
- Str.cpp

12.214 Util::SymmTensorAverage Class Reference

Calculates averages of all components of a Tensor-valued variable.

```
#include <SymmTensorAverage.h>
```

Inheritance diagram for Util::SymmTensorAverage:

classUtil_1_1SymmTensorAverage-eps-converted-to.pdf

Public Member Functions

- SymmTensorAverage (int blockFactor=2)

  Constructor.
- virtual ∼SymmTensorAverage ()

  Destructor.
- void setNSamplePerBlock (int nSamplePerBlock)

  Set nSamplePerBlock.
- void readParameters (std::istream &in)

  Read parameter nSamplePerBlock from file and initialize.
- virtual void loadParameters (Serializable::IArchive &ar)

  Load internal state from an archive.
- virtual void save (Serializable::OArchive &ar)

  Save internal state to an archive.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize this to or from an archive.
- void clear ()

  Clear all accumulators, set to empty initial state.
- void sample (const Tensor &value)

  Add a sampled value to the ensemble.
- const Average & operator() (int i, int j)

  Access the Average object for one tensor component.
- int nSamplePerBlock () const

  Get number of samples per block average.
- int iBlock () const

  Get number of samples in current block average.
- bool isBlockComplete () const

  Is the current block average complete?

Additional Inherited Members

12.214.1 Detailed Description

Calculates averages of all components of a Tensor-valued variable.
SymmTensorAverage is a simple container for an array of Average objects, each of which calculates averages and error estimates for one component of a Tensor.
Definition at line 31 of file SymmTensorAverage.h.

12.214.2 Constructor & Destructor Documentation

12.214.2.1 SymmTensorAverage() `Util::SymmTensorAverage::SymmTensorAverage ( int blockFactor = 2 )`

Constructor.

Parameters

blockFactor	ratio of block sizes for subsequent stages.

Definition at line 19 of file SymmTensorAverage.cpp.
References Util::Dimension, and Util::ParamComposite::setClassName().

12.214.2.2 ∼SymmTensorAverage() `Util::SymmTensorAverage::∼SymmTensorAverage ( ) [virtual]`

Destructor.
Definition at line 38 of file SymmTensorAverage.cpp.

12.214.3 Member Function Documentation

12.214.3.1 setNSamplePerBlock() `void Util::SymmTensorAverage::setNSamplePerBlock ( int nSamplePerBlock )`

Set nSamplePerBlock.
If nSamplePerBlock > 0, the sample function will increment block averages, and reset the average every nSamplePer←Block samples.
If nSamplePerBlock == 0, block averaging is disabled. This is the default (i.e., the initial value set in the constructor).

Parameters

nSamplePerBlock	number of samples per block average output

Definition at line 44 of file SymmTensorAverage.cpp.
References Util::Dimension, nSamplePerBlock(), and UTIL_THROW.

12.214.3.2 readParameters() `void Util::SymmTensorAverage::readParameters ( std::istream & in ) [virtual]`

Read parameter nSamplePerBlock from file and initialize.
See setNSamplePerBlock() for discussion of value.

Parameters

in	input stream

Reimplemented from Util::ParamComposite.
Definition at line 63 of file SymmTensorAverage.cpp.
References Util::Dimension, and UTIL_THROW.

12.214.3.3 loadParameters() `void Util::SymmTensorAverage::loadParameters ( Serializable::IArchive & ar ) [virtual]`

Load internal state from an archive.

Parameters

ar	input/loading archive

Reimplemented from Util::ParamComposite.
Definition at line 82 of file SymmTensorAverage.cpp.
References Util::Dimension, and UTIL_THROW.

12.214.3.4 save() `void Util::SymmTensorAverage::save ( Serializable::OArchive & ar ) [virtual]`

Save internal state to an archive.

Parameters

ar	output/saving archive

Reimplemented from Util::ParamComposite.

Definition at line 102 of file SymmTensorAverage.cpp.

12.214.3.5 serialize() `template<class Archive > void Util::SymmTensorAverage::serialize ( Archive & ar, const unsigned int version )`

Serialize this to or from an archive.

Parameters

ar	input or output archive
version	file version id

Definition at line 181 of file SymmTensorAverage.h.

References Util::Dimension.

12.214.3.6 clear() `void Util::SymmTensorAverage::clear ( )`

Clear all accumulators, set to empty initial state.

Definition at line 108 of file SymmTensorAverage.cpp.

References Util::Dimension.

12.214.3.7 sample() `void Util::SymmTensorAverage::sample ( const Tensor & value )`

Add a sampled value to the ensemble.

Parameters

value	sampled value

Definition at line 124 of file SymmTensorAverage.cpp.

References Util::Dimension.

12.214.3.8 operator()() `const Average & Util::SymmTensorAverage::operator() ( int i, int j )`

Access the Average object for one tensor component.

Parameters

i	first index of associated tensor component
j	second index of associated tensor component

Returns

Average object associated with element (i, j)

Definition at line 145 of file SymmTensorAverage.cpp.

12.214.3.9 nSamplePerBlock() `int Util::SymmTensorAverage::nSamplePerBlock ( ) const [inline]`

Get number of samples per block average.
Returns zero if block averaging is disabled.

Returns

number of samples per block (or 0 if disabled).

Definition at line 162 of file SymmTensorAverage.h.
Referenced by setNSamplePerBlock().

12.214.3.10 iBlock() `int Util::SymmTensorAverage::iBlock ( ) const [inline]`

Get number of samples in current block average.
Returns 0 if block averaging is disabled (i.e., nSamplePerBlock == 0).

Returns

number of samples in current block (or 0 if disabled).

Definition at line 168 of file SymmTensorAverage.h.

12.214.3.11 isBlockComplete() `bool Util::SymmTensorAverage::isBlockComplete ( ) const [inline]`

Is the current block average complete?

Returns

(iBlock $>$ 0) && (iBlock == nSamplePerBlock)

Definition at line 174 of file SymmTensorAverage.h.
The documentation for this class was generated from the following files:

- SymmTensorAverage.h
- SymmTensorAverage.cpp

12.215 Util::Tensor Class Reference

A Tensor represents a Cartesian tensor.
`#include <Tensor.h>`

Public Member Functions

Constructors

- Tensor ()
  Default constructor.
- Tensor (const Tensor &t)
  Copy constructor.
- Tensor (double scalar)
  Constructor, initialize all elements to a scalar value.
- Tensor (const double a[ ][Dimension])
  Construct Tensor from double [][Dimension] 2D C array.

Assignment

- Tensor & operator= (const Tensor &t)
  Copy assignment.
- Tensor & operator= (const double a[ ][Dimension])
  Assignment from C double [Dimension][Dimension] 2D array.

Arithmetic Assignment

- void operator+= (const Tensor &dt)
  Add tensor dt to this tensor.
- void operator-= (const Tensor &dt)
  Subtract tensor dt from this tensor.
- void operator∗= (double s)
  Multiply this tensor by scalar s.
- void operator/= (double s)
  Divide this tensor by scalar s.

Array Subscript

- const double & operator() (int i, int j) const
  Return one element by value.
- double & operator() (int i, int j)
  Return one element by non-const reference.

Tensor valued functions (result assigned to invoking object)

- Tensor & zero ()
  Set all elements of this tensor to zero.
- Tensor & identity ()
  Set this to the identity (unity) tensor.
- Tensor & setRow (int i, const Vector &r)
  Set row i of this Tensor to elements of Vector r.
- Tensor & setColumn (int i, const Vector &r)
  Set column i of this Tensor to elements of Vector r.
- Tensor & add (const Tensor &t1, const Tensor &t2)
  Add tensors t1 and t2.
- Tensor & subtract (const Tensor &t1, const Tensor &t2)
  Subtract tensor t2 from t1.
- Tensor & multiply (const Tensor &t, double s)
  Multiply a tensor t by a scalar s.
- Tensor & divide (const Tensor &t, double s)
  Divide a Tensor t by a scalar s.
- Tensor & transpose (const Tensor &t)

Compute transpose of a tensor.
- Tensor & transpose ()

  Transpose this tensor.
- Tensor & symmetrize (const Tensor &t)

  Compute symmetric part of a tensor t.
- Tensor & symmetrize ()

  Symmetrize this tensor.
- Tensor & dyad (const Vector &v1, const Vector &v2)

  Create dyad of two vectors.

Miscellaneous

- double trace () const

  Return the trace of this tensor.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize this to/from an archive.

Static Members

- static const Tensor Zero = Tensor(0.0)

  Constant Tensor with all zero elements.
- static const Tensor Identity = Tensor().identity()

  Constant idenity Tensor (diagonal diagonal elements all 1).
- static void initStatic ()

  Call to guarantee initialization of Zero and Identity tensors.
- static void commitMpiType ()

  Commit MPI datatype MpiTraits<Tensor>::type.
- bool operator== (const Tensor &t1, const Tensor &t2)

  Equality for Tensors.
- bool operator== (const Tensor &t1, const double t2[ ][Dimension])

  Equality of Tensor and 2D C array.
- std::istream & operator>> (std::istream &in, Tensor &tensor)

  istream extractor for a Tensor.
- std::ostream & operator<< (std::ostream &out, const Tensor &tensor)

  ostream inserter for a Tensor.

12.215.1 Detailed Description

A Tensor represents a Cartesian tensor.
Definition at line 32 of file Tensor.h.

12.215.2 Constructor & Destructor Documentation

12.215.2.1 Tensor() [1/4] `Util::Tensor::Tensor ( ) [inline]`

Default constructor.
Definition at line 399 of file Tensor.h.

12.215.2.2 Tensor() [2/4] Util::Tensor::Tensor (

const Tensor & *t*) [inline]

Copy constructor.

Definition at line 406 of file Tensor.h.

References Util::DimensionSq.

12.215.2.3 Tensor() [3/4] Util::Tensor::Tensor (

double *scalar*) [inline], [explicit]

Constructor, initialize all elements to a scalar value.

Parameters

scalar	initial value for all elements.

Definition at line 417 of file Tensor.h.

References Util::DimensionSq.

12.215.2.4 Tensor() [4/4] Util::Tensor::Tensor (

const double *a[ ][Dimension]*) [inline], [explicit]

Construct Tensor from double [][Dimension] 2D C array.

Parameters

a	2D array a[Dimension][Dimension]

Definition at line 428 of file Tensor.h.

References Util::Dimension.

12.215.3 Member Function Documentation

12.215.3.1 operator=() [1/2] Tensor & Util::Tensor::operator= (

const Tensor & *t*) [inline]

Copy assignment.

Parameters

t	Tensor to assign.

Definition at line 468 of file Tensor.h.

References Util::DimensionSq.

12.215.3.2 operator=() [2/2] Tensor & Util::Tensor::operator= (

const double *a[ ][Dimension]*) [inline]

Assignment from C double [Dimension][Dimension] 2D array.

Parameters

a	2D array a[Dimension][Dimension]

Definition at line 480 of file Tensor.h.
References Util::Dimension.

12.215.3.3 operator+=() `void Util::Tensor::operator+= ( const Tensor & dt ) [inline]`

Add tensor dt to this tensor.
Upon return, ∗this = ∗this + dt.

Parameters

dt	tensor increment (input)

Definition at line 495 of file Tensor.h.
References Util::DimensionSq.

12.215.3.4 operator-=() `void Util::Tensor::operator-= ( const Tensor & dt ) [inline]`

Subtract tensor dt from this tensor.
Upon return, ∗this = ∗this - dt.

Parameters

dt	tensor increment (input)

Definition at line 506 of file Tensor.h.
References Util::DimensionSq.

12.215.3.5 operator∗=() `void Util::Tensor::operator*= ( double s ) [inline]`

Multiply this tensor by scalar s.
Upon return, ∗this = (∗this)∗s.

Parameters

s	scalar multiplier

Definition at line 517 of file Tensor.h.
References Util::DimensionSq.

12.215.3.6 operator/=() `void Util::Tensor::operator/= ( double s ) [inline]`

Divide this tensor by scalar s.
Upon return, ∗this = (∗this)/s.

Parameters

s	scalar divisor (input)

Definition at line 528 of file Tensor.h.
References Util::DimensionSq.

12.215.3.7 operator()() [1/2] `const double & Util::Tensor::operator() ( int i, int j ) const [inline]`

Return one element by value.

Parameters

i	row element index
j	column element index

Returns

element (i, j) of the tensor

Definition at line 539 of file Tensor.h.
References Util::Dimension.

12.215.3.8 operator()() [2/2] `double & Util::Tensor::operator() ( int i, int j ) [inline]`

Return one element by non-const reference.

Parameters

i	row element index
j	column element index

Returns

element i of the tensor

Definition at line 552 of file Tensor.h.
References Util::Dimension.

12.215.3.9 zero() `Tensor & Util::Tensor::zero ( ) [inline]`

Set all elements of this tensor to zero.

Returns

reference to this tensor

Definition at line 441 of file Tensor.h.
References Util::DimensionSq.
Referenced by Util::setToZero().

12.215.3.10 identity() `Tensor & Util::Tensor::identity ( ) [inline]`

Set this to the identity (unity) tensor.

Returns

reference to this tensor

Definition at line 453 of file Tensor.h.
References Util::Dimension, and Util::DimensionSq.

12.215.3.11 setRow() `Tensor & Util::Tensor::setRow ( int i, const Vector & r ) [inline]`

Set row i of this Tensor to elements of Vector r.

Returns

reference to this tensor

Definition at line 737 of file Tensor.h.
References Util::Dimension.

12.215.3.12 setColumn() `Tensor & Util::Tensor::setColumn ( int i, const Vector & r ) [inline]`

Set column i of this Tensor to elements of Vector r.

Returns

reference to this tensor

Definition at line 749 of file Tensor.h.
References Util::Dimension.

12.215.3.13 add() `Tensor & Util::Tensor::add ( const Tensor & t1, const Tensor & t2 ) [inline]`

Add tensors t1 and t2.
Upon return, ∗this = t1 + t2.

Parameters

t1	tensor
t2	tensor

Returns

reference to this tensor

Definition at line 567 of file Tensor.h.
References Util::DimensionSq.

12.215.3.14 subtract() `Tensor & Util::Tensor::subtract ( const Tensor & t1, const Tensor & t2 ) [inline]`

Subtract tensor t2 from t1.

Upon return, ∗this == t1 - t2.

Parameters

t1	tensor (input)
t2	tensor (input)

Returns

reference to this tensor

Definition at line 581 of file Tensor.h.
References Util::DimensionSq.

12.215.3.15 multiply() `Tensor & Util::Tensor::multiply ( const Tensor & t, double s ) [inline]`

Multiply a tensor t by a scalar s.
Upon return, ∗this == v∗s.

Parameters

t	tensor factor
s	scalar factor

Returns

reference to this tensor

Definition at line 595 of file Tensor.h.
References Util::DimensionSq.

12.215.3.16 divide() `Tensor & Util::Tensor::divide ( const Tensor & t, double s ) [inline]`

Divide a Tensor t by a scalar s.
Upon return, ∗this = v/s;

Parameters

t	tensor input
s	scalar denominator

Returns

reference to this tensor

Definition at line 609 of file Tensor.h.
References Util::DimensionSq.

12.215.3.17 transpose() [1/2]

`Tensor & Util::Tensor::transpose ( const Tensor & t ) [inline]`

Compute transpose of a tensor.

Upon return, ∗this is the transpose of t

Parameters

t	input tensor

Returns

reference to this tensor

Definition at line 636 of file Tensor.h.

References Util::Dimension.

12.215.3.18 transpose() [2/2]

`Tensor & Util::Tensor::transpose ( ) [inline]`

Transpose this tensor.

Upon return, ∗this is transposed.

Returns

reference to this tensor

Definition at line 657 of file Tensor.h.

References Util::Dimension.

12.215.3.19 symmetrize() [1/2]

`Tensor & Util::Tensor::symmetrize ( const Tensor & t ) [inline]`

Compute symmetric part of a tensor t.

Upon return, ∗this = [t + t.transpose()]/2

Parameters

t	tensor input

Returns

reference to this tensor

Definition at line 678 of file Tensor.h.

References Util::Dimension.

12.215.3.20 symmetrize() [2/2]

`Tensor & Util::Tensor::symmetrize ( ) [inline]`

Symmetrize this tensor.

Upon return, this is symmetrized, equal to half the sum of the original tensor and its transpose.

Returns

reference to this tensor

Definition at line 701 of file Tensor.h.

References Util::Dimension.

12.215.3.21 dyad() Tensor & Util::Tensor::dyad (
const Vector & *v1*,
const Vector & *v2*) [inline]

Create dyad of two vectors.

Upon return, ∗this equals the dyad v1 ^ v2. Equivalently: (∗this)(i , j) == v1[i]∗v2[j]

Parameters

v1	vector input
v2	vector input

Returns

reference to this tensor

Definition at line 763 of file Tensor.h.

References Util::Dimension.

12.215.3.22 trace() double Util::Tensor::trace ( ) const [inline]

Return the trace of this tensor.

Definition at line 621 of file Tensor.h.

References Util::Dimension.

12.215.3.23 serialize() template<class Archive >

void Util::Tensor::serialize (
Archive & *ar*,
const unsigned int *version*) [inline]

Serialize this to/from an archive.

Parameters

ar	archive
version	archive version id

Definition at line 720 of file Tensor.h.

References Util::DimensionSq.

12.215.3.24 initStatic() void Util::Tensor::initStatic ( ) [static]

Call to guarantee initialization of Zero and Identity tensors.

Call this function once to guarantee that this file is linked.

Definition at line 27 of file Tensor.cpp.

References UTIL_THROW.

Referenced by Util::initStatic().

12.215.3.25 commitMpiType() void Util::Tensor::commitMpiType ( ) [static]

Commit MPI datatype MpiTraits<Tensor>::type.

Definition at line 125 of file Tensor.cpp.

References Util::MpiStructBuilder::addMember(), Util::MpiStructBuilder::commit(), Util::Dimension, and Util::MpiStructBuilder::setBase().

12.215.4 Friends And Related Function Documentation

12.215.4.1 operator== [1/2]

```
bool operator== (
    const Tensor & t1,
    const Tensor & t2 )  [friend]
```

Equality for Tensors.

Definition at line 43 of file Tensor.cpp.

12.215.4.2 operator== [2/2]

```
bool operator== (
    const Tensor & t1,
    const double t2[ ][Dimension] )  [friend]
```

Equality of Tensor and 2D C array.

Definition at line 56 of file Tensor.cpp.

12.215.4.3 operator>>

```
std::istream& operator>> (
    std::istream & in,
    Tensor & tensor )  [friend]
```

istream extractor for a Tensor.

Input elements of a tensor from stream, without line breaks.

Parameters

in	input stream
tensor	Tensor to be read from stream

Returns

modified input stream

Definition at line 93 of file Tensor.cpp.

12.215.4.4 operator<<

```
std::ostream& operator<< (
    std::ostream & out,
    const Tensor & tensor )  [friend]
```

ostream inserter for a Tensor.

Output elements of a tensor to stream, without line breaks.

Parameters

out	output stream
tensor	Tensor to be written to stream

Returns

modified output stream

Definition at line 104 of file Tensor.cpp.

12.215.5 Member Data Documentation

12.215.5.1 Zero `const Tensor Util::Tensor::Zero = Tensor(0.0) [static]`

Constant Tensor with all zero elements.

Definition at line 297 of file Tensor.h.

12.215.5.2 Identity `const Tensor Util::Tensor::Identity = Tensor().identity() [static]`

Constant idenity Tensor (diagonal diagonal elements all 1).

Definition at line 302 of file Tensor.h.

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

- Tensor.h
- Tensor.cpp

12.216 Util::TensorAverage Class Reference

Calculates averages of all components of a Tensor-valued variable.

`#include <TensorAverage.h>`

Inheritance diagram for Util::TensorAverage:

classUtil_1_1TensorAverage-eps-converted-to.pdf

Public Member Functions

- TensorAverage (int blockFactor=2)

  Constructor.
- virtual ~TensorAverage ()

  Destructor.
- void setNSamplePerBlock (int nSamplePerBlock)

  Set nSamplePerBlock.
- void readParameters (std::istream &in)

  Read parameter nSamplePerBlock from file and initialize.
- virtual void loadParameters (Serializable::IArchive &ar)

  Load internal state from an archive.
- virtual void save (Serializable::OArchive &ar)

  Save internal state to an archive.
- template<class Archive >
  void serialize (Archive &ar, const unsigned int version)

  Serialize this to or from an archive.
- void clear ()

Clear all accumulators, set to empty initial state.

- void sample (const Tensor &value)

  Add a sampled value to the ensemble.
- const Average & operator() (int i, int j)

  Access the Average object for one tensor component.
- int nSamplePerBlock () const

  Get number of samples per block average.
- int iBlock () const

  Get number of samples in current block average.
- bool isBlockComplete () const

  Is the current block average complete?

Additional Inherited Members

12.216.1 Detailed Description

Calculates averages of all components of a Tensor-valued variable.
TensorAverage is a simple container for an array of Average objects, each of which calculates averages and error estimates for one component of a Tensor.
Definition at line 32 of file TensorAverage.h.

12.216.2 Constructor & Destructor Documentation

12.216.2.1 TensorAverage() `Util::TensorAverage::TensorAverage ( int blockFactor = 2 )`

Constructor.

Parameters

blockFactor	ratio of block sizes for subsequent stages.

Definition at line 21 of file TensorAverage.cpp.
References Util::Dimension, and Util::ParamComposite::setClassName().

12.216.2.2 ∼TensorAverage() `Util::TensorAverage::∼TensorAverage ( ) [virtual]`

Destructor.
Definition at line 40 of file TensorAverage.cpp.

12.216.3 Member Function Documentation

12.216.3.1 setNSamplePerBlock() `void Util::TensorAverage::setNSamplePerBlock ( int nSamplePerBlock )`

Set nSamplePerBlock.
If nSamplePerBlock > 0, the sample function will increment block averages, and reset the average every nSamplePerBlock samples.
If nSamplePerBlock == 0, block averaging is disabled. This is the default (i.e., the initial value set in the constructor).

Parameters

nSamplePerBlock	number of samples per block average output

Definition at line 46 of file TensorAverage.cpp.
References Util::Dimension, nSamplePerBlock(), and UTIL_THROW.

12.216.3.2 readParameters()

`void Util::TensorAverage::readParameters ( std::istream & in ) [virtual]`

Read parameter nSamplePerBlock from file and initialize.
See setNSamplePerBlock() for discussion of value.

Parameters

in	input stream

Reimplemented from Util::ParamComposite.
Definition at line 65 of file TensorAverage.cpp.
References Util::Dimension, and UTIL_THROW.

12.216.3.3 loadParameters()

`void Util::TensorAverage::loadParameters ( Serializable::IArchive & ar ) [virtual]`

Load internal state from an archive.

Parameters

ar	input/loading archive

Reimplemented from Util::ParamComposite.
Definition at line 84 of file TensorAverage.cpp.
References Util::Dimension, and UTIL_THROW.

12.216.3.4 save()

`void Util::TensorAverage::save ( Serializable::OArchive & ar ) [virtual]`

Save internal state to an archive.

Parameters

ar	output/saving archive

Reimplemented from Util::ParamComposite.
Definition at line 104 of file TensorAverage.cpp.

12.216.3.5 serialize()

```
template<class Archive >
void Util::TensorAverage::serialize (
            Archive & ar,
            const unsigned int version )
```

Serialize this to or from an archive.

Parameters

ar	input or output archive
version	file version id

Definition at line 185 of file TensorAverage.h.
References Util::Dimension.

12.216.3.6 clear() `void Util::TensorAverage::clear ( )`

Clear all accumulators, set to empty initial state.
Definition at line 110 of file TensorAverage.cpp.
References Util::Dimension.

12.216.3.7 sample() `void Util::TensorAverage::sample ( const Tensor & value )`

Add a sampled value to the ensemble.

Parameters

value	sampled value

Definition at line 126 of file TensorAverage.cpp.
References Util::Dimension.

12.216.3.8 operator()() `const Average & Util::TensorAverage::operator() ( int i, int j )`

Access the Average object for one tensor component.

Parameters

i	first index of associated tensor component
j	second index of associated tensor component

Returns

Average object associated with element (i, j)

Definition at line 147 of file TensorAverage.cpp.
References Util::Dimension.

12.216.3.9 nSamplePerBlock() `int Util::TensorAverage::nSamplePerBlock ( ) const [inline]`

Get number of samples per block average.
Returns zero if block averaging is disabled.

Returns

number of samples per block (or 0 if disabled).

Definition at line 166 of file TensorAverage.h.
Referenced by setNSamplePerBlock().

12.216.3.10 iBlock() `int Util::TensorAverage::iBlock ( ) const [inline]`

Get number of samples in current block average.
Returns 0 if block averaging is disabled (i.e., nSamplePerBlock == 0).

Returns

number of samples in current block (or 0 if disabled)

Definition at line 172 of file TensorAverage.h.

12.216.3.11 isBlockComplete() `bool Util::TensorAverage::isBlockComplete ( ) const [inline]`

Is the current block average complete?
Returns true iff blocking is enabled and iBlock == nSamplePerBlock

Returns

(iBlock > 0) && (iBlock == nSamplePerBlock)

Definition at line 178 of file TensorAverage.h.
The documentation for this class was generated from the following files:

- TensorAverage.h
- TensorAverage.cpp

12.217 Util::TextFileIArchive Class Reference

Loading archive for text istream.
`#include <TextFileIArchive.h>`

Public Member Functions

- TextFileIArchive ()
  Constructor.
- TextFileIArchive (std::string filename)
  Constructor.
- TextFileIArchive (std::ifstream &file)
  Constructor.
- virtual ~TextFileIArchive ()
  Destructor.
- std::ifstream & file ()
  Get the underlying ifstream by reference.
- template<typename T >
  TextFileIArchive & operator& (T &data)
  Load one object.
- template<typename T >
  TextFileIArchive & operator>> (T &data)

Load one object.

- template<typename T >
  void unpack (T &data)

  Load a single T object.
- template<typename T >
  void unpack (T ∗array, int n)

  Load a C-array of T objects.
- template<typename T >
  void unpack (T ∗array, int m, int n, int np)

  Load a 2D C array.

Static Public Member Functions

- static bool is_saving ()

  Returns true;.
- static bool is_loading ()

  Returns false;.

12.217.1 Detailed Description

Loading archive for text istream.
Definition at line 30 of file TextFileIArchive.h.

12.217.2 Constructor & Destructor Documentation

12.217.2.1 TextFileIArchive() [1/3] `Util::TextFileIArchive::TextFileIArchive ( )`

Constructor.
Definition at line 18 of file TextFileIArchive.cpp.

12.217.2.2 TextFileIArchive() [2/3] `Util::TextFileIArchive::TextFileIArchive ( std::string filename )`

Constructor.

Parameters

filename	name of file to open for reading.

Definition at line 27 of file TextFileIArchive.cpp.

12.217.2.3 TextFileIArchive() [3/3] `Util::TextFileIArchive::TextFileIArchive ( std::ifstream & file )`

Constructor.

Parameters

file	output file

Definition at line 37 of file TextFileIArchive.cpp.

References file(), and UTIL_THROW.

12.217.2.4 ∼TextFileIArchive() `Util::TextFileIArchive::∼TextFileIArchive ( ) [virtual]`

Destructor.

Definition at line 51 of file TextFileIArchive.cpp.

12.217.3 Member Function Documentation

12.217.3.1 is_saving() `bool Util::TextFileIArchive::is_saving ( ) [inline], [static]`

Returns true;.

Definition at line 125 of file TextFileIArchive.h.

12.217.3.2 is_loading() `bool Util::TextFileIArchive::is_loading ( ) [inline], [static]`

Returns false;.

Definition at line 128 of file TextFileIArchive.h.

12.217.3.3 file() `std::ifstream & Util::TextFileIArchive::file ( )`

Get the underlying ifstream by reference.

Definition at line 61 of file TextFileIArchive.cpp.

Referenced by TextFileIArchive().

12.217.3.4 operator&() `template<typename T >`

```
TextFileIArchive & Util::TextFileIArchive::operator& (
              T & data ) [inline]
```

Load one object.

Definition at line 137 of file TextFileIArchive.h.

12.217.3.5 operator>>() `template<typename T >`

```
TextFileIArchive & Util::TextFileIArchive::operator>> (
              T & data ) [inline]
```

Load one object.

Definition at line 147 of file TextFileIArchive.h.

12.217.3.6 unpack() [1/3] `template<typename T >`

```
void Util::TextFileIArchive::unpack (
              T & data ) [inline]
```

Load a single T object.

Parameters

data	object to be loaded from this archive.

Definition at line 159 of file TextFileIArchive.h.

12.217.3.7 unpack() [2/3]

```
template<typename T >
void Util::TextFileIArchive::unpack (
        T * array,
        int n ) [inline]
```

Load a C-array of T objects.

Parameters

array	pointer to array of T objecs.
n	number of elements in array

Definition at line 166 of file TextFileIArchive.h.

12.217.3.8 unpack() [3/3]

```
template<typename T >
void Util::TextFileIArchive::unpack (
        T * array,
        int m,
        int n,
        int np )
```

Load a 2D C array.

Parameters

array	pointer to first row or element
m	logical number of rows
n	logical number of columns
np	physical number of columns (elements allocated per row)

Definition at line 177 of file TextFileIArchive.h.

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

- TextFileIArchive.h
- TextFileIArchive.cpp

12.218 Util::TextFileOArchive Class Reference

Saving archive for character based ostream.

```
#include <TextFileOArchive.h>
```

Public Member Functions

- TextFileOArchive ()

  Constructor.
- TextFileOArchive (std::string filename)

  Constructor.
- TextFileOArchive (std::ofstream &file)

  Constructor.
- virtual ∼TextFileOArchive ()

  Destructor.
- std::ofstream & file ()

Get the underlying ifstream by reference.

- template<typename T >
  TextFileOArchive & operator& (T &data)

    Save one T object to this archive.
- template<typename T >
  TextFileOArchive & operator<< (T &data)

    Save one T object to this archive.
- template<typename T >
  void pack (const T &data)

    Save one T object to this archive.
- template<typename T >
  void pack (const T *array, int n)

    Save a C-array of T objects to this archive.
- template<typename T >
  void pack (const T *array, int m, int n, int np)

    Save a 2D C array to this archive.

Static Public Member Functions

- static bool is_saving ()

    Returns true;.
- static bool is_loading ()

    Returns false;.

12.218.1 Detailed Description

Saving archive for character based ostream.
Definition at line 30 of file TextFileOArchive.h.

12.218.2 Constructor & Destructor Documentation

12.218.2.1 TextFileOArchive() [1/3] `Util::TextFileOArchive::TextFileOArchive ( )`

Constructor.
Definition at line 16 of file TextFileOArchive.cpp.

12.218.2.2 TextFileOArchive() [2/3] `Util::TextFileOArchive::TextFileOArchive ( std::string filename )`

Constructor.

Parameters

filename	name of file to open for reading.

Definition at line 25 of file TextFileOArchive.cpp.

12.218.2.3 TextFileOArchive() [3/3] `Util::TextFileOArchive::TextFileOArchive ( std::ofstream & file )`

Constructor.

Parameters

file	output file

Definition at line 35 of file TextFileOArchive.cpp.
References file(), and UTIL_THROW.

12.218.2.4 ∼TextFileOArchive() `Util::TextFileOArchive::∼TextFileOArchive ( ) [virtual]`

Destructor.
Definition at line 48 of file TextFileOArchive.cpp.

12.218.3 Member Function Documentation

12.218.3.1 is_saving() `bool Util::TextFileOArchive::is_saving ( ) [inline], [static]`

Returns true;.
Definition at line 125 of file TextFileOArchive.h.

12.218.3.2 is_loading() `bool Util::TextFileOArchive::is_loading ( ) [inline], [static]`

Returns false;.
Definition at line 128 of file TextFileOArchive.h.

12.218.3.3 file() `std::ofstream & Util::TextFileOArchive::file ( )`

Get the underlying ifstream by reference.
Definition at line 58 of file TextFileOArchive.cpp.
Referenced by TextFileOArchive().

12.218.3.4 operator&() `template<typename T >`

```
TextFileOArchive & Util::TextFileOArchive::operator& (
            T & data ) [inline]
```

Save one T object to this archive.
Definition at line 137 of file TextFileOArchive.h.

12.218.3.5 operator<<() `template<typename T >`

```
TextFileOArchive & Util::TextFileOArchive::operator<< (
            T & data ) [inline]
```

Save one T object to this archive.
Definition at line 147 of file TextFileOArchive.h.

12.218.3.6 pack() [1/3] `template<typename T >`

```
void Util::TextFileOArchive::pack (
            const T & data ) [inline]
```

Save one T object to this archive.

Parameters

data	object to be written to file

Definition at line 159 of file TextFileOArchive.h.

12.218.3.7 pack() [2/3]

```
template<typename T >
void Util::TextFileOArchive::pack (
        const T * array,
        int n ) [inline]
```

Save a C-array of T objects to this archive.

Parameters

array	C array of T objects (pointer to first element)
n	number of elements

Definition at line 178 of file TextFileOArchive.h.

12.218.3.8 pack() [3/3]

```
template<typename T >
void Util::TextFileOArchive::pack (
        const T * array,
        int m,
        int n,
        int np ) [inline]
```

Save a 2D C array to this archive.

Parameters

array	address of first element array[0][0] of 2D array
m	logical number of rows
n	logical number of columns
np	physical number of columns

Definition at line 190 of file TextFileOArchive.h.

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

- TextFileOArchive.h
- TextFileOArchive.cpp

12.219 Util::Timer Class Reference

Wall clock timer.

`#include <Timer.h>`

Public Member Functions

- Timer ()

  Default constructor.
- void start (TimePoint begin)

Start timing from an externally supplied time.
- void start ()

 Start timing from now (internally computed).
- void stop (TimePoint end)

 Stop the clock at an externally supplied time.
- void stop ()

 Stop the clock now (internally supplied).
- bool isRunning ()

 Is this Timer running?
- void clear ()

 Reset accumulated time to zero.
- double time ()

 Return the accumulated time, in seconds.

Static Public Member Functions

- static TimePoint now ()

 Return current time point.

12.219.1 Detailed Description

Wall clock timer.
A Timer keeps track of the time elapsed during one or more interval. Each interval begins when start() is called and ends when stop() is called. If start() and stop() are invoked repeatedly, the timer accumulates the time elapses in multiple intervals. The accumulated time is returned by the time() method, and can be reset to zero by the clear() method.
Definition at line 34 of file Timer.h.

12.219.2 Constructor & Destructor Documentation

12.219.2.1 Timer() `Util::Timer::Timer ( )`

Default constructor.
Constructor.
Definition at line 16 of file Timer.cpp.
References clear().

12.219.3 Member Function Documentation

12.219.3.1 start() [1/2] `void Util::Timer::start ( TimePoint begin ) [inline]`

Start timing from an externally supplied time.
Set start time and set isRunning = true.

Parameters

begin	starting TimePoint.

Definition at line 134 of file Timer.h.
References UTIL_THROW.

Referenced by Pscf::Pspg::Continuous::AmIterator< D >::solve().

12.219.3.2 start() [2/2] `void Util::Timer::start ( ) [inline]`

Start timing from now (internally computed).

Set start time and set isRunning = true.

Definition at line 147 of file Timer.h.

References now(), and UTIL_THROW.

12.219.3.3 stop() [1/2] `void Util::Timer::stop ( TimePoint end ) [inline]`

Stop the clock at an externally supplied time.

Increment accumulated time, set isRunning = false.

Definition at line 159 of file Timer.h.

References UTIL_THROW.

Referenced by Pscf::Pspg::Continuous::AmIterator< D >::solve().

12.219.3.4 stop() [2/2] `void Util::Timer::stop ( ) [inline]`

Stop the clock now (internally supplied).

Increment accumulated time, set isRunning = false.

Definition at line 175 of file Timer.h.

References now().

12.219.3.5 isRunning() `bool Util::Timer::isRunning ( ) [inline]`

Is this Timer running?

Definition at line 194 of file Timer.h.

12.219.3.6 clear() `void Util::Timer::clear ( ) [inline]`

Reset accumulated time to zero.

Definition at line 181 of file Timer.h.

Referenced by Timer().

12.219.3.7 time() `double Util::Timer::time ( )`

Return the accumulated time, in seconds.

Definition at line 23 of file Timer.cpp.

Referenced by Pscf::Pspg::Continuous::AmIterator< D >::solve().

12.219.3.8 now() `Timer::TimePoint Util::Timer::now ( ) [inline], [static]`

Return current time point.

Return current time point (static function)

Definition at line 123 of file Timer.h.

Referenced by Pscf::Pspg::Continuous::AmIterator< D >::solve(), start(), and stop().

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

- Timer.h
- Timer.cpp

12.220 Util::Vector Class Reference

A Vector is a Cartesian vector.

`#include <Vector.h>`

Public Member Functions

Constructors

- Vector ()

  Default constructor.
- Vector (const Vector &v)

  Copy constructor.
- Vector (double scalar)

  Constructor, initialize all elements to a scalar value.
- Vector (const double *v)

  Construct Vector from C double[3] array.
- Vector (double x, double y, double z=0.0)

  Construct Vector from its coordinates.
- Vector & zero ()

  Set all elements of a 3D vector to zero.
- template<class Archive >

  void serialize (Archive &ar, const unsigned int version)

  Serialize to/from an archive.

Assignment

- Vector & operator= (const Vector &v)

  Copy assignment.
- Vector & operator= (const double *v)

  Assignment from C double[3] array.

Arithmetic Assignment

- void operator+= (const Vector &dv)

  Add vector dv to this vector.
- void operator-= (const Vector &dv)

  Subtract vector dv from this vector.
- void operator*= (double s)

  Multiply this vector by scalar s.
- void operator/= (double s)

  Divide this vector by scalar s.

Array Subscript

- const double & operator[ ] (int i) const

  Return one Cartesian element by value.
- double & operator[ ] (int i)

  Return one element of the vector by references.

Scalar-valued functions

- double square () const

  Return square magnitude of this vector.
- double abs () const

  Return absolute magnitude of this vector.
- double dot (const Vector &v) const

Return dot product of this vector and vector v.

- double projection (const Vector &p) const

  Return projection of this vector along vector p.

Vector valued functions (result assigned to invoking object)

- Vector & add (const Vector &v1, const Vector &v2)

  Add vectors v1 and v2.
- Vector & subtract (const Vector &v1, const Vector &v2)

  Subtract vector v2 from v1.
- Vector & multiply (const Vector &v, double s)

  Multiply a vector v by a scalar s.
- Vector & divide (const Vector &v, double s)

  Divide vector v by scalar s.
- Vector & cross (const Vector &v1, const Vector &v2)

  Calculate cross product of vectors v1 and v2.
- Vector & versor (const Vector &v)

  Calculate unit vector parallel to input vector v.
- Vector & parallel (const Vector &v, const Vector &p)

  Calculate component of vector v parallel to vector p.
- Vector & transverse (const Vector &v, const Vector &p)

  Calculate component of vector v transverse to vector p.
- int minId (const Vector &v)

  Computes the index corresponding to minimum element in a vector.
- int maxId (const Vector &v)

  Computes the index corresponding to maximum element in a vector.

Static Members

- static const Vector Zero = Vector(0.0)

  Zero Vector = {0.0, 0.0, 0.0}.
- static void initStatic ()

  Initialize Zero Vector.
- static void commitMpiType ()

  Commit MPI datatype MpiTraits< Vector >::type.
- bool operator== (const Vector &v1, const Vector &v2)

  Equality for Vectors.
- bool operator== (const Vector &v1, const double ∗v2)

  Equality of Vector and C array.
- std::istream & operator>> (std::istream &in, Vector &vector)

  istream extractor for a Vector.
- std::ostream & operator<< (std::ostream &out, const Vector &vector)

  ostream inserter for a Vector.

12.220.1 Detailed Description

A Vector is a Cartesian vector.

The Cartesian elements of a Vector can be accessed using array notation: The elements of a three dimensional Vector v are v[0], v[1], and v[2]. The subscript operator [] returns elements as references, which can be used on either the left or right side of an assignment operator.

The arithmetic assignment operators +=, -=, ∗=, and /= are overloaded. The operators += and -= represent increment or decrement by a vector, while ∗= and /= represent multiplication or division by a scalar.

All other unary and binary mathematical operations are implemented as methods. Operations that yield a scalar result, such as a dot product, are implemented as methods that return the resulting value. Operations that yield a Vector, such as vector addition, are implemented by methods that assign the result to the invoking vector, and return a reference to the invoking vector. For example,

```
Vector a, b, c;
double s;
a[0] = 0.0
a[1] = 1.0
a[2] = 2.0
b[0] =  0.5
b[1] = -0.5
b[2] = -1.5
// Set s = a.b
s = a.dot(b)
// Set c = a + b
c.add(a, b)
// Set a = a + b
a += b
// Set b = b*2
b *= 2
```

This syntax for Vector valued operations avoids dynamic allocation of temporary Vector objects, by requiring that the invoking function provide an object to hold the result.

For efficiency, all methods in this class are declared inline.

Definition at line 75 of file Vector.h.

12.220.2 Constructor & Destructor Documentation

12.220.2.1 Vector() [1/5] `Util::Vector::Vector ( ) [inline]`

Default constructor.

Definition at line 463 of file Vector.h.

12.220.2.2 Vector() [2/5] `Util::Vector::Vector ( const Vector & v ) [inline]`

Copy constructor.

Parameters

v	Vector to be copied

Definition at line 470 of file Vector.h.

12.220.2.3 Vector() [3/5] `Util::Vector::Vector ( double scalar ) [inline], [explicit]`

Constructor, initialize all elements to a scalar value.

Parameters

scalar	initial value for all elements.

Definition at line 481 of file Vector.h.

12.220.2.4 Vector() [4/5] `Util::Vector::Vector ( const double * v ) [inline], [explicit]`

Construct Vector from C double[3] array.

Parameters

v	array of 3 coordinates

Definition at line 492 of file Vector.h.

12.220.2.5 Vector() [5/5]

```
Util::Vector::Vector (
        double x,
        double y,
        double z = 0.0 ) [inline]
```

Construct Vector from its coordinates.

Parameters

x	x-axis coordinate, v[0]
y	y-axis coordinate, v[1]
z	z-axis coordinate, v[2]

Definition at line 503 of file Vector.h.

12.220.3 Member Function Documentation

12.220.3.1 zero()

```
Vector & Util::Vector::zero ( ) [inline]
```

Set all elements of a 3D vector to zero.
Definition at line 514 of file Vector.h.
Referenced by Util::setToZero().

12.220.3.2 serialize()

```
template<class Archive >
void Util::Vector::serialize (
        Archive & ar,
        const unsigned int version ) [inline]
```

Serialize to/from an archive.

Parameters

ar	archive
version	archive version id

Definition at line 818 of file Vector.h.

12.220.3.3 operator=() [1/2]

```
Vector & Util::Vector::operator= (
        const Vector & v ) [inline]
```

Copy assignment.

Parameters

v	Vector to assign.

Definition at line 526 of file Vector.h.

12.220.3.4 operator=() [2/2]

```
Vector & Util::Vector::operator= (
    const double * v ) [inline]
```

Assignment from C double[3] array.

Parameters

v	C-array of components

Definition at line 538 of file Vector.h.

12.220.3.5 operator+=()

```
void Util::Vector::operator+= (
    const Vector & dv ) [inline]
```

Add vector dv to this vector.
Upon return, ∗this = this + dv.

Parameters

dv	vector increment (input)

Definition at line 550 of file Vector.h.

12.220.3.6 operator-=()

```
void Util::Vector::operator-= (
    const Vector & dv ) [inline]
```

Subtract vector dv from this vector.
Upon return, ∗this = this + dv.

Parameters

dv	vector increment (input)

Definition at line 561 of file Vector.h.

12.220.3.7 operator∗=()

```
void Util::Vector::operator*= (
    double s ) [inline]
```

Multiply this vector by scalar s.
Upon return, ∗this = (∗this)∗s.

Parameters

s	scalar multiplier

Definition at line 572 of file Vector.h.

12.220.3.8 operator/=() `void Util::Vector::operator/= ( double s ) [inline]`

Divide this vector by scalar s.
Upon return, ∗this = (∗this)/s.

Parameters

s	scalar divisor (input)

Definition at line 583 of file Vector.h.

12.220.3.9 operator[]() [1/2] `const double & Util::Vector::operator[] ( int i ) const [inline]`

Return one Cartesian element by value.

Parameters

i	element index

Returns

element i of the vector

Definition at line 594 of file Vector.h.
References Util::Dimension.

12.220.3.10 operator[]() [2/2] `double & Util::Vector::operator[] ( int i ) [inline]`

Return one element of the vector by references.

Parameters

i	element index

Returns

element i of this vector

Definition at line 605 of file Vector.h.
References Util::Dimension.

12.220.3.11 square() `double Util::Vector::square ( ) const [inline]`

Return square magnitude of this vector.

Returns

square magnitude of this vector

Definition at line 616 of file Vector.h.
Referenced by abs(), parallel(), and transverse().

12.220.3.12 abs() `double Util::Vector::abs ( ) const [inline]`

Return absolute magnitude of this vector.

Returns

absolute magnitude (norm) of this vector.

Definition at line 625 of file Vector.h.
References square().
Referenced by projection(), and versor().

12.220.3.13 dot() `double Util::Vector::dot ( const Vector & v ) const [inline]`

Return dot product of this vector and vector v.

Parameters

v	input vector

Returns

dot product of this vector and vector v

Definition at line 632 of file Vector.h.
Referenced by parallel(), Util::product(), projection(), and transverse().

12.220.3.14 projection() `double Util::Vector::projection ( const Vector & p ) const [inline]`

Return projection of this vector along vector p.

Parameters

p	vector parallel to direction along which to project

Returns

scalar projection this->dot(p)/p.abs()

Definition at line 641 of file Vector.h.
References abs(), and dot().

12.220.3.15 add() `Vector & Util::Vector::add ( const Vector & v1, const Vector & v2 ) [inline]`

Add vectors v1 and v2.
Upon return, ∗this = v1 + v2.

Parameters

v1	vector (input)
v2	vector (input)

Returns

modified invoking vector

Definition at line 657 of file Vector.h.

12.220.3.16 subtract() `Vector & Util::Vector::subtract ( const Vector & v1, const Vector & v2 ) [inline]`

Subtract vector v2 from v1.
Upon return, ∗this = v1 - v2.

Parameters

v1	vector (input)
v2	vector (input)

Returns

modified invoking vector

Definition at line 672 of file Vector.h.

12.220.3.17 multiply() `Vector & Util::Vector::multiply ( const Vector & v, double s ) [inline]`

Multiply a vector v by a scalar s.
Upon return, ∗this = v∗s.

Parameters

v	vector input
s	scalar input

Returns

modified invoking vector

Definition at line 686 of file Vector.h.

12.220.3.18 divide() `Vector & Util::Vector::divide ( const Vector & v,`

double *s*) [inline]

Divide vector v by scalar s.

Upon return, ∗this = v/s;

Parameters

v	vector input
s	scalar input

Returns

modified invoking vector

Definition at line 700 of file Vector.h.

12.220.3.19 cross()

```
Vector & Util::Vector::cross (
    const Vector & v1,
    const Vector & v2 ) [inline]
```

Calculate cross product of vectors v1 and v2.

Upon return, ∗this = v1 x v2.

Parameters

v1	input vector
v2	input vector

Returns

modified invoking vector

Definition at line 714 of file Vector.h.

12.220.3.20 versor()

```
Vector & Util::Vector::versor (
    const Vector & v ) [inline]
```

Calculate unit vector parallel to input vector v.

Upon return ∗this = unit vector.

Parameters

v	input vector

Returns

modified invoking Vector

Definition at line 726 of file Vector.h.

References abs().

12.220.3.21 parallel()

```
Vector & Util::Vector::parallel (
    const Vector & v,
    const Vector & p ) [inline]
```

Calculate component of vector v parallel to vector p.
Upon return, the invoking vector is equal to the vector projection of vector v along a direction parallel to vector p.
The vector projection of v along p is parallel to p and has an absolute magnitude equal to the scalar projection of v along p.

Parameters

v	vector to project
p	vector along which to project

Returns

modified invoking Vector

Definition at line 749 of file Vector.h.
References dot(), and square().

12.220.3.22 transverse()

```
Vector & Util::Vector::transverse (
    const Vector & v,
    const Vector & p ) [inline]
```

Calculate component of vector v transverse to vector p.
Upon return, the invoking vector is equal to the vector projection of vector v perpendicular to vector p.

Parameters

v	input vector
p	vector perpendicular to which to project.

Returns

modified invoking Vector

Definition at line 771 of file Vector.h.
References dot(), and square().

12.220.3.23 minId()

```
int Util::Vector::minId (
    const Vector & v )
```

Computes the index corresponding to minimum element in a vector.

Parameters

v	input vector

Returns

index of the minimum element.

12.220.3.24 maxId()

```
int Util::Vector::maxId (
    const Vector & v )
```

Computes the index corresponding to maximum element in a vector.

Parameters

v	input vector

Returns

index of the maximum element.

12.220.3.25 initStatic() `void Util::Vector::initStatic ( ) [static]`

Initialize Zero Vector.

Definition at line 119 of file Vector.cpp.

Referenced by Util::initStatic().

12.220.3.26 commitMpiType() `void Util::Vector::commitMpiType ( ) [static]`

Commit MPI datatype MpiTraits<Vector>::type.

Commit MPI Datatype.

Definition at line 97 of file Vector.cpp.

References Util::MpiStructBuilder::addMember(), Util::MpiStructBuilder::commit(), and Util::MpiStructBuilder::setBase().

12.220.4 Friends And Related Function Documentation

12.220.4.1 operator== [1/2]

```
bool operator== (
    const Vector & v1,
    const Vector & v2 ) [friend]
```

Equality for Vectors.

Definition at line 26 of file Vector.cpp.

12.220.4.2 operator== [2/2]

```
bool operator== (
    const Vector & v1,
    const double * v2 ) [friend]
```

Equality of Vector and C array.

Definition at line 36 of file Vector.cpp.

12.220.4.3 operator>>

```
std::istream& operator>> (
    std::istream & in,
    Vector & vector ) [friend]
```

istream extractor for a Vector.

Input elements of a vector from stream, without line breaks.

Parameters

in	input stream
vector	Vector to be read from stream

Returns

modified input stream

Definition at line 65 of file Vector.cpp.

12.220.4.4 operator<< `std::ostream& operator<< ( std::ostream & out, const Vector & vector ) [friend]`

ostream inserter for a Vector.
Output elements of a vector to stream, without line breaks.

Parameters

out	output stream
vector	Vector to be written to stream

Returns

modified output stream

Definition at line 76 of file Vector.cpp.

12.220.5 Member Data Documentation

12.220.5.1 Zero `const Vector Util::Vector::Zero = Vector(0.0) [static]`

Zero Vector = {0.0, 0.0, 0.0}.
Definition at line 369 of file Vector.h.
The documentation for this class was generated from the following files:

- Vector.h
- Vector.cpp

12.221 Util::XdrFileIArchive Class Reference

Loading / input archive for binary XDR file.
`#include <XdrFileIArchive.h>`

Public Member Functions

- XdrFileIArchive ()

  Constructor.
- XdrFileIArchive (std::string filename)

  Constructor.
- XdrFileIArchive (std::ofstream &file)

  Constructor.
- virtual ∼XdrFileIArchive ()

  Destructor.
- void init (FILE ∗file)

  Initialize by associating with an open file.

- template<typename T >
  XdrFileIArchive & operator& (T &data)

  Load one object.
- template<typename T >
  XdrFileIArchive & operator>> (T &data)

  Load one object.
- FILE ∗ file ()

  Get the underlying file handle.
- XDR ∗ xdrPtr ()

  Get a pointer to the enclosed XDR object.

Static Public Member Functions

- static bool is_saving ()

  Returns false.
- static bool is_loading ()

  Returns true.

12.221.1 Detailed Description

Loading / input archive for binary XDR file.
XDR is a standard protocol for writing and reading binary in a portable format. This archive saves data to an associated file in XDR format. It depends on the unix xdr library <rpc/xdr.h>. Because this library is written in C (not C++), this archive uses a standard C library file handle, not a C++ iostream.
Definition at line 39 of file XdrFileIArchive.h.

12.221.2 Constructor & Destructor Documentation

12.221.2.1 XdrFileIArchive() [1/3] `Util::XdrFileIArchive::XdrFileIArchive ( )`

Constructor.
Definition at line 17 of file XdrFileIArchive.cpp.

12.221.2.2 XdrFileIArchive() [2/3] `Util::XdrFileIArchive::XdrFileIArchive ( std::string filename )`

Constructor.

Parameters

filename	name of file to open for reading.

Definition at line 27 of file XdrFileIArchive.cpp.
References UTIL_THROW.

12.221.2.3 XdrFileIArchive() [3/3] `Util::XdrFileIArchive::XdrFileIArchive ( std::ofstream & file )`

Constructor.

Parameters

file	output file

12.221.2.4 ∼XdrFileIArchive() `Util::XdrFileIArchive::∼XdrFileIArchive ( ) [virtual]`

Destructor.

Definition at line 45 of file XdrFileIArchive.cpp.

12.221.3 Member Function Documentation

12.221.3.1 is_saving() `bool Util::XdrFileIArchive::is_saving ( ) [inline], [static]`

Returns false.

Definition at line 121 of file XdrFileIArchive.h.

12.221.3.2 is_loading() `bool Util::XdrFileIArchive::is_loading ( ) [inline], [static]`

Returns true.

Definition at line 124 of file XdrFileIArchive.h.

12.221.3.3 init() `void Util::XdrFileIArchive::init ( FILE * file )`

Initialize by associating with an open file.

Parameters

file	C library file handle, must be open for reading.

Definition at line 51 of file XdrFileIArchive.cpp.

References file().

12.221.3.4 operator&() `template<typename T > XdrFileIArchive & Util::XdrFileIArchive::operator& ( T & data ) [inline]`

Load one object.

Definition at line 133 of file XdrFileIArchive.h.

12.221.3.5 operator>>() `template<typename T > XdrFileIArchive & Util::XdrFileIArchive::operator>> ( T & data ) [inline]`

Load one object.

Definition at line 143 of file XdrFileIArchive.h.

12.221.3.6 file() `FILE * Util::XdrFileIArchive::file ( ) [inline]`

Get the underlying file handle.

Definition at line 152 of file XdrFileIArchive.h.
Referenced by init().

12.221.3.7 xdrPtr() `XDR * Util::XdrFileIArchive::xdrPtr ( ) [inline]`

Get a pointer to the enclosed XDR object.
Definition at line 158 of file XdrFileIArchive.h.
The documentation for this class was generated from the following files:

- XdrFileIArchive.h
- XdrFileIArchive.cpp

12.222 Util::XdrFileOArchive Class Reference

Saving / output archive for binary XDR file.
`#include <XdrFileOArchive.h>`

Public Member Functions

- XdrFileOArchive ()

  Constructor.
- XdrFileOArchive (std::string filename)

  Constructor.
- virtual ∼XdrFileOArchive ()

  Destructor.
- void init (FILE ∗file)

  Associate with an open file and initialize.
- FILE ∗ file ()

  Get the underlying ifstream by reference.
- template<typename T >
  XdrFileOArchive & operator& (T &data)

  Save one object.
- template<typename T >
  XdrFileOArchive & operator<< (T &data)

  Save one object.
- XDR ∗ xdrPtr ()

  Get a pointer to the enclosed XDR object.

Static Public Member Functions

- static bool is_saving ()

  Returns true;.
- static bool is_loading ()

  Returns false;.

12.222.1 Detailed Description

Saving / output archive for binary XDR file.
XDR is a standard protocol for writing and reading binary in a portable format. This archive saves data to an associated file in XDR format. It depends on the unix xdr library <rpc/xdr.h>. Because this library is written in C (not C++), this archive uses a standard C library file handle, not a C++ iostream.
Definition at line 39 of file XdrFileOArchive.h.

12.222.2 Constructor & Destructor Documentation

12.222.2.1 XdrFileOArchive() [1/2] `Util::XdrFileOArchive::XdrFileOArchive ( )`

Constructor.

Definition at line 17 of file XdrFileOArchive.cpp.

12.222.2.2 XdrFileOArchive() [2/2] `Util::XdrFileOArchive::XdrFileOArchive ( std::string filename )`

Constructor.

Parameters

filename	name of file to open for reading.

Definition at line 26 of file XdrFileOArchive.cpp.

References UTIL_THROW.

12.222.2.3 ∼XdrFileOArchive() `Util::XdrFileOArchive::∼XdrFileOArchive ( ) [virtual]`

Destructor.

Definition at line 43 of file XdrFileOArchive.cpp.

12.222.3 Member Function Documentation

12.222.3.1 is_saving() `bool Util::XdrFileOArchive::is_saving ( ) [inline], [static]`

Returns true;.

Definition at line 111 of file XdrFileOArchive.h.

12.222.3.2 is_loading() `bool Util::XdrFileOArchive::is_loading ( ) [inline], [static]`

Returns false;.

Definition at line 114 of file XdrFileOArchive.h.

12.222.3.3 init() `void Util::XdrFileOArchive::init ( FILE * file )`

Associate with an open file and initialize.

Parameters

file	C file handle, must be open for writing.

Definition at line 49 of file XdrFileOArchive.cpp.

References file().

12.222.3.4 file() `FILE * Util::XdrFileOArchive::file ( ) [inline]`

Get the underlying ifstream by reference.
Definition at line 142 of file XdrFileOArchive.h.
Referenced by init().

12.222.3.5 operator&() `template<typename T >`

`XdrFileOArchive & Util::XdrFileOArchive::operator& ( T & data ) [inline]`

Save one object.
Definition at line 123 of file XdrFileOArchive.h.

12.222.3.6 operator<<() `template<typename T >`

`XdrFileOArchive & Util::XdrFileOArchive::operator<< ( T & data ) [inline]`

Save one object.
Definition at line 133 of file XdrFileOArchive.h.

12.222.3.7 xdrPtr() `XDR * Util::XdrFileOArchive::xdrPtr ( ) [inline]`

Get a pointer to the enclosed XDR object.
Definition at line 148 of file XdrFileOArchive.h.
The documentation for this class was generated from the following files:

- XdrFileOArchive.h
- XdrFileOArchive.cpp

12.223 Util::XmlAttribute Class Reference

Parser for an XML attribute.
`#include <XmlAttribute.h>`
Inheritance diagram for Util::XmlAttribute:

classUtil_1_1XmlAttribute-eps-converted-to.pdf

Public Member Functions

- XmlAttribute ()

  Constructor.
- virtual ~XmlAttribute ()

  Destructor.
- bool match (const std::string &string, int begin)

  Return true if an attribute is found, false otherwise.
- bool match (XmlBase &parser)

  If successful return true and advance cursor or parent parser.
- const std::string & label ()

  Return label string.
- std::stringstream & value ()

  Return value string, without quotes.

12.223.1 Detailed Description

Parser for an XML attribute.
Definition at line 23 of file XmlAttribute.h.

12.223.2 Constructor & Destructor Documentation

12.223.2.1 XmlAttribute() `Util::XmlAttribute::XmlAttribute ( )`

Constructor.
Definition at line 14 of file XmlAttribute.cpp.

12.223.2.2 ∼XmlAttribute() `Util::XmlAttribute::∼XmlAttribute ( ) [virtual]`

Destructor.
Definition at line 23 of file XmlAttribute.cpp.

12.223.3 Member Function Documentation

12.223.3.1 match() [1/2] `bool Util::XmlAttribute::match ( const std::string & string, int begin )`

Return true if an attribute is found, false otherwise.
Definition at line 26 of file XmlAttribute.cpp.
References Util::XmlBase::c(), Util::XmlBase::cursor(), Util::XmlBase::isEnd(), Util::XmlBase::next(), Util::rStrip(), Util↩ ::XmlBase::setString(), Util::XmlBase::skip(), and Util::XmlBase::string().
Referenced by match(), and Util::XmlStartTag::matchAttribute().

12.223.3.2 match() [2/2] `bool Util::XmlAttribute::match ( XmlBase & parser )`

If successful return true and advance cursor or parent parser.

Parameters

parser	parent parser object

Definition at line 82 of file XmlAttribute.cpp.
References Util::XmlBase::cursor(), match(), Util::XmlBase::setCursor(), and Util::XmlBase::string().

12.223.3.3 label() `const std::string& Util::XmlAttribute::label ( ) [inline]`

Return label string.
Definition at line 53 of file XmlAttribute.h.
Referenced by Util::XmlXmlTag::match().

12.223.3.4 value() `std::stringstream& Util::XmlAttribute::value ( ) [inline]`

Return value string, without quotes.
Definition at line 59 of file XmlAttribute.h.

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

- XmlAttribute.h
- XmlAttribute.cpp

12.224 Util::XmlBase Class Reference

Base class for classes that parse XML markup tags.

```
#include <XmlBase.h>
```

Inheritance diagram for Util::XmlBase:

classUtil_1_1XmlBase-eps-converted-to.pdf

Public Member Functions

- XmlBase ()

  Constructor.
- ∼XmlBase ()

  Destructor.
- void setString (const std::string &string, int cursor=0)

  Initialize string and cursor.
- void setCursor (int cursor)

  Set cursor.
- void skip ()

  Skip leading white space, if any.
- void next ()

  Advance to the next character.
- const std::string & string () const

  Return the associated string.
- int cursor () const

  Return the index of the current character.
- int c () const

  Return the current character.
- bool isEnd () const

  Has the cursor reached the end of the string?

12.224.1 Detailed Description

Base class for classes that parse XML markup tags.
Definition at line 22 of file XmlBase.h.

12.224.2 Constructor & Destructor Documentation

12.224.2.1 XmlBase() `Util::XmlBase::XmlBase ( )`

Constructor.

Definition at line 16 of file XmlBase.cpp.

12.224.2.2 ∼XmlBase() `Util::XmlBase::∼XmlBase ( )`

Destructor.

Definition at line 26 of file XmlBase.cpp.

12.224.3 Member Function Documentation

12.224.3.1 setString() `void Util::XmlBase::setString (`

```
const std::string & string,
int cursor = 0 )
```

Initialize string and cursor.

Definition at line 32 of file XmlBase.cpp.

References cursor(), setCursor(), and string().

Referenced by Util::XmlAttribute::match(), Util::XmlEndTag::match(), and Util::XmlStartTag::matchLabel().

12.224.3.2 setCursor() `void Util::XmlBase::setCursor (`

```
int cursor )
```

Set cursor.

String must already be set.

Definition at line 42 of file XmlBase.cpp.

References cursor(), and UTIL_THROW.

Referenced by Util::XmlAttribute::match(), and setString().

12.224.3.3 skip() `void Util::XmlBase::skip ( ) [inline]`

Skip leading white space, if any.

Definition at line 88 of file XmlBase.h.

Referenced by Util::XmlAttribute::match(), Util::XmlEndTag::match(), Util::XmlStartTag::matchAttribute(), and Util::XmlStartTag::matchLabel().

12.224.3.4 next() `void Util::XmlBase::next ( ) [inline]`

Advance to the next character.

Definition at line 104 of file XmlBase.h.

Referenced by Util::XmlAttribute::match(), Util::XmlEndTag::match(), Util::XmlStartTag::matchAttribute(), and Util::XmlStartTag::matchLabel().

12.224.3.5 string() `const std::string & Util::XmlBase::string ( ) const [inline]`

Return the associated string.

Definition at line 118 of file XmlBase.h.

Referenced by Util::XmlStartTag::finish(), Util::XmlAttribute::match(), Util::XmlEndTag::match(), Util::XmlStartTag::matchLabel(), and setString().

12.224.3.6 cursor() `int Util::XmlBase::cursor ( ) const [inline]`

Return the index of the current character.

Definition at line 124 of file XmlBase.h.

Referenced by Util::XmlAttribute::match(), Util::XmlEndTag::match(), Util::XmlStartTag::matchLabel(), setCursor(), and setString().

12.224.3.7 c() `int Util::XmlBase::c ( ) const [inline]`

Return the current character.

Definition at line 130 of file XmlBase.h.

Referenced by Util::XmlAttribute::match(), Util::XmlEndTag::match(), Util::XmlStartTag::matchAttribute(), and Util::Xml←StartTag::matchLabel().

12.224.3.8 isEnd() `bool Util::XmlBase::isEnd ( ) const [inline]`

Has the cursor reached the end of the string?

Definition at line 136 of file XmlBase.h.

Referenced by Util::XmlAttribute::match(), Util::XmlEndTag::match(), Util::XmlStartTag::matchAttribute(), and Util::Xml←StartTag::matchLabel().

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

- XmlBase.h
- XmlBase.cpp

12.225 Util::XmlEndTag Class Reference

Parser for an XML end tag.

```
#include <XmlEndTag.h>
```

Inheritance diagram for Util::XmlEndTag:

classUtil_1_1XmlEndTag-eps-converted-to.pdf

Public Member Functions

- XmlEndTag ()

  Constructor.
- ∼XmlEndTag ()

  Destructor.
- bool match (const std::string &string, int begin)

  Attempt to match any end tag.
- void match (const std::string expected, const std::string &string, int begin)

  Match a required end tag.
- const std::string label ()

  Label string.

12.225.1 Detailed Description

Parser for an XML end tag.

Definition at line 24 of file XmlEndTag.h.

12.225.2 Constructor & Destructor Documentation

12.225.2.1 XmlEndTag() `Util::XmlEndTag::XmlEndTag ( )`

Constructor.
Definition at line 14 of file XmlEndTag.cpp.

12.225.2.2 ∼XmlEndTag() `Util::XmlEndTag::∼XmlEndTag ( )`

Destructor.
Definition at line 17 of file XmlEndTag.cpp.

12.225.3 Member Function Documentation

12.225.3.1 match() [1/2]

```
bool Util::XmlEndTag::match (
        const std::string & string,
        int begin )
```

Attempt to match any end tag.
Return true if end tag found, false otherwise.

Parameters

string	containing text of XML tag
begin	index of first character

Definition at line 20 of file XmlEndTag.cpp.
References Util::XmlBase::c(), Util::XmlBase::cursor(), Util::XmlBase::isEnd(), Util::XmlBase::next(), Util::XmlBase::setString(), Util::XmlBase::skip(), and Util::XmlBase::string().
Referenced by match().

12.225.3.2 match() [2/2]

```
void Util::XmlEndTag::match (
        const std::string expected,
        const std::string & string,
        int begin )
```

Match a required end tag.
Throw exception is specified end tag does not match.

Parameters

expected	expected label string
string	containing text of XML tag
begin	index of first character

Definition at line 65 of file XmlEndTag.cpp.
References Util::Log::file(), label(), match(), and UTIL_THROW.

12.225.3.3 label() `const std::string Util::XmlEndTag::label ( ) [inline]`

Label string.
Definition at line 64 of file XmlEndTag.h.
Referenced by match().
The documentation for this class was generated from the following files:

- XmlEndTag.h
- XmlEndTag.cpp

12.226 Util::XmlStartTag Class Reference

Parser for an XML start tag.
#include <XmlStartTag.h>
Inheritance diagram for Util::XmlStartTag:

classUtil_1_1XmlStartTag-eps-converted-to.pdf

Public Member Functions

- XmlStartTag ()

  Constructor.
- ∼XmlStartTag ()

  Destructor.
- bool matchLabel (const std::string &string, int begin)

  Match opening bracket and any label.
- void matchLabel (const std::string expected, const std::string &string, int begin)

  Match opening bracket and a specific required label.
- bool matchAttribute (XmlAttribute &attribute)

  Attempt to match an attribute.
- void finish ()

  Check if end bracket was found.
- const std::string label ()

  Label string.
- bool endBracket ()

  True if a closing bracket was found.

12.226.1 Detailed Description

Parser for an XML start tag.
Usage:

```
XmlStartTag tag;
XmlAttribute attribute;
std::string line;
tag.matchLabel(line, 0);
while (matchAttribute(attribute)) {
   // process attribute;
}
tag.finish();
```

Definition at line 36 of file XmlStartTag.h.

12.226.2 Constructor & Destructor Documentation

12.226.2.1 XmlStartTag() `Util::XmlStartTag::XmlStartTag ( )`

Constructor.
Definition at line 14 of file XmlStartTag.cpp.

12.226.2.2 ∼XmlStartTag() `Util::XmlStartTag::∼XmlStartTag ( )`

Destructor.
Definition at line 18 of file XmlStartTag.cpp.

12.226.3 Member Function Documentation

12.226.3.1 matchLabel() [1/2] `bool Util::XmlStartTag::matchLabel (`

```
const std::string & string,
int begin )
```

Match opening bracket and any label.

Parameters

string	containing text of XML tag
begin	index of first character

Returns

true if match, false otherwise

Definition at line 21 of file XmlStartTag.cpp.
References Util::XmlBase::c(), Util::XmlBase::cursor(), Util::XmlBase::isEnd(), Util::XmlBase::next(), Util::XmlBase::setString(), Util::XmlBase::skip(), and Util::XmlBase::string().
Referenced by matchLabel().

12.226.3.2 matchLabel() [2/2] `void Util::XmlStartTag::matchLabel (`

```
const std::string expected,
const std::string & string,
int begin )
```

Match opening bracket and a specific required label.
Throws exception if no match.

Parameters

expected	expected label string
string	containing text of XML tag
begin	index of first character

Definition at line 58 of file XmlStartTag.cpp.
References Util::Log::file(), label(), matchLabel(), and UTIL_THROW.

12.226.3.3 matchAttribute() `bool Util::XmlStartTag::matchAttribute ( XmlAttribute & attribute )`

Attempt to match an attribute.

Parameters

attribute	on return, matched attribute, if any

Returns

true if an attribute is found, false otherwise

Definition at line 71 of file XmlStartTag.cpp.
References Util::XmlBase::c(), Util::XmlBase::isEnd(), Util::XmlAttribute::match(), Util::XmlBase::next(), and Util::XmlBase::skip().

12.226.3.4 finish() `void Util::XmlStartTag::finish ( )`

Check if end bracket was found.
Throws exception if no end bracket was found.
Definition at line 93 of file XmlStartTag.cpp.
References endBracket(), Util::Log::file(), Util::XmlBase::string(), and UTIL_THROW.

12.226.3.5 label() `const std::string Util::XmlStartTag::label ( ) [inline]`

Label string.
Definition at line 90 of file XmlStartTag.h.
Referenced by matchLabel().

12.226.3.6 endBracket() `bool Util::XmlStartTag::endBracket ( ) [inline]`

True if a closing bracket was found.
Definition at line 96 of file XmlStartTag.h.
Referenced by finish().
The documentation for this class was generated from the following files:

- XmlStartTag.h
- XmlStartTag.cpp

12.227 Util::XmlXmlTag Class Reference

Parser for an XML file declaration tag (first line in file).
`#include <XmlXmlTag.h>`
Inheritance diagram for Util::XmlXmlTag:

classUtil_1_1XmlXmlTag-eps-converted-to.pdf

Public Member Functions

- XmlXmlTag ()

    Constructor.
- ∼XmlXmlTag ()

    Destructor.
- bool match (const std::string &string, int begin)

    Attempt to match entire xml tag.

12.227.1 Detailed Description

Parser for an XML file declaration tag (first line in file).
The match function attempts to match an xml file declaration tag, such as: <?xml version="1.0" encoding="UTF-8"?>.
Definition at line 27 of file XmlXmlTag.h.

12.227.2 Constructor & Destructor Documentation

12.227.2.1 XmlXmlTag() `Util::XmlXmlTag::XmlXmlTag ( )`

Constructor.
Definition at line 14 of file XmlXmlTag.cpp.

12.227.2.2 ∼XmlXmlTag() `Util::XmlXmlTag::∼XmlXmlTag ( )`

Destructor.
Definition at line 18 of file XmlXmlTag.cpp.

12.227.3 Member Function Documentation

12.227.3.1 match()

```
bool Util::XmlXmlTag::match (
        const std::string & string,
        int begin )
```

Attempt to match entire xml tag.

Parameters

string	containing text of XML tag
begin	index of first character

Returns

true on match, false otherwise

Definition at line 21 of file XmlXmlTag.cpp.
References Util::XmlAttribute::label().
The documentation for this class was generated from the following files:

- XmlXmlTag.h
- XmlXmlTag.cpp

13 File Documentation

13.1 global.h File Reference

File containing preprocessor macros for error handling.

```
#include <mpi.h>
#include <util/misc/Log.h>
#include "assert.h"
#include "misc/Exception.h"
```

Macros

- #define NDEBUG

  Include access to a Log file.
- #define UTIL_FUNC __PRETTY_FUNCTION__

  Macro for the name of the current function (compiler dependent).
- #define UTIL_THROW(msg)

  Macro for throwing an Exception, reporting function, file and line number.
- #define UTIL_CHECK(condition) if (!(condition)) { UTIL_THROW("Failed assertion: " #condition); }

  Assertion macro suitable for serial or parallel production code.
- #define UTIL_ASSERT(condition) {}

  Assertion macro suitable for debugging serial or parallel code.

13.1.1 Detailed Description

File containing preprocessor macros for error handling.

13.1.2 Macro Definition Documentation

13.1.2.1 NDEBUG `#define NDEBUG`

Include access to a Log file.
If defined, disable all C assert(...) statements.
Definition at line 32 of file global.h.

13.1.2.2 UTIL_FUNC `#define UTIL_FUNC __PRETTY_FUNCTION__`

Macro for the name of the current function (compiler dependent).
Definition at line 42 of file global.h.

13.1.2.3 UTIL_THROW `#define UTIL_THROW( msg )`

Value:

```
{ \
Exception e(UTIL_FUNC, msg, __FILE__, __LINE__); \
MpiThrow(e); }
```

Macro for throwing an Exception, reporting function, file and line number.
Definition at line 51 of file global.h.

13.1.2.4 UTIL_CHECK `#define UTIL_CHECK( condition ) if (!(condition)) { UTIL_THROW("Failed assertion: " #condition); }`

Assertion macro suitable for serial or parallel production code.
Definition at line 68 of file global.h.

13.1.2.5 UTIL_ASSERT `#define UTIL_ASSERT( condition ) {}`

Assertion macro suitable for debugging serial or parallel code.
Definition at line 75 of file global.h.

13.2 MpiSendRecv.h File Reference

```
#include <util/global.h>
#include <util/mpi/MpiTraits.h>
#include <util/containers/DArray.h>
#include <util/containers/DMatrix.h>
```

Namespaces

- Util

    Utility classes for scientific computation.

Functions

- template< typename T >
  void Util::send (MPI::Comm &comm, T &data, int dest, int tag)

    Send a single T value.
- template< typename T >
  void Util::recv (MPI::Comm &comm, T &data, int source, int tag)

    Receive a single T value.
- template< typename T >
  void Util::bcast (MPI::Intracomm &comm, T &data, int root)

    Broadcast a single T value.
- template< typename T >
  void Util::send (MPI::Comm &comm, T ∗array, int count, int dest, int tag)

    Send a C-array of T values.
- template< typename T >
  void Util::recv (MPI::Comm &comm, T ∗array, int count, int source, int tag)

    Receive a C-array of T objects.
- template< typename T >
  void Util::bcast (MPI::Intracomm &comm, T ∗array, int count, int root)

    Broadcast a C-array of T objects.
- template< typename T >
  void Util::send (MPI::Comm &comm, DArray< T > &array, int count, int dest, int tag)

    Send a DArray<T> container.
- template< typename T >
  void Util::recv (MPI::Comm &comm, DArray< T > &array, int count, int source, int tag)

    Receive a DArray<T> container.
- template< typename T >
  void Util::bcast (MPI::Intracomm &comm, DArray< T > &array, int count, int root)

Broadcast a DArray<T> container.

- template<typename T >
  void Util::send (MPI::Comm &comm, DMatrix< T > &matrix, int m, int n, int dest, int tag)

  Send a DMatrix<T> container.
- template<typename T >
  void Util::recv (MPI::Comm &comm, DMatrix< T > &matrix, int m, int n, int source, int tag)

  Receive a DMatrix<T> container.
- template<typename T >
  void Util::bcast (MPI::Intracomm &comm, DMatrix< T > &matrix, int m, int n, int root)

  Broadcast a DMatrix<T> container.
- template<> void Util::send< bool > (MPI::Comm &comm, bool &data, int dest, int tag)

  Explicit specialization of send for bool data.
- template<> void Util::recv< bool > (MPI::Comm &comm, bool &data, int source, int tag)

  Explicit specialization of recv for bool data.
- template<> void Util::bcast< bool > (MPI::Intracomm &comm, bool &data, int root)

  Explicit specialization of bcast for bool data.
- template<> void Util::send< std::string > (MPI::Comm &comm, std::string &data, int dest, int tag)

  Explicit specialization of send for std::string data.
- template<> void Util::recv< std::string > (MPI::Comm &comm, std::string &data, int source, int tag)

  Explicit specialization of recv for std::string data.
- template<> void Util::bcast< std::string > (MPI::Intracomm &comm, std::string &data, int root)

  Explicit specialization of bcast for std::string data.

13.2.1 Detailed Description

This file contains templates for global functions send<T>, recv<T> and bcast<T>. These are wrappers for the MPI send, recv (receive), and bcast (broadcast) functions. Overloaded forms of these functions are provided to transmit single values and arrays. The main difference between the wrappers and the underlying MPI functions is that the wrapper functions do not require an MPI type handle as a parameter. Instead, the MPI type associated with C++ type T (the template parameter) is inferred by the function implementations, by methods that are described below. The most important advantage of this is that it allows the wrapper functions to be used within other templates that take type T as a template parameter. The corresponding MPI methods cannot be used in generic templates because they require MPI type handle parameters that have different values for different date types.

The implementation of the templates send<T>, recv<T>, bcast<T> for single values of type T rely on the existence of an associated explicit specialization of the class template MpiTraits<typename T>. If it exists, the class MpiTraits<T> maps C++ type T onto an associated MPI type. Each specialization MpiTraits<T> has a static member MpiTraits<T>::type that contains an opaque handle for the MPI type associated with C++ type T. Explicit specializations for the most common built-in C++ types are defined in MpiTraits.h.

The send<T>, recv<T>, and bcast<T> templates can also be used to transmit instances of a user defined class T if an appropriate MPI type exists. To make this work, the user must define and commit an associated user-defined MPI data type, and also define an explicit specialization MpiTraits<T> to associate this MPI type with C++ type T. Specialized MPI data types and MpiTraits classes for Util::Vector and Util::Vector are defined in the header and implementation files for these classes. User defined MPI types must be committed before they can be used.

Explicit specializations of send<T>, recv<T> and bcast<T> may also be provided for some types for which the algorithm based on MpiTraits is awkward or unworkable. Explicit specializations are declared in this file for bool and std::string. The implementations of send<T> recv<T>, and bcast<T> for T=bool transmit boolean values as integers. The implementations for T = std::string transmit strings as character arrays. No MpiTraits classes are needed or provided for bool or std::string, because the compiler will always use these explicit specializations, which do not rely on MpiTraits classes, rather than the main function templates. It may also be more convenient for some user-defined classes to provide explicit specializations of these three functions, rather than defining an associated MPI type and MpiTraits specialization. The templates defined here can be used to transmit instances of type T either if: (i) Explicit specializations are defined for these three functions, or (ii) an associated MpiTraits class and MPI data type are defined.

Overloaded forms of send<T>, recv<T>, and bcast<T> are provided to transmit 1D and 2D C arrays of data and DArray<T> and DMatrix<T> containers. These functions send the data in one transmission, as a contiguous buffer, if an MPI type is available, but send each element in a separate transmission if no MpiType exists but an explicit specialization exists for the required scalar form of send, recv, or bcast.