THREAD INJECTION TO MAKE DNS CHANNELFLOW RUN IN PARALLEL

Part 1

Igor Grossman and Graham Thorpe
College of Engineering and Science
Victoria University, Melbourne, Australia

Version 0.1.1
Initial revision
May 2016

THREAD INJECTION TO MAKE DNS CHANNELFLOW RUN IN PARALLEL: Part 1

Igor Grossman and Graham Thorpe
College of Engineering and Science
Victoria University
Melbourne
Australia

ABSTRACT

The flows of Newtonian fluids are accurately described by the Navier-Stokes equation. The equation is solved simultaneously with the mass continuity equation, and it may also be solved concurrently with equations that govern the flow of scalar quantities. Analytical solutions of the equation cannot be obtained except for the simplest of geometries, and generally when the flow is laminar. As a result the equation must be solved numerically. The problem is compounded when the flow is turbulent as such flows are characterized by their displaying a wide range of temporal and spatial scales that must be resolved. This is impractical for many industrial applications; hence engineers and scientists usually resort to empirical models to account for the effects of turbulence on the flow field. However, it is useful to have benchmark solutions against which the empirical models can be compared, and these solutions can be obtained by direct numerical solution of the Navier-Stokes equation. This is computationally demanding and in this work we demonstrate in detail how thread injection can be used to parallelize the numerical solution. The program is a modified form Channelflow which is open source software produced by chanelflow.org. The present work is quite voluminous and for this reason it has been presented in two fascicles, namely Parts 1 and 2.

Readers requiring more details and assistance are most welcome to contact Mr Igor Grossman at igor.grossman@live.vu.edu.au.

Keywords: DNS, turbulence, parallelization, thread injection, computer code

Table of Contents

THREAD INJECTION TO MAKE DNS CHANNELFLOW RUN IN PARALLEL

Part 1

1 INTRODUCTION

Engineers and scientists are often confronted with the need to quantify the details of turbulent flows. Such flows are ubiquitous in nature and in many manufactured artifacts such as chemical reactors, flows around bluff bodies such as buildings and road vehicles, fluid flows in internal combustion engines and so on – the list is endless. A feature of turbulent flows is that they appear to be chaotic, and embedded within in them are eddies that have a range of length and time scales. We attempt to understand these phenomena by drawing on the axioms of continuum mechanics. In particular, we rely on the Navier-Stokes equation that encapsulates the conservation of momentum when it is applied to fluids, and it is usually coupled with the conservation of mass [1]. The Navier-Stokes equations contains a non-linear term that is associated with convective acceleration, and it is this term that accounts for the instabilities that give rise to the many length scales associated with turbulent flows. It is this non-linear term that results in the solution of the Navier-Stokes equation being computing resource intensive.. It is not uncommon, for processing times to take several weeks or months to complete. As a result considerable contemporary research is focused on developing faster and more cost-effective computational fluid dynamics (CFD) simulation tools. Existing strands of research that have this objective are:

- Approximation of the Navier-Stokes equation, reducing the dimensions size and simplification of geometry that allows one to obtain analytical solutions. Some of the simplest models are based on one-equation models for the mixing length.
- Averaged Navier-Stokes where all the spatial and time scales are averaged and the effect of turbulence is typically simulated through a k-ε model. This is methodology is known as Reynolds Averaged Navier-Stokes (RANS) modelling.
- Large eddy simulation (LES) adopts an approach in which all of the large spatial and time scales are solved for, and the remaining small scales are simulated. Models that are assuming classical status include the Smagorinsky eddy-viscosity sub-grid scale model and the Dynamic Smagorinsky model.
- Direct numerical simulation (DNS) is a simplification-free approach and it is therefore most accurate. However, this approach is computing resource intensive.

Historically, improvements to DNS simulations occurred as a result of improvements in computer hardware. This has been augmented by developing software and hardware that allows software to run on multi-core processors and computer clusters. The purpose of this report is to develop a tool and methodology to speed up DNS simulations using streams of code running in parallel in the same computer power range.

This report presents a modification of Channelflow, open source software developed by Gibson [channelflow.org]. The idea is to explore the parallelization of Channelflow by means of thread injection. The additional source code is given in this report, and should the reader require further information they should contact Igor Grossman at igor.grossman@gmail.com.

The report is presented in two parts, mainly for the convenience of being able to upload and download the documents in a timely way using technology that is reasonably globally widespread. Technological developments will no doubt obviate the need for this pafrtitin of the report.

2 BACKGROUND INFORMATION

Direct Numerical Simulation (DNS) is a branch of computational flow dynamics where solution of Navier-Stokes equations achieved in most precision form [1]. In DNS turbulence explicitly resolved, rather than modeled like in Reynolds-averaged Navier-Stokes (RANS) or large-eddy simulation (LES). DNS captures all scales including the small eddies that give rise to the viscous dissipation of energy, and this obviates the need for sub-grid models that are a key component of LES [2]. From an information point of view the big advantage of DNS is its ability to provide detailed knowledge of the flow unaffected by approximations. Therefore DNS can be viewed as a numerical experiment based on the axioms of continuum mechanics. However this advantage of DNS comes at a high computational price. Even on the most powerful computers DNS simulations may take months and even years. The main target of this work is to propose a numerical recipe for speeding up such calculations without loss of accuracy and within the same computer power range.

Computer code can be viewed as list of instructions. The operation system executes these instructions when a program loaded. We can see it as a main thread or main stream. A stream is can be regarded as a carrier of our list of instructions. We can create many such streams. In this terminology we can see an analogy between turbulent flow consisting of number of streams and computer process holding a number of streams. One of the differences is that the CPU stream can exists by itself without holding any instructions to execute. And when instructions are loaded in the stream, they are instantly executed. This abstraction and data model can be a basis for development our attempt to speed up DNS - making it run in the different streams when it required, which can be seen as a parallelization of DNS calculations.

3 RUNTIME MODEL

Even as our approach is quite general, we will use channel flow direct numerical simulator where we will implement our model.

Channelflow [Gibson, channelflow.org] is a numerical simulator of the incompressible Navier-Stokes equations where geometry presented as a channel.
Channelflow is an open source software written in C++..
The solution of Navier Stokes equations in channelflow based on direct numerical simulation (DNS) algorithm with spectral decomposition. Software provides examples to simulate equilibrium, traveling waves and periodic orbits of Navier-Stokes.
Channelflow based on relatively modern software design.
The main goal of our research is to apply multithreading to channelflow with a purpose to improve performance.

First we develop a standalone Thread Pool class. The object of this class is to hold all our streams ready to execute tasks in parallel. Creation of the streams has an overhead, so we require that the pool will be created only once and destroyed only when the simulation is finished. To communicate with a main thread, the thread pool will use a hardware interrupt signal notifying it with instructions. The pool is organized as follows:

- CreateThreads - will create and start number of threads specified by parameter m_nbrThreads.
- Run – will unleash treads and let them run.
- DestroyThreadPool - will stop all threads running and remove object of Thread Pool from the system.
- SetLimits - is method for specifying boundary and boundary conditions.
- GetInstance - return pointer for the object of the ThreadPool, only one object of this class can be created.
- WaitToComplete - is a method to synchronise all threads completion. It ensures there will no return from the ThreadPool until all threads have completed their tasks. After all the tasks are finished all of the threads enter a sleep mode,
- WakeUp - methods let all threads know that new execution task arrived and they need to proceed with calculations.
- When a new task arrives, the thread pool streams are re-activated and they start to execute their tasks in parallel.

As can be seen the thread pool class does not have any knowledge about what instructions to execute, it is just a carrier for them. This makes the object of this class extremely well suited to make an injection and take over sequential calculations by processing them in parallel.

The thread pool will have only two interface methods: set Task and Execute Task-which will supply information to the pool exactly what they need to execute.

The calls diagrams, class lists and source code below are here to help and support the code logic and usage of the proposed platform.

3.1 TARGET PLATFORMS

The project has been developed and built using open source GNU C++ compiler. It constructed in such a way that it should compile under any widely used C++ ANSI compiler. All examples and source code were built and run on an HPC (Edward supercomputer) located at Melbourne University. Make file was generated using Cmake – an open source utility for building and maintaining Linux projects. The project depends on boost , C++ libraries, and fftw libraries.

Notes The following commands must be executed prior to attempting to build the project:

- module load cmake
- module load boost
- module load fftw

3.2 PERFORMANCE

The software executes commands about three orders of magnitude faster than personal computers. It has about six orders of magnitude more RAM that a personal computer. However, its main advantage is that it is based on the GNU/Linux operation system, which renders it compatible with any other flavors of Linux operation system

3.3 EXAMPLES

Examples are located at Channelflow/Examples directory on Edward supercomputer. They contain Makefiles and source code for a number of examples and help one to see how to invoke and call utilities of the proposed changes to Channelflow [channelflow.org].

.

4 Class Index

4.1 Class Hierarchy

This inheritance list is sorted approximately, but not exactly, in alphabetical order:

5 Class Index

5.1 Class List

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

6 File Index

6.1 File List

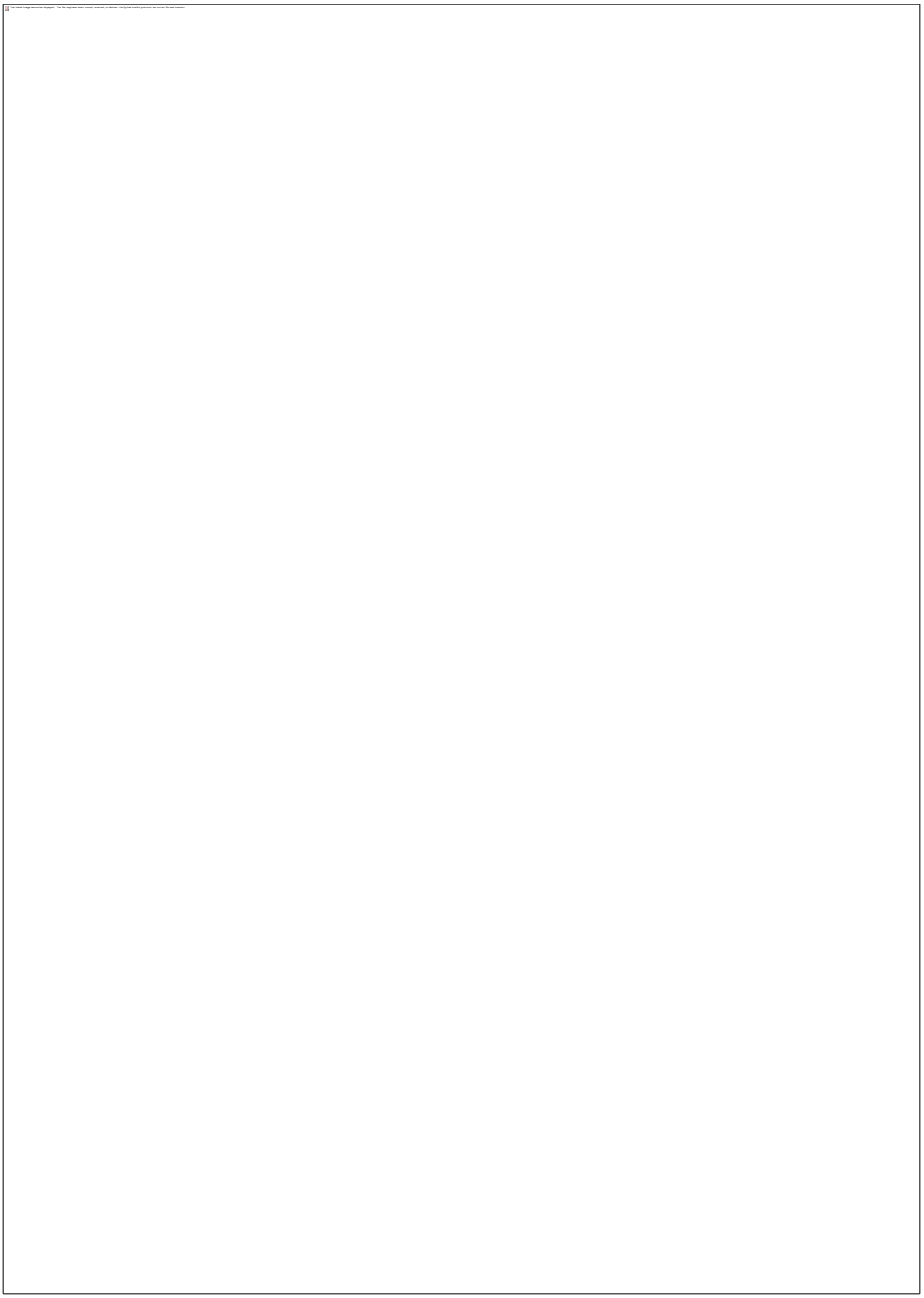
Here is a list of all files with brief descriptions:

7 Class Documentation

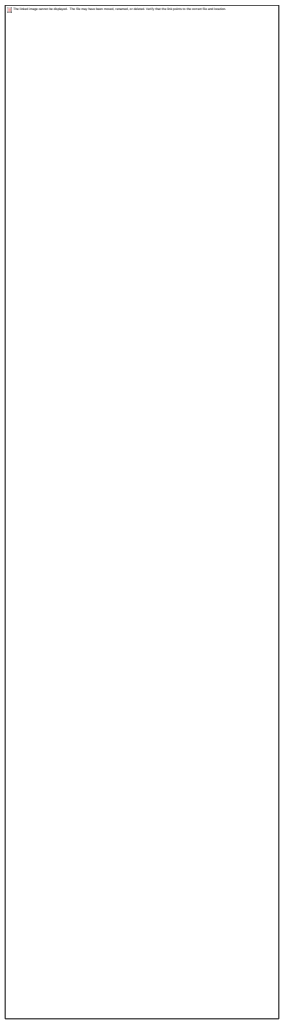
7.1 array< T > Class Template Reference

#include <array.h>

Inheritance diagram for array< T >:



Collaboration diagram for array< T >:



7.1.1 Public Member Functions

- array (int N=0)
- array (int N, const T &t)
- array (const array &a)
- ~array ()
- bool operator== (const array &a)
- bool operator!= (const array &a)
- void resize (int N)
- void fill (const T &t)
- array & operator= (const array &a)
- T & operator[] (int i)
- const T & operator[] (int i) const
- array subvector (int offset, int N) const
- int N () const
- int length () const

- const T * pointer () const
- T * pointer ()
- void save (const std::string &filename) const
- void binaryDump (std::ostream &os) const
- void binaryLoad (std::istream &is)

7.1.2 Private Attributes

- T * data_
- int N_

7.1.2.1 template<class T> class array< T >

7.1.3 Constructor & Destructor Documentation

7.1.3.1 template<class T > array< T >::array (int *N* = 0) [inline]

References array< T >::data_.

```
94                  :
95   data_(new T[N]),
96   N_(N)
97 {
98   assert(data_ != 0);
99   //for (int i=0; i<N_; ++i)
100   //data_[i] = T();
101 }
```

7.1.3.2 template<class T> array< T >::array (int *N*, const T & *t*) [inline]

References array< T >::data_, and array< T >::N_.

```
104                         :
105   data_(new T[N]),
106   N_(N)
107 {
108   assert(data_ != 0);
109   for (int i=0; i<N_; ++i)
110     data_[i] = t;
111 }
```

7.1.3.3 template<class T> array< T >::array (const array< T > & *a*) [inline]

References array< T >::data_, and array< T >::N_.

```
114                             :
115   data_(new T[a.N_]),
116   N_(a.N_)
117 {
118   assert(data_ != 0);
119   T* dptr = data_;
120   T* aptr = a.data_;
121   for (int i=0; i<N_; ++i)
122     *dptr++ = *aptr++;
123 }
```

7.1.3.4 template<class T > array< T >::~array () [inline]

References array< T >::data_.

```
126                 {
127   //cout << "Vector dtor " << long(data_) << endl;
128   delete[] data_;
129   data_=0;
130 }
```

7.1.4 Member Function Documentation

7.1.4.1 template<class T > void array< T >::binaryDump (std::ostream & *os*) const [inline]

References array< T >::data_, array< T >::N_, and write().

```
238                                     {
239   write(os, N_);
240   T* d = data_;
241   T* end = data_ + N_;
242   const int size = sizeof(T);
243   while (d < end)
244     os.write((char*) d++, size);
245 }
```

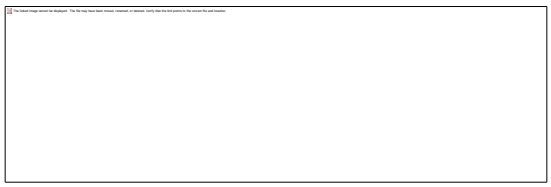
Here is the call graph for this function:

7.1.4.2 template<class T > void array< T >::binaryLoad (std::istream & *is*) [inline]

References array< T >::data_, array< T >::N(), array< T >::N_, and read().

```
248                                 {
249   int N;
250   read(is, N);
251
252   // Change length if necess.
253   if (N != N_) {
254     delete[] data_;
255     data_ = new T[N_ = N];
256   }
257
258   // How can this be made to work with endianness and arbitrary types T?
259   T* d = data_;
260   T* end = data_ + N_;
261   const int size = sizeof(T);
262   while (d < end)
263     is.read((char*) d++, size);
264 }
```

Here is the call graph for this function:



7.1.4.3 template<class T> void array< T >::fill (const T & *t*) [inline]

References array< T >::data_, and array< T >::N_.

```
152                                 {
153   for(int i=0; i<N_; ++i)
154     data_[i] = t;
155 }
```

7.1.4.4 template<class T > int array< T >::length () const [inline]

References array< T >::N_.

```
205 {return N_;}
```

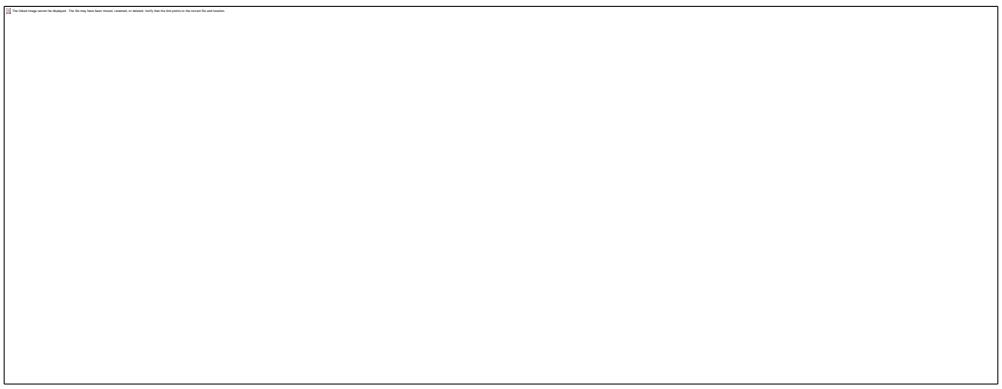
7.1.4.5 template<class T > int array< T >::N () const [inline]

References array< T >::N_.

Referenced by array< T >::binaryLoad(), FieldSeries::interpolate(), polyInterp(), and FieldSeries::push().

```
208 {return N_;}
```

Here is the caller graph for this function:



7.1.4.6 template<class T > bool array< T >::operator!= (const array< T > & *a*) [inline]

```
202 {return !(*this == a);}
```

7.1.4.7 template<class T > array< T > & array< T >::operator= (const array< T > & *a*) [inline]

References array< T >::data_, and array< T >::N_.

```
168                                 {
169   if (data_ != a.data_) {
170     if (N_ != a.N_) {
171       //cout << "delete in operator= on " << long(data_) << endl;
172       delete[] data_;
173       data_ = new T[a.N_];
174       N_ = a.N_;
175       assert(data_ != 0);
176     }
177     T* dptr = data_;
178     T* aptr = a.data_;
179     for (int i=0; i<N_; ++i)
180       *dptr++ = *aptr++;
181   }
```

```
182   return *this;
183 }
```

7.1.4.8 template<class T > bool array< T >::operator== (const array< T > & *a*) [inline]

References array< T >::data_, and array< T >::N_.

```
186                                 {
187   if (this == &a)
188     return true;
189   if (N_ != a.N_)
190     return false;
191
192   T* dptr = data_;
193   T* aptr = a.data_;
194   for (int n=0; n<N_; ++n)
195     if (*dptr++ != *aptr++)
196       return false;
197
198   return true;
199 }
```

7.1.4.9 template<class T > const T & array< T >::operator[] (int *i*) const [inline]

References array< T >::data_, and array< T >::N_.

```
88                                      {
89   assert(i>=0 && i<N_);
90   return data_[i];
91 }
```

7.1.4.10 template<class T > T & array< T >::operator[] (int *i*) [inline]

References array< T >::data_, and array< T >::N_.

```
82                                  {
83   assert(i>=0 && i<N_);
84   return data_[i];
85 }
```

7.1.4.11 template<class T > T * array< T >::pointer () [inline]

References array< T >::data_.

```
212 {return data_;}
```

7.1.4.12 template<class T > const T * array< T >::pointer () const [inline]

References array< T >::data_.

```
215 {return data_;}
```

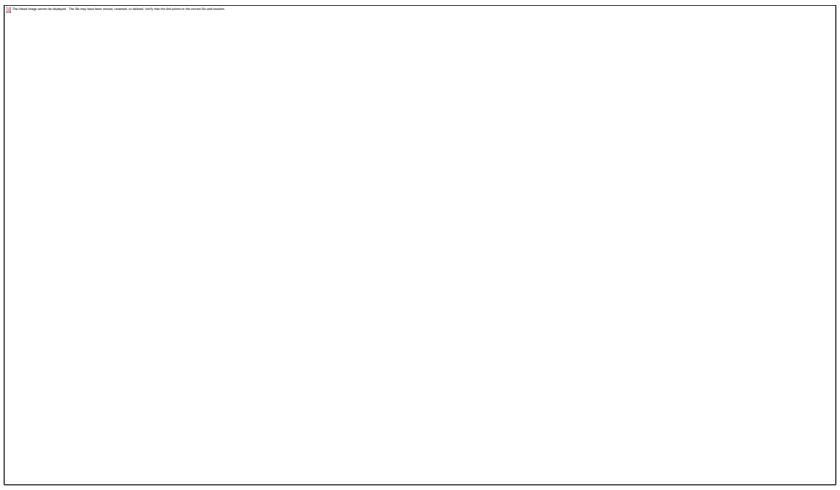
7.1.4.13 template<class T > void array< T >::resize (int *N*) [inline]

References array< T >::data_, and array< T >::N_.

Referenced by BasisFunc::BasisFunc(), CNABstyleDNS::CNABstyleDNS(), MultistepDNS::MultistepDNS(), PPEDNS::PPEDNS(), and RungeKuttaDNS::RungeKuttaDNS().

```
133                     {
134
135   if (N != N_) {
136     // Allocate new space
137     T* newdata_ = new T[N];
138     assert(newdata_ != 0);
139
140     // Copy some/all of old data into new space
141     int M = (N<N_) ? N : N_; // lesser of N, N_
142     for(int i=0; i<M; ++i)
143       newdata_[i] = data_[i];
144
145     // Delete old space, reset pointer to new space, and update size.
146     delete[] data_;
147     data_ = newdata_;
148     N_ = N;
149   }
150 }
```

Here is the caller graph for this function:



7.1.4.14 template<class T > void array< T >::save (const std::string & *filename*) const [inline]

References array< T >::data_, and array< T >::N_.

```
218                                         {
219   std::ofstream os(filebase.c_str());
220   os << std::setprecision(17);
221     os << "% " << N_ << " 1\n";
222   for (int i=0; i<N_; ++i)
223     os << data_[i] << '\n';
224   os.close();
225 }
```

7.1.4.15 template<class T > array< T > array< T >::subvector (int *offset*, int *N*) const [inline]

References array< T >::data_.

```
160 {
161   array<T> subvec(N);
162   for (int i=0; i<N; ++i)
163     subvec[i] = data_[i+offset];
164   return subvec;
165 }
```

7.1.5 Member Data Documentation

7.1.5.1 template<class T> T* array< T >::data_ [private]

Referenced by array< T >::array(), array< T >::binaryDump(), array< T >::binaryLoad(), array< T >::fill(), array< T >::operator=(), array< T >::operator==(), array< T >::operator[](), array< T >::pointer(), array< T >::resize(), array< T >::save(), array< T >::subvector(), and array< T >::~array().

7.1.5.2 template<class T> int array< T >::N_ [private]

Referenced by array< T >::array(), array< T >::binaryDump(), array< T >::binaryLoad(), array< T >::fill(), array< T >::length(), array< T >::N(), array< T >::operator=(), array< T >::operator==(), array< T >::operator[](), array< T >::resize(), and array< T >::save().

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

- _cuda/channelflow-0.9.22/channelflow/array.h

7.1.5.4

7.2 Attributes Class Reference

#include <dns.h>

7.2.1 Public Member Functions

- Attributes (Real _Lx, Real _Lz, int _kxmax, int _kzmax)
- Attributes ()
- void operator= (const Attributes &atr)
- void setAttributes (Real _Lx, Real _Lz, int _kxmax, int _kzmax)

7.2.2 Public Attributes

- Real Lx
- Real Lz
- int kxmax
- int kzmax

7.2.3 Constructor & Destructor Documentation

7.2.3.1 Attributes::Attributes (Real *_Lx*, Real *_Lz*, int *_kxmax*, int *_kzmax*)

References kxmax, kzmax, Lx, and Lz.

```
122 {
123   Lx = _Lx;
124   Lz = _Lz ;
125   kxmax = _kxmax ;
126   kzmax = _kzmax ;
127
128 }
```

7.2.3.2 Attributes::Attributes ()

References kxmax, kzmax, Lx, and Lz.

```
130 {
131   Lx = 0;
132   Lz = 0 ;
133   kxmax = 0 ;
134   kzmax = 0 ;
135 }
```

7.2.4 Member Function Documentation

7.2.4.1 void Attributes::operator= (const Attributes & *atr*)

References kxmax, kzmax, Lx, and Lz.

```
145 {
146   Lx =  attr.Lx;
147   Lz =  attr.Lz ;
148   kxmax =  attr.kxmax ;
149   kzmax =  attr.kzmax ;
150 }
```

7.2.4.2 void Attributes::setAttributes (Real *_Lx*, Real *_Lz*, int *_kxmax*, int *_kzmax*)

References kxmax, kzmax, Lx, and Lz.

```
137 {
138   Lx = _Lx;
139   Lz = _Lz ;
140   kxmax = _kxmax ;
141   kzmax = _kzmax ;
142
143 }
```

7.2.5 Member Data Documentation

7.2.5.1 int Attributes::kxmax

Referenced by Attributes(), operator=(), and setAttributes().

7.2.5.2 int Attributes::kzmax

Referenced by Attributes(), operator=(), and setAttributes().

7.2.5.3 Real Attributes::Lx

Referenced by Attributes(), operator=(), and setAttributes().

7.2.5.4 Real Attributes::Lz

Referenced by Attributes(), operator=(), and setAttributes().

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

- _cuda/channelflow-0.9.22/channelflow/dns.h
- _cuda/channelflow-0.9.22/channelflow/dns.cpp

7.2.5.6

7.3 BandedTridiag Class Reference

#include <bandedtridiag.h>

7.3.1 Public Member Functions

- BandedTridiag ()
- BandedTridiag (int M)
- BandedTridiag (const BandedTridiag &A)
- BandedTridiag (const std::string &filebase)
- ~BandedTridiag ()
- BandedTridiag & operator= (const BandedTridiag &A)
- bool operator== (const BandedTridiag &A) const
- int numrows () const
- Real & band (int j)
- Real & diag (int i)
- Real & updiag (int i)
- Real & lodiag (int i)
- const Real & band (int i) const
- const Real & diag (int i) const
- const Real & updiag (int i) const
- const Real & lodiag (int i) const
- Real & elem (int i, int j)
- const Real & elem (int i, int j) const
- void ULdecomp ()
- void ULsolve (Vector &b) const
- void multiply (const Vector &x, Vector &b) const
- void ULsolveStrided (Vector &b, int offset, int stride) const
- void multiplyStrided (const Vector &x, Vector &b, int offset, int stride) const
- void print () const
- void ULprint () const
- void test () const
- void save (const std::string &filebase) const

7.3.2 Private Member Functions

- int numNonzeros (int M) const
- int numRows (int nnz) const

7.3.3 Private Attributes

- int M_
- int Mbar_
- Real * a_
- Real * d_

- bool UL_

7.3.4 Constructor & Destructor Documentation

7.3.4.1 BandedTridiag::BandedTridiag ()

```
44   : M_(0),
45     Mbar_(-1),
46     a_(0),
47     d_(0),
48     UL_(false)
49 {}
```

7.3.4.2 BandedTridiag::BandedTridiag (int *M*)

References a_.

```
52   : M_(M),
53     Mbar_(M-1),
54     a_(new Real[4*M-2]),
55     d_(a_ + Mbar_),
56     UL_(false)
57 {
58   assert(M>=0);
59   for (int i=0; i<4*M-2; ++i)
60     a_[i] = 0.0;
61 }
```

7.3.4.3 BandedTridiag::BandedTridiag (const BandedTridiag & *A*)

References a_, and M_.

```
64   : M_(A.M_),
65     Mbar_(A.Mbar_),
66     a_(new Real[4*M_ - 2]),
67     d_(a_ + Mbar_),
68     UL_(false)
69 {
70   for (int i=0; i<4*M_-2; ++i)
71     a_[i] = A.a_[i];
72 }
```

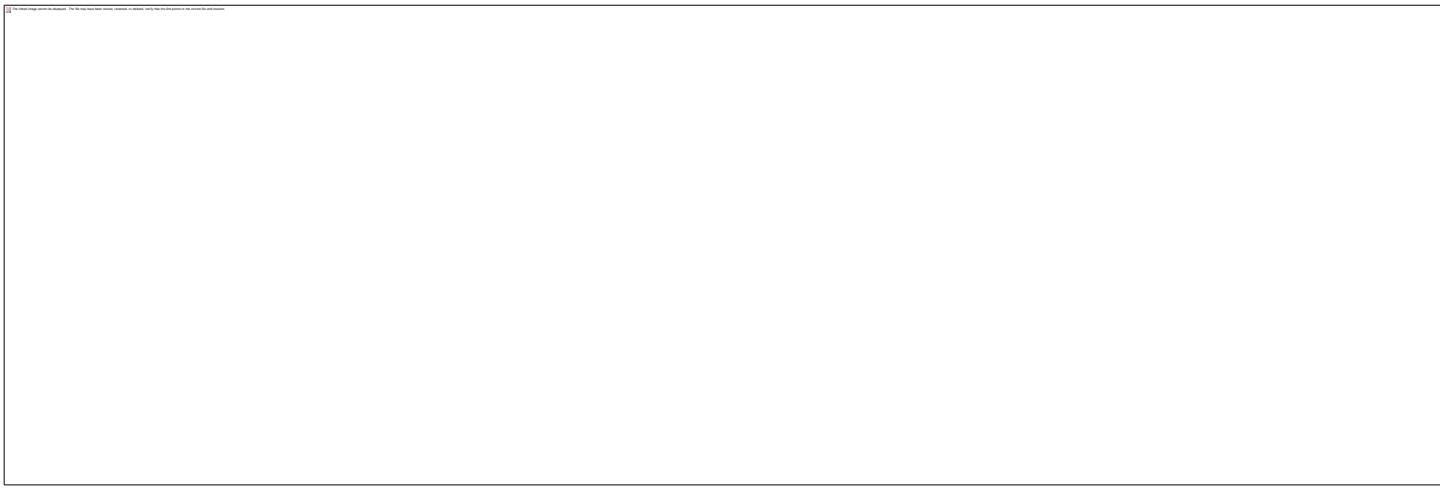
7.3.4.4 BandedTridiag::BandedTridiag (const std::string & *filebase*)

References a_, d_, elem(), M_, Mbar_, and UL_.

```
74                                      :
75   M_(0),
76   Mbar_(-1),
77   a_(0),
78   d_(0),
79   UL_(false)
80 {
81   //cout << "BandedTridiag::BandedTridiag(string& filebase)" << endl;
82   string filename(filebase);
83   filename += string(".asc");
84   ifstream is(filename.c_str());
85
86   // Read in header. Form is "% N U" for a banded tridiag with 8 non-zero elems,
87   // U==0 => UL decomp has not been performed. U==1 indicates it has.
88   char c;
89   is >> c;
90   if (c != '%') {
91     string message("BandedTridiag(filebase): bad header in file ");
92     message += filename;
93     cerr << message << endl;
94     assert(false);
95   }
96   is >> M_;
97   int ul;
98   is >> ul;
99   UL_ = (ul==0) ? false : true;
100
101   //cout << "M, UL == " << M_ << ' ' << UL_ << endl;
102
103   a_ = new Real[4*M_-2];
104   int n; // FOR-SCOPE BUG
105   for (n=0; n<4*M_-2; ++n)
106     a_[n] = 0.0;
107
108   Mbar_ = (M_-1);
109   d_ = a_ + Mbar_;
110
111   int nnz;
112   switch (M_) {
113   case 0:
114     nnz=0; break;
115   case 1:
116     nnz=1; break;
```

```
117   default:
118     nnz = 4*(M_-1);
119   }
120
121   int i,j;
122   Real x;
123   for (n=0; n<nnz; ++n) {
124     is >> i >> j >> x;
125     elem(i,j) = x;
126   }
127 }
```

Here is the call graph for this function:



7.3.4.5 BandedTridiag::~BandedTridiag ()

References a_, and d_.

```
37                          {
38   d_ = 0;
39   delete[] a_;
40 }
```

7.3.5 Member Function Documentation

7.3.5.1 const Real & BandedTridiag::band (int *i*) const [inline]

References a_, M_, and Mbar_.

```
118                                     {
119   assert(j>=0 && j<M_);
120   return a_[Mbar_-j];
121 }
```

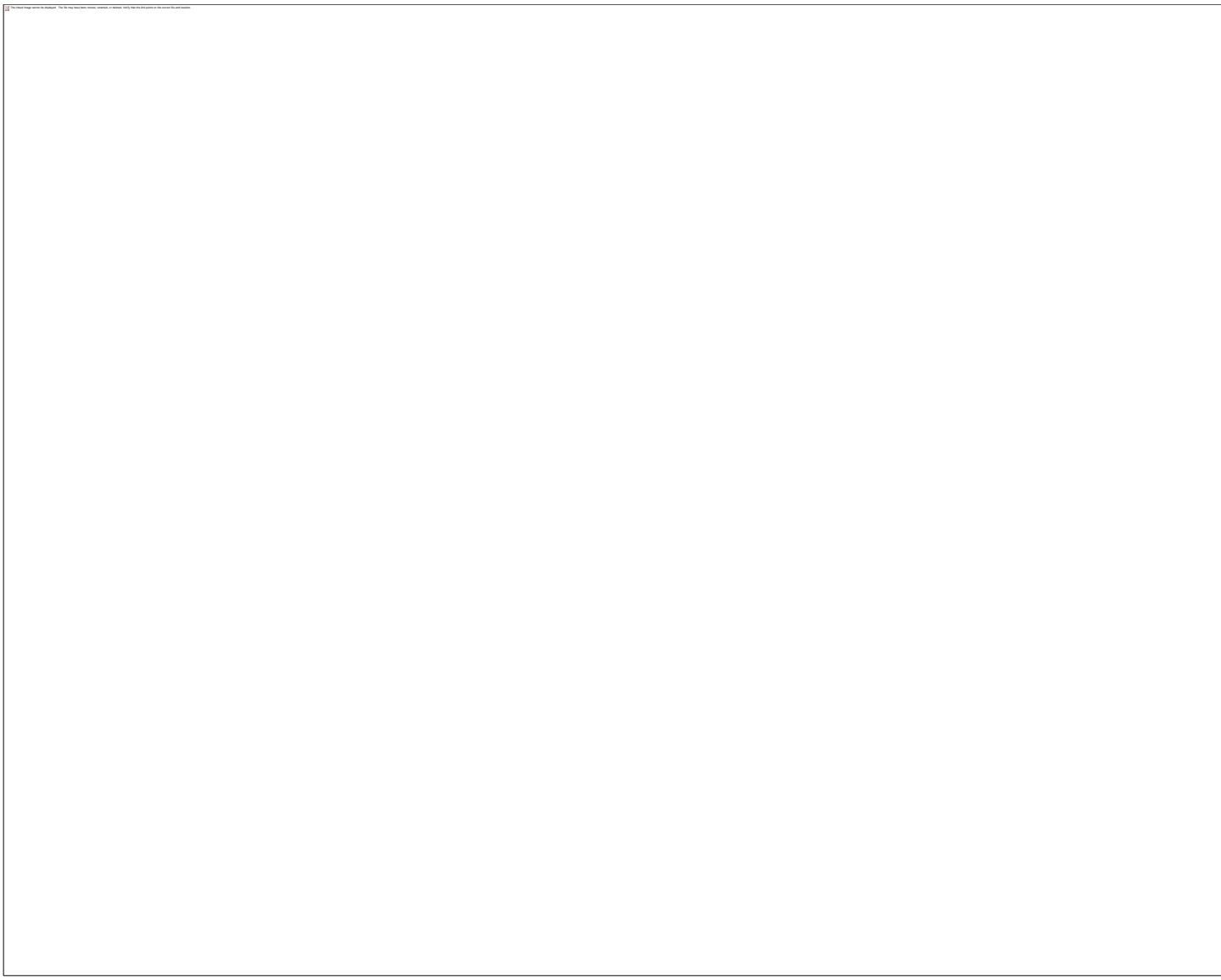
7.3.5.2 Real & BandedTridiag::band (int *j*) [inline]

References a_, M_, and Mbar_.

Referenced by elem(), HelmholtzSolver::HelmholtzSolver(), multiply(), multiplyStrided(), save(), ULdecomp(), ULprint(), ULsolve(), and ULsolveStrided().

```
98                                     {
99   assert(j>=0 && j<M_);
100   return a_[Mbar_-j];
101 }
```

Here is the caller graph for this function:



7.3.5.3 const Real & BandedTridiag::diag (int *i*) const [inline]

References d_, and M_.

```
123                                     {
124   assert(i>=0 && i<M_);
125   return d_[3*i];
126 }
```

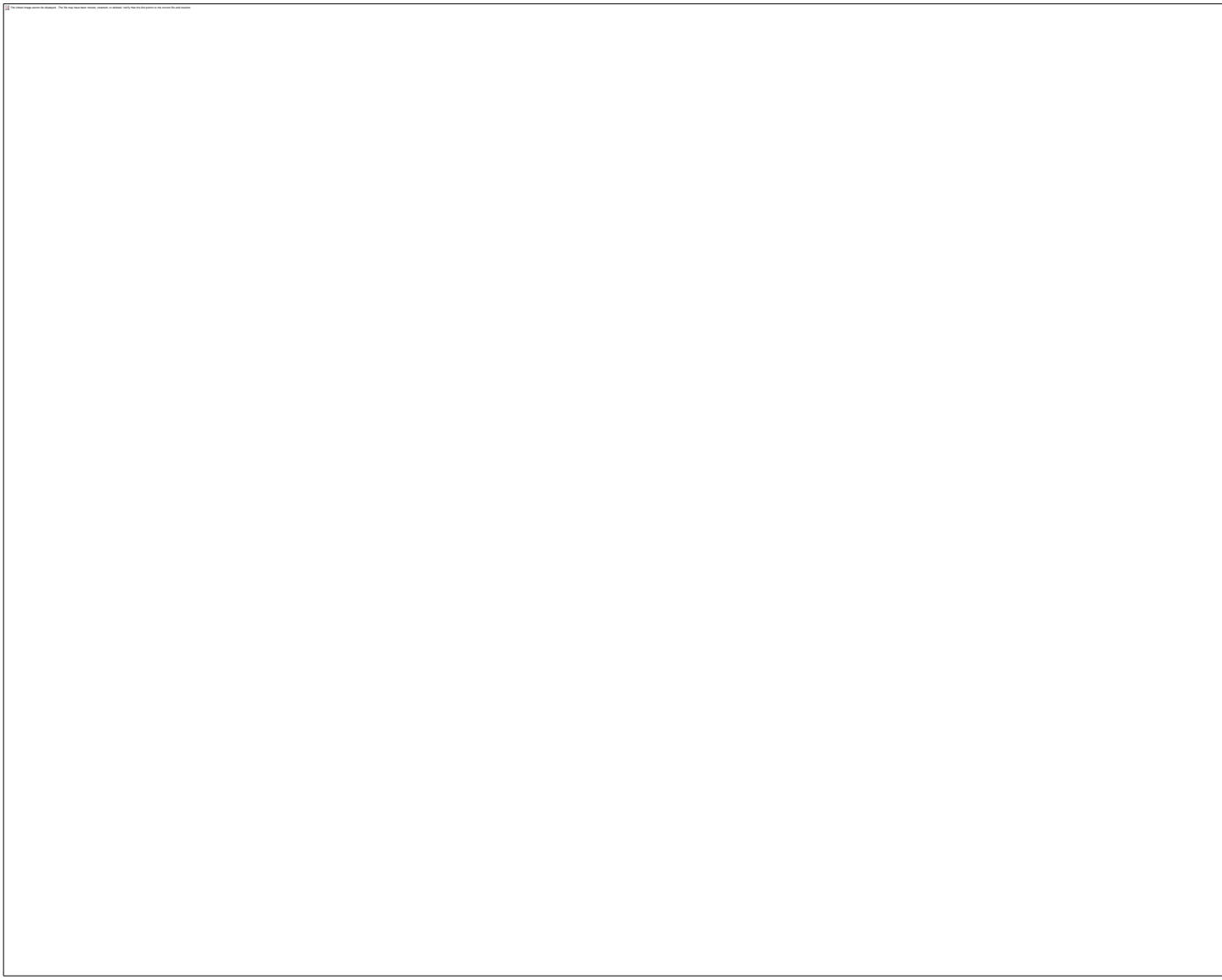
7.3.5.4 Real & BandedTridiag::diag (int *i*) [inline]

References d_, and M_.

Referenced by elem(), HelmholtzSolver::HelmholtzSolver(), multiply(), multiplyStrided(), save(), ULdecomp(), ULprint(), ULsolve(), and ULsolveStrided().

```
103                                 {
104   assert(i>=0 && i<M_);
105   return d_[3*i];
106 }
```

Here is the caller graph for this function:

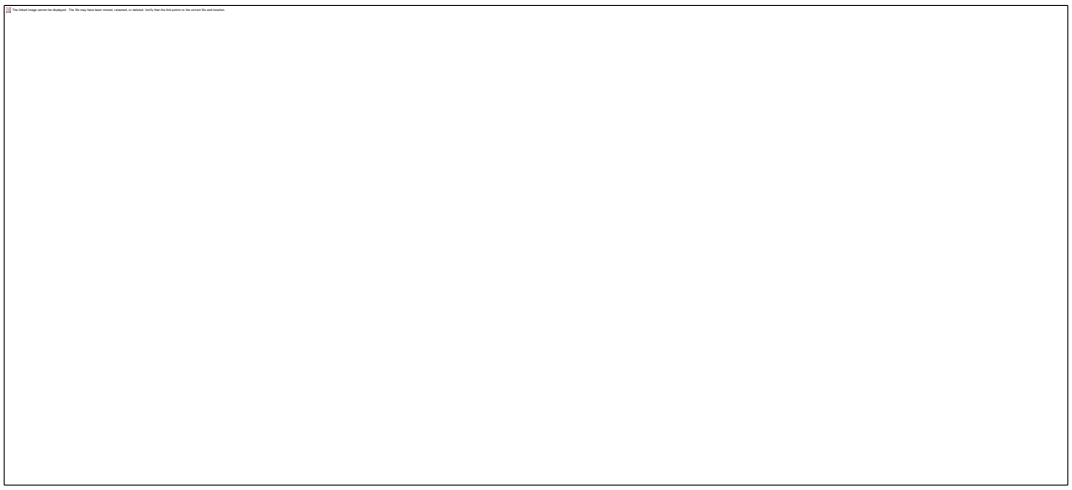


7.3.5.5 const Real & BandedTridiag::elem (int *i*, int *j*) const

References abs(), band(), diag(), lodiag(), M_, and updiag().

```
162                                     {
163   assert(i==0  || (i>=0 && i<M_ && j>=0 && j<M_ && abs(i-j) <= 1));
164   if (i==0)
165     return band(j);
166   else if (i==j)
167     return diag(i);
168   else if (i<j)
169     return updiag(i);
170   else
171     return lodiag(i);
172 }
```

Here is the call graph for this function:



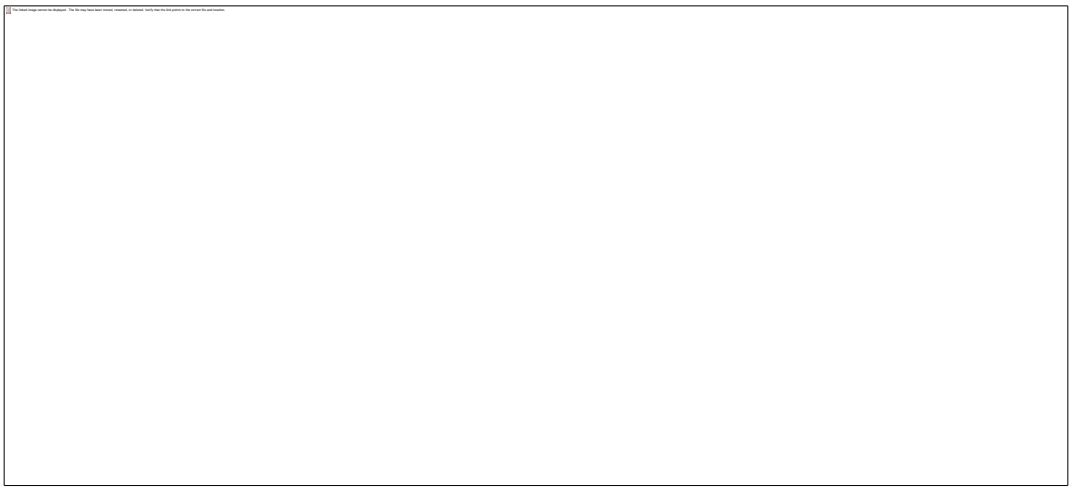
7.3.5.6 Real & BandedTridiag::elem (int *i*, int *j*)

References abs(), band(), diag(), lodiag(), M_, and updiag().

Referenced by BandedTridiag(), and print().

```
174                                     {
175   assert(i==0  || (i>=0 && i<M_ && j>=0 && j<M_ && abs(i-j) <= 1));
176   if (i==0)
177     return band(j);
178   else if (i==j)
179     return diag(i);
180   else if (i<j)
181     return updiag(i);
182   else
183     return lodiag(i);
184 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.3.5.7 const Real & BandedTridiag::lodiag (int *i*) const [inline]

References d_, and M_.

```
133                                         {
134   assert(i>=0 && i<M_);
135   return d_[3*i + 1];
136 }
```

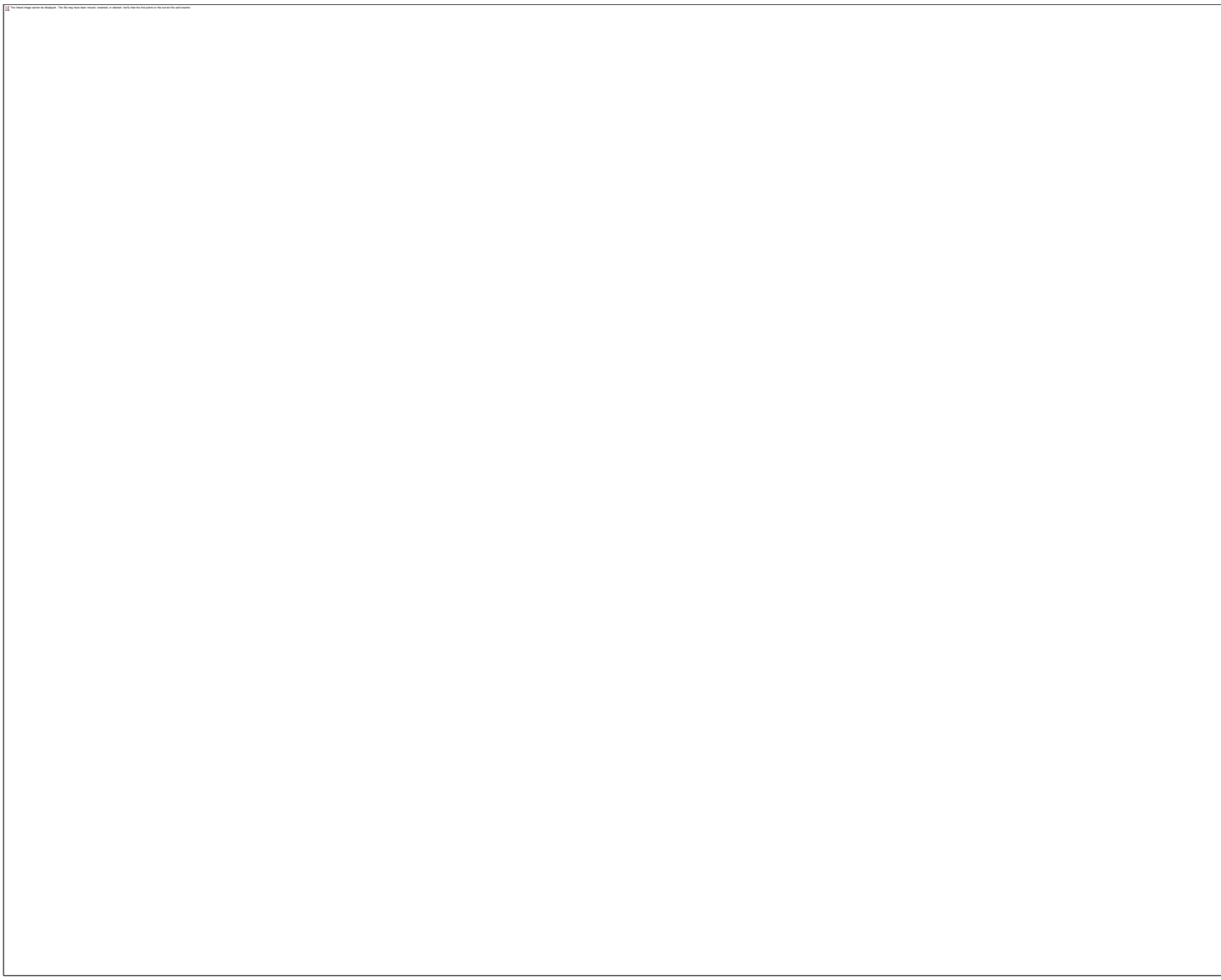
7.3.5.8 Real & BandedTridiag::lodiag (int *i*) [inline]

References d_, and M_.

Referenced by elem(), HelmholtzSolver::HelmholtzSolver(), multiply(), multiplyStrided(), save(), ULdecomp(), ULprint(), ULsolve(), and ULsolveStrided().

```
113                                   {
114   assert(i>=0 && i<M_);
115   return d_[3*i + 1];
116 }
```

Here is the caller graph for this function:



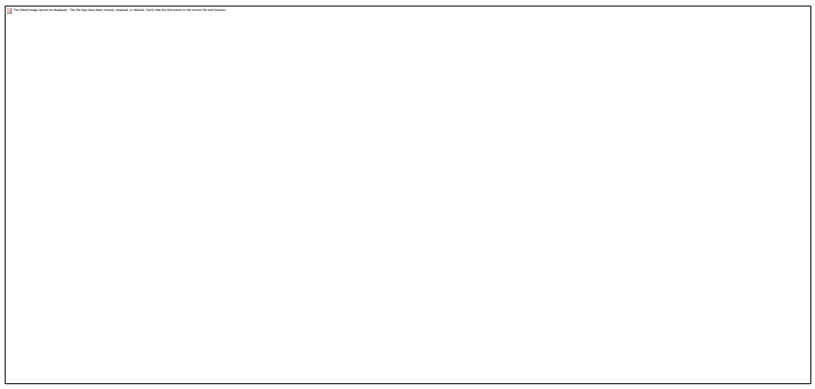
7.3.5.9 void BandedTridiag::multiply (const Vector & *x*, Vector & *b*) const

References band(), diag(), lodiag(), M_, Mbar_, and updiag().

```
302                                                 {
303
304   Real sum=0.0;
305
306   // row 0
307   for (int j=0; j<M_; ++j)
308       sum += band(j)*x[j];
309   b[0] = sum;
310
```

```
311   // rows 1 to (Mbar-1)
312   for (int i=1; i<Mbar_; ++i)
313     b[i] = lodiag(i)*x[i-1] + diag(i)*x[i] + updiag(i) * x[i+1];
314
315   b[Mbar_] = lodiag(Mbar_)*x[Mbar_-1] + diag(Mbar_)*x[Mbar_];
316
317
318 }
```

Here is the call graph for this function:



7.3.5.10 void BandedTridiag::multiplyStrided (const Vector & *x*, Vector & *b*, int *offset*, int *stride*) const

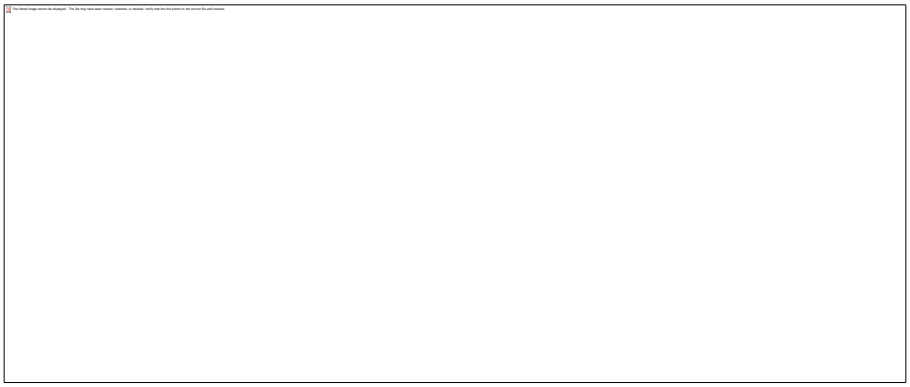
References band(), diag(), lodiag(), M_, Mbar_, and updiag().

Referenced by HelmholtzSolver::residual(), and HelmholtzSolver::solve().

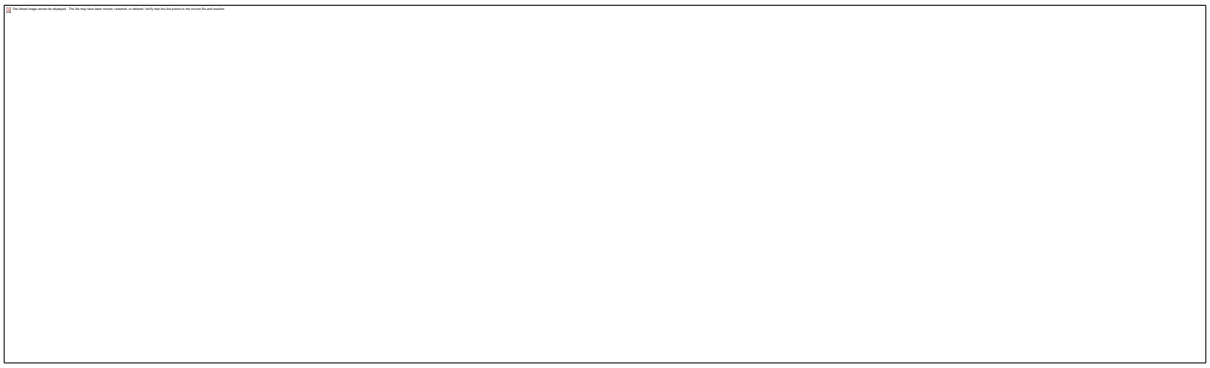
```
320                                                                 {
321
322   assert(offset == 0 || offset == 1);
323   assert(stride == 1 || stride == 2);
324
325   Real sum=0.0;
326
327   // row 0
328   for (int j=0; j<M_; ++j)
329       sum += band(j)*x[offset + stride*j];
330   b[offset] = sum;
331
332   // rows 1 to (Mbar-1)
333   for (int i=1; i<Mbar_; ++i)
334     b[offset + stride*i] = lodiag(i)*x[offset + stride*(i-1)]
335       + diag(i)*x[offset + stride*i] + updiag(i) * x[offset + stride*(i+1)];
336
```

```
337   b[offset + stride*Mbar_] = lodiag(Mbar_)*x[offset + stride*(Mbar_-1)]
338     + diag(Mbar_)*x[offset + stride*Mbar_];
339
340 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.3.5.11 int BandedTridiag::numNonzeros (int *M*) const [private]

7.3.5.12 int BandedTridiag::numRows (int *nnz*) const [private]

7.3.5.13 int BandedTridiag::numrows () const [inline]

7.3.5.14 BandedTridiag & BandedTridiag::operator= (const BandedTridiag & *A*)

References a_, d_, M_, Mbar_, and UL_.

```
130                                                 {
131   if (this != &A) {
132     if (M_ != A.M_) {
133       delete[] a_;
134       M_ = A.M_;
```

```
135     Mbar_ = A.Mbar_;
136     UL_ = A.UL_;
137     a_ = new Real[4*M_-2];
138     d_ = a_ + Mbar_;
139   }
140   for (int i=0; i<4*M_-2; ++i)
141     a_[i] = A.a_[i];
142 }
143 return *this;
144 }
```

7.3.5.15 bool BandedTridiag::operator== (const BandedTridiag & *A*) const

References a_, EPSILON, M_, Mbar_, REAL_DIGITS, and UL_.

```
146                                          {
147 if (M_ != A.M_ || Mbar_ != A.Mbar_ || UL_ != A.UL_)
148   return false;
149
150 for (int n=0; n<4*M_-2; ++n)
151   if (fabs(a_[n] - A.a_[n]) > EPSILON) {
152     cout << "BandedTridiag::operator== failed on a[" << n << "]\n";
153     cout << setprecision(REAL_DIGITS);
154     cout << a_[n] << ' ' << A.a_[n] << endl;
155     return false;
156   }
157
158 return true;
159 }
```

7.3.5.16 void BandedTridiag::print () const

References abs(), elem(), and M_.

```
186                          {
187 cout << "[\n";
188 Real eij=0.0;
189 for (int i=0; i<M_; ++i) {
190   for (int j=0; j<M_; ++j) {
191     if (i==0  || (abs(i-j) <= 1))
192       eij = elem(i,j);
193     else
194       eij = 0.0;
195     cout << eij << ' ';
196   }
197   cout << ";\n";
```

```
198   }
199   cout << "]\n";
200 }
```

Here is the call graph for this function:

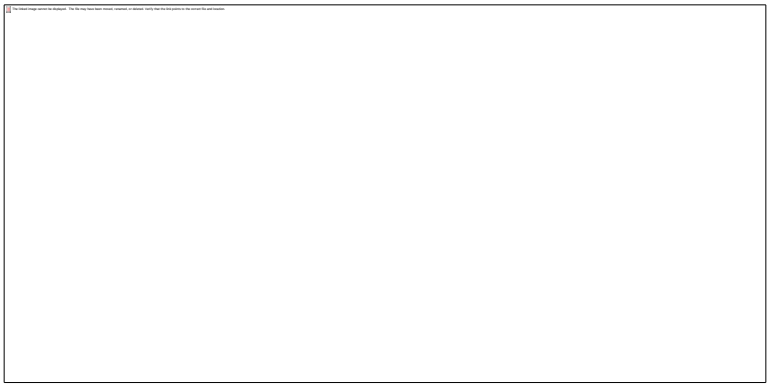
7.3.5.17 void BandedTridiag::save (const std::string & *filebase*) const

References band(), diag(), lodiag(), M_, REAL_DIGITS, UL_, and updiag().

```
349                                                 {
350   string filename(filebase);
351   filename += string(".asc");
352   ofstream os(filename.c_str());
353   os << setprecision(REAL_DIGITS);
354   os << "% " << M_ << ' ' << (UL_ ? 1 : 0) << endl;
355
356   // Print the row-0 band elems
357   for (int j=0; j<M_; ++j)
358     os << "0 " << j << ' ' << band(j) << '\n';
359
360   // Print the lodiag,diag,updiag triplets in rows 1 through M_-2.
361   for (int i=1; i<M_-1; ++i) {
362     os << i << ' ' << (i-1) << ' ' << lodiag(i) << '\n';
363     os << i << ' ' <<  i    << ' ' << diag(i) << '\n';
364     os << i << ' ' << (i+1) << ' ' << updiag(i) << '\n';
365   }
366
367   // Print the lodiag,diag pair in the last row. 2 elems
368   if (M_ > 1) {
369     int i = M_-1;
370     os <<  i << ' ' << (i-1) << ' ' << lodiag(i) << '\n';
```

```
371     os <<  i << ' ' <<   i   << ' ' << diag(i) << '\n';
372   }
373   return;
374 }
```

Here is the call graph for this function:



7.3.5.18 void BandedTridiag::test () const

7.3.5.19 void BandedTridiag::ULdecomp ()

References band(), diag(), lodiag(), M_, UL_, and updiag().
Referenced by HelmholtzSolver::HelmholtzSolver().

```
239                         {
240   int Mb = M_-1;
241   Real w;
242   Real Akk;
243 
244   for (int k=Mb; k>1; --k) {
245     Akk = diag(k);
246     assert(Akk != 0.0);
247     w = lodiag(k); // elem(k,k-1);
248     diag(k-1) -= w*(updiag(k-1) /= Akk); // elem(k-1,k) /= Akk);
249     band(k-1) -= w*(band(k) /= Akk);
250   }
251   band(0) -= lodiag(1)*(band(1) /= diag(1));
252   UL_ = true;
253 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

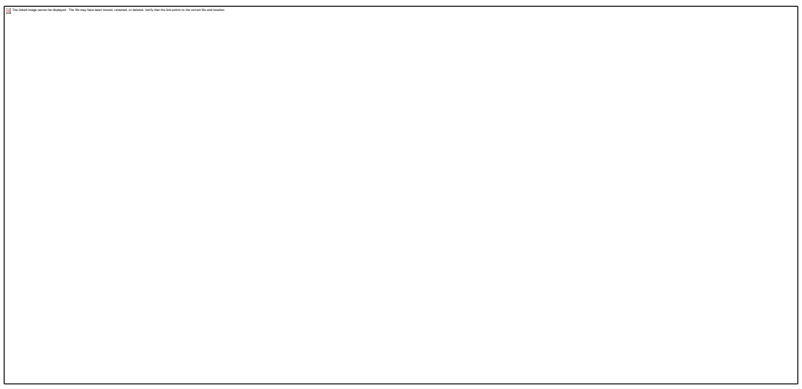
7.3.5.20 void BandedTridiag::ULprint () const

References band(), diag(), lodiag(), M_, and updiag().

```
201                                {
202   Real one = 1.0;
203   Real zero = 0.0;
204   Real eij=0.0;
205   cout << "U = [\n";
206   int i; // MSVC++  FOR-SCOPE BUG
207   for (i=0; i<M_; ++i) {
208     for (int j=0; j<M_; ++j) {
209       if (i==0)
210         eij = band(j);
211       else if (i==j)
212         eij = one;
213       else if (i==j-1)
214         eij = updiag(i);
215       else
216         eij = zero;
217       cout << eij << ' ';
218     }
219     cout << ";\n";
220   }
221   cout << "]\n";
222
223   cout << "L = [\n";
224   for (i=0; i<M_; ++i) {
225     for (int j=0; j<M_; ++j) {
```

```
226       if (i==j+1)
227         eij = lodiag(i);
228       else if (i==j)
229         eij = diag(i);
230       else
231         eij = zero;
232       cout << eij << ' ';
233     }
234     cout << ";\n";
235   }
236   cout << "]\n";
237 }
```

Here is the call graph for this function:



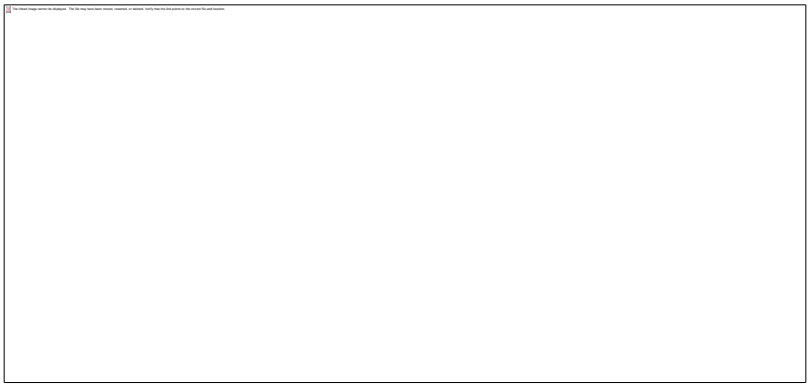
7.3.5.21 void BandedTridiag::ULsolve (Vector & *b*) const

References band(), diag(), lodiag(), M_, UL_, and updiag().

```
282                                 {
283   assert(UL_ == true);
284   int i,j;
285   int Mb=M_-1;
286 
287   // Solve Uy=b by backsubstitution, iterating last row to zeroth.
288 
289   // Last row needs no calculation due to sparsity structure.
290   // b[Nb] = b[Nb];           // row  M-1
291   for (i=Mb-1; i>0; --i)      // rows M-2 through 1
292     b[i] -= updiag(i)*b[i+1];
293   for (j=i+1; j<M_; ++j)       // row 0
294     b[0] -= band(j)*b[j];
295 
296   // Solve Lx=y by forward substitution
```

```
297   b[0] /= diag(0);              // row  0
298   for (i=1; i<M_; ++i)          // rows 1 through M-1
299     (b[i] -= lodiag(i)*b[i-1]) /= diag(i);
300 }
```

Here is the call graph for this function:



7.3.5.22 void BandedTridiag::ULsolveStrided (Vector & *b*, int *offset*, int *stride*) const

References band(), diag(), lodiag(), M_, UL_, and updiag().

Referenced by HelmholtzSolver::solve().

```
256                                                    {
257   assert(UL_ == true);
258   assert(offset == 0 || offset == 1);
259   assert(stride == 1 || stride == 2);
260
261   int i,j;
262   int Mb=M_-1;
263
264   // Solve Uy=b by backsubstitution, iterating last row to zeroth.
265
266   // Last row needs no calculation due to sparsity structure.
267   // b[Nb] = b[Nb];           // row  M-1
268   for (i=Mb-1; i>0; --i)      // rows M-2 through 1
269     b[offset + i*stride] -= updiag(i)*b[offset + stride*(i+1)];
270   for (j=i+1; j<M_; ++j)      // row 0
271     b[offset] -= band(j)*b[offset + stride*j];
272
273   // Solve Lx=y by forward substitution
274   b[offset] /= diag(0);              // row  0
275   for (i=1; i<M_; ++i) {
276     assert(diag(i) != 0.0);  // rows 1 through M-1
277     (b[offset + stride*i] -= lodiag(i)*b[offset + stride*(i-1)]) /= diag(i);
```

```
278   }
279 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.3.5.23 const Real & BandedTridiag::updiag (int *i*) const [inline]

References d_, and M_.

```
128                                         {
129   assert(i>=0 && i<M_);
130   return d_[3*i - 1];
131 }
```

7.3.5.24 Real & BandedTridiag::updiag (int *i*) [inline]

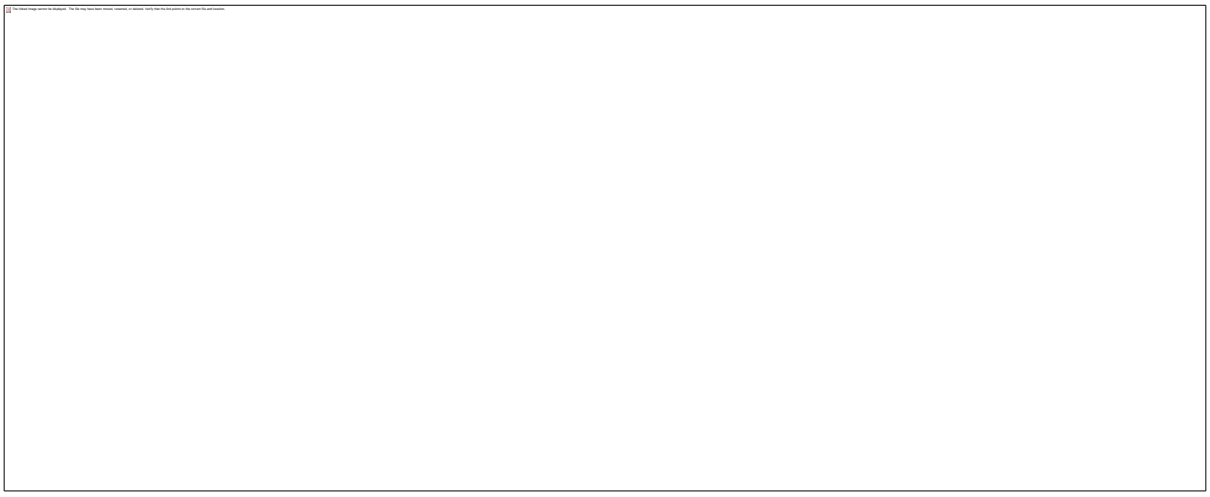
References d_, and M_.

Referenced by elem(), HelmholtzSolver::HelmholtzSolver(), multiply(), multiplyStrided(), save(), ULdecomp(), ULprint(), ULsolve(), and ULsolveStrided().

```
108                                   {
109   assert(i>=0 && i<M_);
```

```
110   return d_[3*i - 1];
111 }
```

Here is the caller graph for this function:



7.3.6 Member Data Documentation

7.3.6.1 Real* BandedTridiag::a_ [private]

Referenced by band(), BandedTridiag(), operator=(), operator==(), and ~BandedTridiag().

7.3.6.2 Real* BandedTridiag::d_ [private]

Referenced by BandedTridiag(), diag(), lodiag(), operator=(), updiag(), and ~BandedTridiag().

7.3.6.3 int BandedTridiag::M_ [private]

Referenced by band(), BandedTridiag(), diag(), elem(), lodiag(), multiply(), multiplyStrided(), operator=(), operator==(), print(), save(), ULdecomp(), ULprint(), ULsolve(), ULsolveStrided(), and updiag().

7.3.6.4 int BandedTridiag::Mbar_ [private]

Referenced by band(), BandedTridiag(), multiply(), multiplyStrided(), operator=(), and operator==().

7.3.6.5 bool BandedTridiag::UL_ [private]

Referenced by BandedTridiag(), operator=(), operator==(), save(), ULdecomp(), ULsolve(), and ULsolveStrided().

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

- _cuda/channelflow-0.9.22/channelflow/bandedtridiag.h
- _cuda/channelflow-0.9.22/channelflow/bandedtridiag.cpp

7.3.6.7

7.4 BaseTask Class Reference

#include <dns.h>

Inheritance diagram for BaseTask:

7.4.1 Public Member Functions

- BaseTask ()
- virtual void Run (int tn, int nThreads)

7.4.2 Constructor & Destructor Documentation

7.4.2.1 BaseTask::BaseTask ()

```
173 {
174
175 }
```

7.4.3 Member Function Documentation

7.4.3.1 void BaseTask::Run (int *tn*, int *nThreads*) [virtual]

Reimplemented in skewsymmetricNL_task, skewsymmetricNL_task2, skewsymmetricNL_task3, and skewsymmetricNL_task4.

Referenced by ThreadPool::Run().

```
178 {
179  //cout << "Igor test Based task Run  running " << endl;
180 }
```

Here is the caller graph for this function:

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

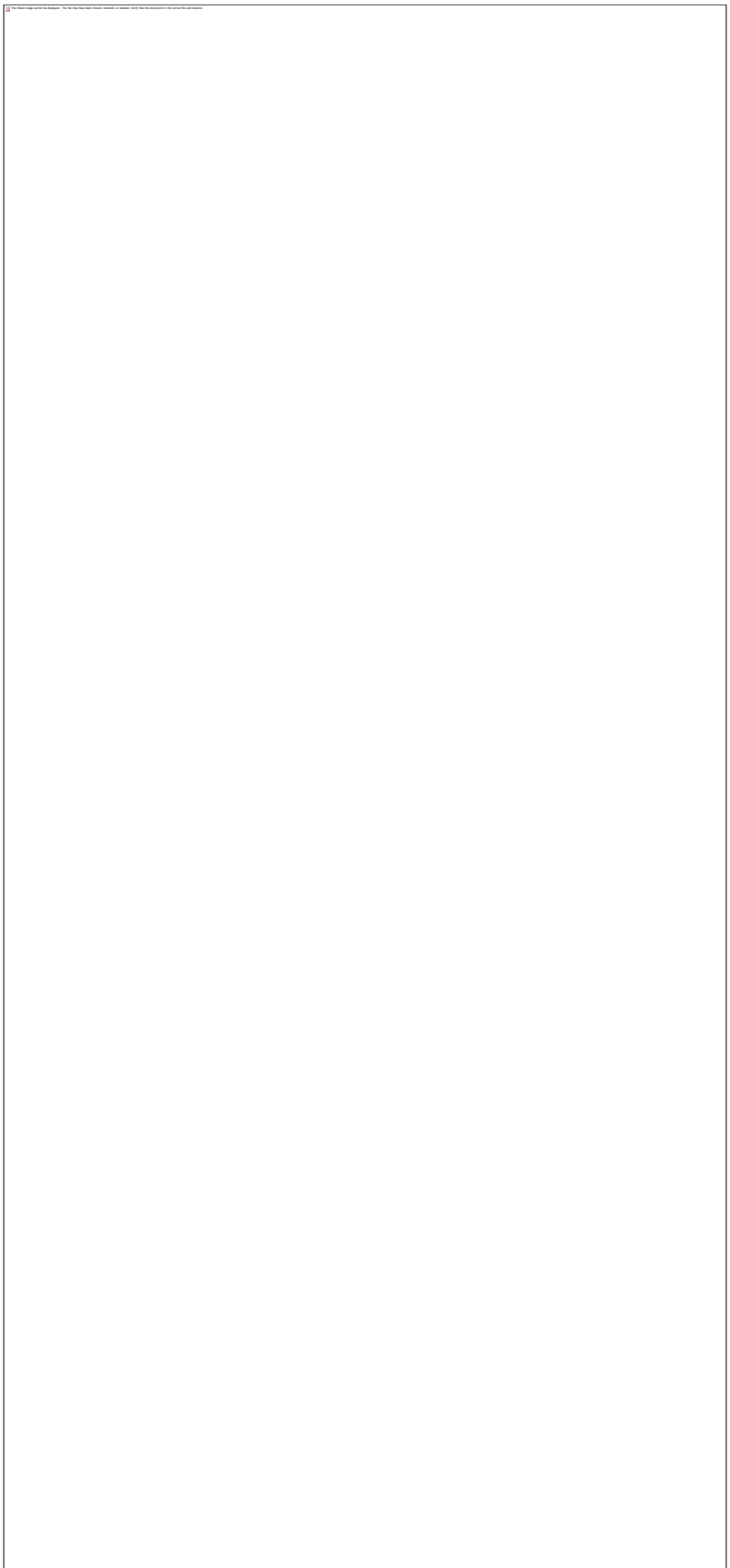
- _cuda/channelflow-0.9.22/channelflow/dns.h
- _cuda/channelflow-0.9.22/channelflow/dns.cpp

7.4.3.3

7.5 BasisFunc Class Reference

#include <basisfunc.h>

Collaboration diagram for BasisFunc:


The inkwell image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.5.1 Public Member Functions

- BasisFunc ()
- BasisFunc (const std::string &filebase)
- BasisFunc (int Nd, int Ny, int kx, int kz, Real Lx, Real Lz, Real a, Real b, fieldstate s=Spectral)
- BasisFunc (int Ny, int kx, int kz, Real Lx, Real Lz, Real a, Real b, fieldstate s=Spectral)
- BasisFunc (int Ny, const BasisFunc &f)
- BasisFunc (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, const ComplexChebyCoeff &w, Real lambda, int kx, int kz, Real Lx, Real Lz)
- void save (const std::string &filebase, fieldstate s=Physical) const
- void binaryDump (std::ostream &os) const
- void binaryLoad (std::istream &is)
- void randomize (bool adiri, bool bdiri, Real decay=0.7)
- void interpolate (const BasisFunc &f)
- void reflect (const BasisFunc &f)
- int Nd () const
- int Ny () const
- int kx () const
- int kz () const
- Real Lx () const
- Real Lz () const
- Real a () const
- Real b () const
- Real rmsval () const
- Real lambda () const
- fieldstate state () const
- void resize (int Ny)
- void setBounds (Real Lx, Real Lz, Real a, Real b)
- void setkxkz (int kx, int kz)
- void setState (fieldstate s)
- void setRmsval (Real lamda)
- void setToZero ()
- void conjugate ()
- void fill (const BasisFunc &f)
- void chebyfft ()
- void ichebyfft ()
- void makeSpectral ()
- void makePhysical ()
- void makeState (fieldstate s)
- void chebyfft (const ChebyTransform &t)
- void ichebyfft (const ChebyTransform &t)
- void makeSpectral (const ChebyTransform &t)

- void makePhysical (const ChebyTransform &t)
- void makeState (fieldstate s, const ChebyTransform &t)
- const ComplexChebyCoeff & u () const
- const ComplexChebyCoeff & v () const
- const ComplexChebyCoeff & w () const
- const ComplexChebyCoeff & component (int i) const
- const ComplexChebyCoeff & operator[] (int i) const
- ComplexChebyCoeff & u ()
- ComplexChebyCoeff & v ()
- ComplexChebyCoeff & w ()
- ComplexChebyCoeff & component (int i)
- ComplexChebyCoeff & operator[] (int i)
- Real bcNorm (bool a_dirichlet=true, bool b_dirichlet=true) const
- bool geometryCongruent (const BasisFunc &f) const
- bool congruent (const BasisFunc &f) const
- BasisFunc & operator*= (const BasisFunc &g)
- BasisFunc & operator+= (const BasisFunc &g)
- BasisFunc & operator-= (const BasisFunc &g)
- BasisFunc & operator*= (Real c)
- BasisFunc & operator*= (Complex c)

7.5.2 Protected Attributes

- int Nd_
- int Ny_
- int kx_
- int kz_
- Real Lx_
- Real Lz_
- Real a_
- Real b_
- fieldstate state_
- Real lambda_
- array< ComplexChebyCoeff > u_

7.5.3 Constructor & Destructor Documentation

7.5.3.1 BasisFunc::BasisFunc ()

```
36  :
37  Nd_(0),
38  Ny_(0),
39  kx_(0),
40  kz_(0),
```

```
41   Lx_(0),
42   Lz_(0),
43   a_(0),
44   b_(0),
45   state_(Spectral),
46   lambda_(0),
47   u_(0)
48 {}
```

7.5.3.2 BasisFunc::BasisFunc (const std::string & *filebase*)

References a_, b_, kx_, kz_, lambda_, Lx_, Lz_, Nd_, Ny_, array< T >::resize(), state_, and u_.

```
104 :
105   Nd_(0),
106   Ny_(0),
107   kx_(0),
108   kz_(0),
109   Lx_(0),
110   Lz_(0),
111   a_(0),
112   b_(0),
113   state_(Spectral),
114   lambda_(0),
115   u_(0)
116 {
117   string filename = filebase + string(".bf");
118   ifstream is(filename.c_str());
119   if (!is.good()) {
120     cerr << "BasisFunc::BasisFunc(filebase) : can't open file " << filename << '\n';
121     abort();
122   }
123   char c;
124   is >> c;
125   if (c != '%') {
126     cerr << "BasisFunc::BasisFunc(filebase): bad header in file " << filename << endl;
127     assert(false);
128   }
129   is >> Nd_
130      >> Ny_
131      >> kx_
132      >> kz_
133      >> Lx_
134      >> Lz_
```

```
135     >> a_
136     >> b_
137     >> state_
138     >> lambda_;
139   u_.resize(Nd_);
140   for (int n=0; n<Nd_; ++n) {
141     u_[n].resize(Ny_);
142     u_[n].setBounds(a_,b_);
143     u_[n].setState(state_);
144   }
145
146   Real r;
147   Real i;
148   for (int ny=0; ny<Ny_; ++ny)
149     for (int n=0; n<Nd_; ++n) {
150       is >> r >> i;
151       u_[n].set(ny,Complex(r,i));
152     }
153 }
```

Here is the call graph for this function:

7.5.3.3 BasisFunc::BasisFunc (int *Nd*, int *Ny*, int *kx*, int *kz*, Real *Lx*, Real *Lz*, Real *a*, Real *b*, fieldstate *s* = Spectral)

References a_, b_, Nd_, Ny_, state_, and u_.

```
67   :
68   Nd_(Nd),
69   Ny_(Ny),
70   kx_(kx),
71   kz_(kz),
72   Lx_(Lx),
73   Lz_(Lz),
74   a_(a),
75   b_(b),
76   state_(s),
77   lambda_(0.0),
78   u_(Nd)
79 {
80   for (int n=0; n<Nd_; ++n)
81     u_[n] = ComplexChebyCoeff(Ny_,a_,b_,state_);
```

```
82 }
```

7.5.3.4 BasisFunc::BasisFunc (int *Ny*, int *kx*, int *kz*, Real *Lx*, Real *Lz*, Real *a*, Real *b*, fieldstate *s* = Spectral)

References a_, b_, Nd_, Ny_, state_, and u_.

```
86   :
87   Nd_(3),
88   Ny_(Ny),
89   kx_(kx),
90   kz_(kz),
91   Lx_(Lx),
92   Lz_(Lz),
93   a_(a),
94   b_(b),
95   state_(s),
96   lambda_(0.0),
97   u_(3)
98 {
99   for (int n=0; n<Nd_; ++n)
100    u_[n] = ComplexChebyCoeff(Ny_,a_,b_,state_);
101 }
```

7.5.3.5 BasisFunc::BasisFunc (int *Ny*, const BasisFunc & *f*)

```
51   :
52   Nd_(f.Nd_),
53   Ny_(Ny),
54   kx_(f.kx_),
55   kz_(f.kz_),
56   Lx_(f.Lx_),
57   Lz_(f.Lz_),
58   a_(f.a_),
59   b_(f.b_),
60   state_(f.state_),
61   lambda_(f.lambda_),
62   u_(f.Nd())
63 {}
```

7.5.3.6 BasisFunc::BasisFunc (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, const ComplexChebyCoeff & *w*, Real *lambda*, int *kx*, int *kz*, Real *Lx*, Real *Lz*)

References congruent(), Nd_, and u_.

```
181 :
182   Nd_(3),
183   Ny_(u.numModes()),
184   kx_(kx),
185   kz_(kz),
186   Lx_(Lx),
187   Lz_(Lz),
188   a_(u.a()),
189   b_(u.b()),
190   state_(u.state()),
191   lambda_(lam),
192   u_(3)
193 {
194   u_[0] = u;
195   u_[1] = v;
196   u_[2] = w;
197   for (int n=1; n<Nd_; ++n) {
198     assert(u_[0].congruent(u_[n]));
199     ;  // this statement needed for optimized compilation, #define assert(x) ;
200   }
201 }
```

Here is the call graph for this function:

7.5.4 Member Function Documentation

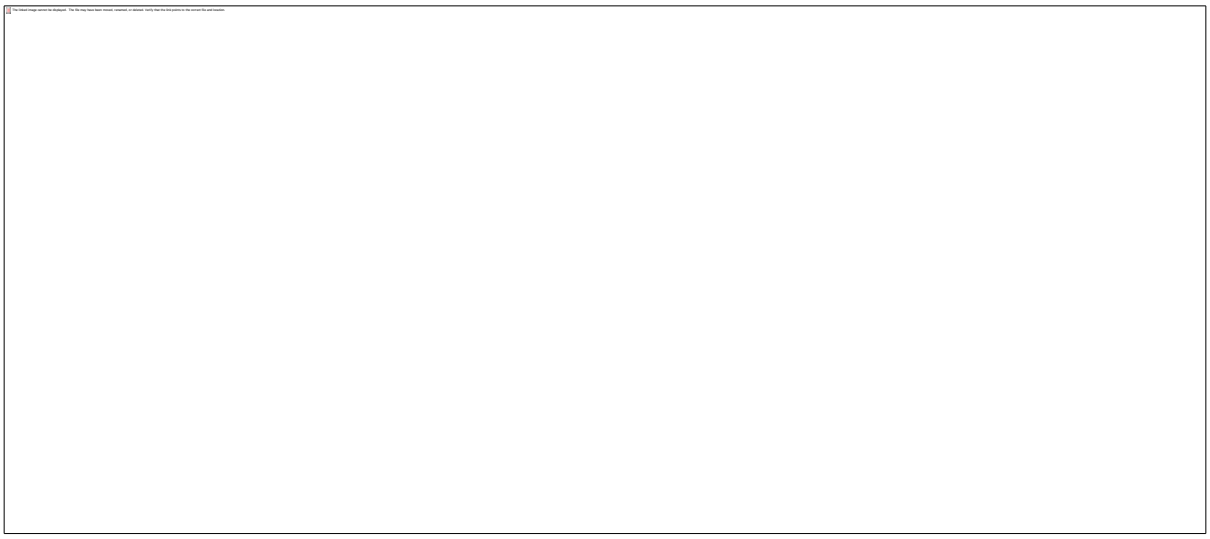
7.5.4.1 Real BasisFunc::a () const [inline]

References a_.

Referenced by FlowField::congruent(), cross(), divergence(), dot(), dotgrad(), gradient(), reflect(), and ydiff().

```
213 {return a_;}
```

Here is the caller graph for this function:



7.5.4.2 Real BasisFunc::b () const [inline]

References b_.

Referenced by FlowField::congruent(), cross(), divergence(), dot(), dotgrad(), gradient(), reflect(), and ydiff().

```
214 {return b_;}
```

Here is the caller graph for this function:

7.5.4.3 Real BasisFunc::bcNorm (bool *a_dirichlet* = true, bool *b_dirichlet* = true) const

References abs2(), Nd_, and u_.

```
383                                                 {
384   Real err=0.0;
385   if (a_dirichlet)
386     for (int n=0; n<Nd_; ++n)
387       err += abs2(u_[n].eval_a());
388   if (b_dirichlet)
389     for (int n=0; n<Nd_; ++n)
390       err += abs2(u_[n].eval_b());
391   return sqrt(err);
392 }
```

Here is the call graph for this function:

7.5.4.4 void BasisFunc::binaryDump (std::ostream & *os*) const

References a_, b_, kx_, kz_, lambda_, Lx_, Lz_, Nd_, Ny_, state_, u_, and write().

```
235                                         {
236   write(os, Nd_);
237   write(os, Ny_);
238   write(os, kx_);
239   write(os, kz_);
240   write(os, Lx_);
241   write(os, Lz_);
242   write(os, a_);
243   write(os, b_);
244   write(os, state_);
245   write(os, lambda_);
246   for (int n=0; n<Nd_; ++n)
```

```
247     u_[n].binaryDump(os);
248 }
```

Here is the call graph for this function:

7.5.4.5 void BasisFunc::binaryLoad (std::istream & *is*)

References a_, b_, kx_, kz_, lambda_, Lx_, Lz_, Nd_, Ny_, read(), state_, and u_.

```
250                                     {
251   if (!is.good()) {
252     cerr << "BasisFunc::binaryLoad(istream) : input error" << endl;
253     abort();
254   }
255   read(is, Ny_);
256   read(is, kx_);
257   read(is, kz_);
258   read(is, Lx_);
259   read(is, Lz_);
260   read(is, a_);
261   read(is, b_);
262   read(is, state_);
263   read(is, lambda_);
264   for (int n=0; n<Nd_; ++n)
265     u_[n].binaryLoad(is);
266 }
```

Here is the call graph for this function:

7.5.4.6 void BasisFunc::chebyfft (const ChebyTransform & *t*)

References chebyfft(), Nd_, Physical, Spectral, state_, and u_.

```
438                                     {
439   assert(state_ == Physical);
440   for (int n=0; n<Nd_; ++n)
441     u_[n].chebyfft(t);
442   state_ = Spectral;
443 }
```

Here is the call graph for this function:

7.5.4.7 void BasisFunc::chebyfft ()

References Nd_, Ny_, Physical, Spectral, state_, and u_.

Referenced by chebyfft().

```
404                       {
405   assert(state_ == Physical);
406   ChebyTransform t(Ny_);
407   for (int n=0; n<Nd_; ++n)
408     u_[n].chebyfft(t);
409   state_ = Spectral;
410 }
```

Here is the caller graph for this function:

7.5.4.8 ComplexChebyCoeff & BasisFunc::component (int *i*)

References Nd_, and u_.

```
479                                 {
480   assert(i>=0 && i<Nd_);
481   return u_[i];
482 }
```

7.5.4.9 const ComplexChebyCoeff & BasisFunc::component (int *i*) const

References Nd_, and u_.

```
475                                       {
476   assert(i>=0 && i<Nd_);
477   return u_[i];
478 }
```

7.5.4.10 bool BasisFunc::congruent (const BasisFunc & *f*) const

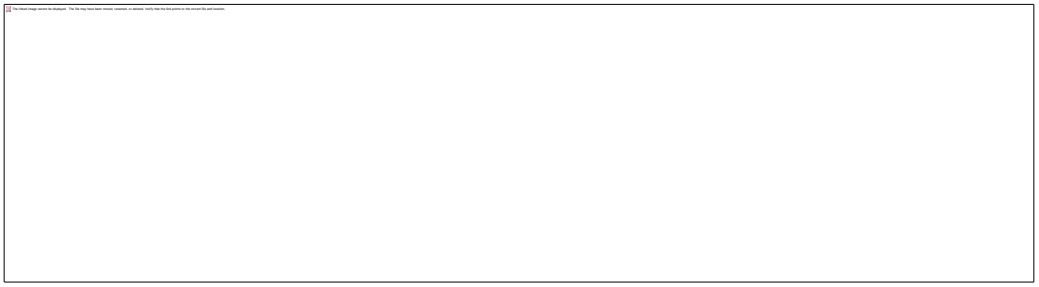
References geometryCongruent(), kx_, kz_, and state_.

Referenced by BasisFunc(), divDist2(), and operator==().

```
399                                          {
400   return (geometryCongruent(Phi) && kx_==Phi.kx_ && kz_==Phi.kz_
401           && state_ == Phi.state_);
402 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



7.5.4.11 void BasisFunc::conjugate ()

References kx_, kz_, Nd_, and u_.

```
376                          {
377   for (int n=0; n<Nd_; ++n)
378     u_[n].conjugate();
379   kx_ *= -1;
380   kz_ *= -1; // Note that no neg vals of kz appear in FlowFields.
381 }
```

7.5.4.12 void BasisFunc::fill (const BasisFunc & *f*)

References Nd_, and u_.

```
372                                  {
373   for (int n=0; n<Nd_; ++n)
374     u_[n].fill(f.u_[n]);
375 }
```

7.5.4.13 bool BasisFunc::geometryCongruent (const BasisFunc & *f*) const

References a_, b_, Lx_, Lz_, Nd_, and Ny_.

Referenced by chebyDist2(), chebyInnerProduct(), congruent(), cross(), dot(), dotgrad(), L2Dist2(), and L2InnerProduct().

```
394                                                 {
395   return (Nd_ == Phi.Nd_ && Ny_==Phi.Ny_ && Lx_==Phi.Lx_ && Lz_==Phi.Lz_
&&
396         a_==Phi.a_ && b_==Phi.b_);
397 }
```

Here is the caller graph for this function:

7.5.4.14 void BasisFunc::ichebyfft (const ChebyTransform & *t*)

References ichebyfft(), Nd_, Physical, Spectral, state_, and u_.

```
445                                                 {
446   assert(state_ == Spectral);
447   for (int n=0; n<Nd_; ++n)
448     u_[n].ichebyfft(t);
449   state_ = Physical;
450 }
```

Here is the call graph for this function:

7.5.4.15 void BasisFunc::ichebyfft ()

References Nd_, Ny_, Physical, Spectral, state_, and u_.

Referenced by ichebyfft().

```
412                       {
413   assert(state_ == Spectral);
414   ChebyTransform t(Ny_);
415   for (int n=0; n<Nd_; ++n)
416     u_[n].ichebyfft(t);
417   state_ = Physical;
418 }
```

Here is the caller graph for this function:

7.5.4.16 void BasisFunc::interpolate (const BasisFunc & *f*)

References a_, b_, Nd_, Physical, Spectral, state_, and u_.

```
319                                       {
320   assert(phi.state_ == Spectral);
321   assert(a_ >= phi.a_ && b_ <= phi.b_);
322   assert(Nd_ == phi.Nd_);
323   for (int n=0; n<Nd_; ++n)
324     u_[n].interpolate(phi.u_[n]);
325   state_ = Physical;
326 }
```

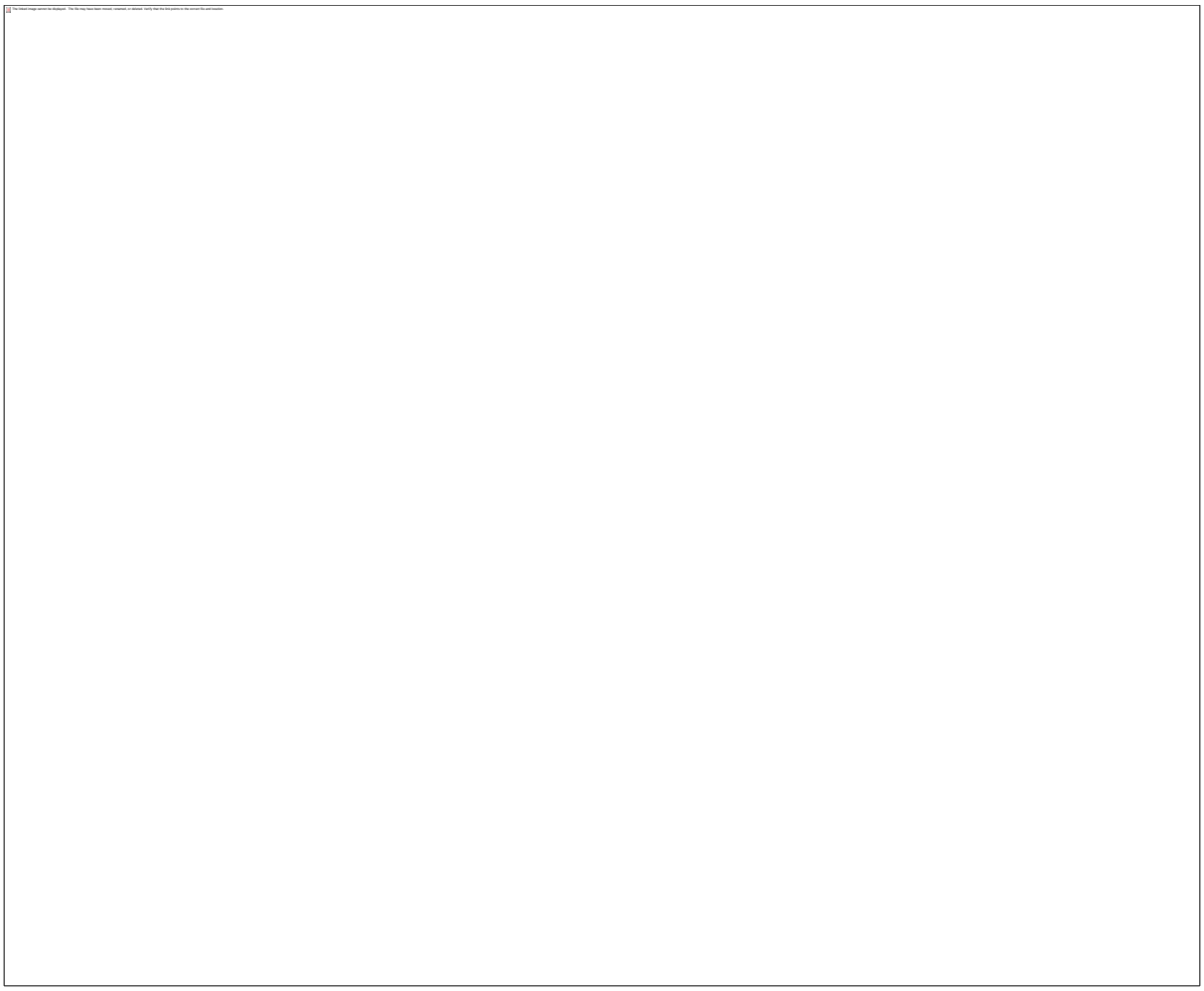
7.5.4.17 int BasisFunc::kx () const [inline]

References kx_.

Referenced by FlowField::addProfile(), chebyDist2(), chebyInnerProduct(), cross(), curl(), divDist2(), divergence(), divNorm2(), dotgrad(), gradient(), L2Dist2(), L2InnerProduct(), laplacian(), xdiff(), and ydiff().

```
209 {return kx_;}
```

Here is the caller graph for this function:



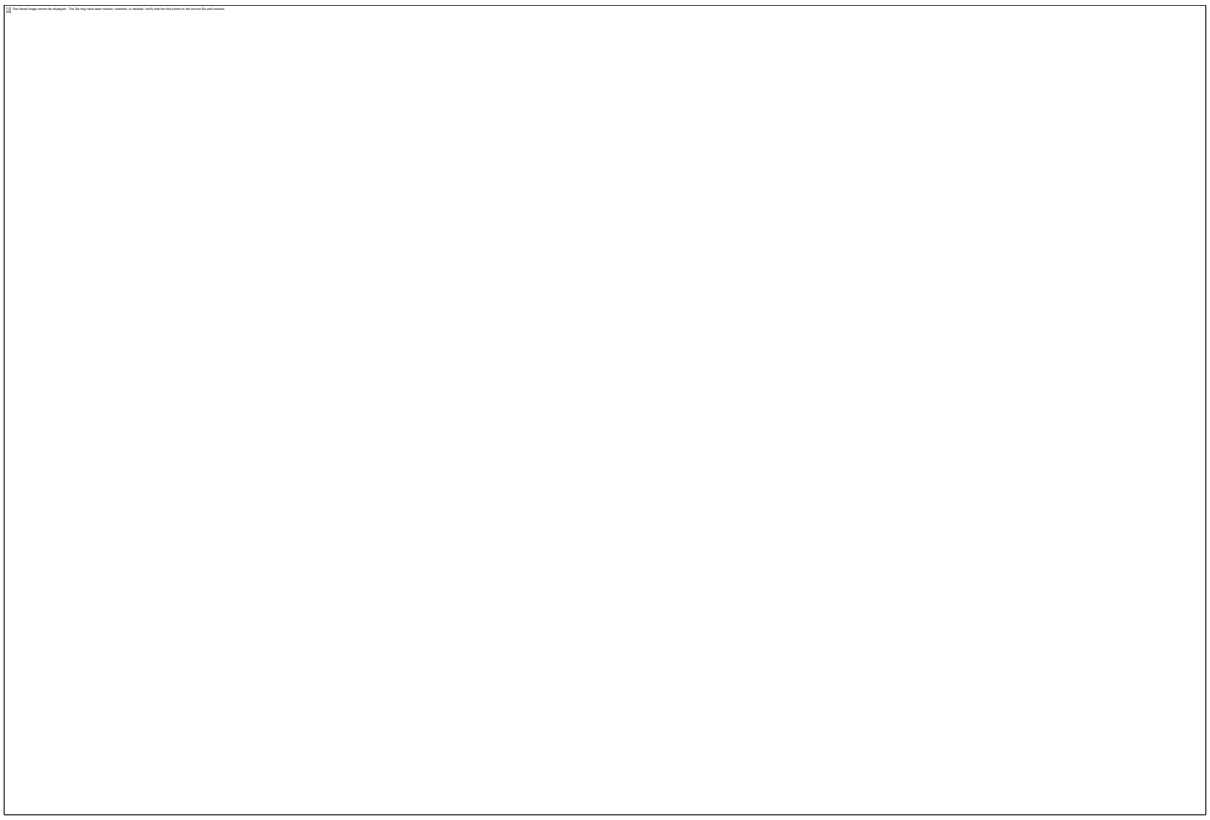
7.5.4.18 int BasisFunc::kz () const [inline]

References kz_.

Referenced by FlowField::addProfile(), chebyDist2(), chebyInnerProduct(), cross(), curl(), divDist2(), divergence(), divNorm2(), dotgrad(), gradient(), L2Dist2(), L2InnerProduct(), laplacian(), ydiff(), and zdiff().

```
210 {return kz_;}
```

Here is the caller graph for this function:



7.5.4.19 Real BasisFunc::lambda () const [inline]

References lambda_.

```
216 {return lambda_;}
```

7.5.4.20 Real BasisFunc::Lx () const [inline]

References Lx_.

Referenced by bcDist2(), bcNorm2(), chebyDist2(), chebyInnerProduct(), chebyNorm(), chebyNorm2(), FlowField::congruent(), cross(), curl(), divDist2(), divergence(), divNorm2(), dotgrad(), gradient(), L2Dist2(), L2InnerProduct(), L2Norm(), L2Norm2(), laplacian(), xdiff(), and ydiff().

```
211 {return Lx_;}
```

Here is the caller graph for this function:

7.5.4.21 Real BasisFunc::Lz () const [inline]

References Lz_.

Referenced by bcDist2(), bcNorm2(), chebyDist2(), chebyInnerProduct(), chebyNorm(), chebyNorm2(), FlowField::congruent(), cross(), curl(), divDist2(), divergence(), divNorm2(), dotgrad(), gradient(), L2Dist2(), L2InnerProduct(), L2Norm(), L2Norm2(), laplacian(), ydiff(), and zdiff().

```
212 {return Lz_;}
```

Here is the caller graph for this function:

7.5.4.22 void BasisFunc::makePhysical (const ChebyTransform & *t*)

References makePhysical(), Nd_, Physical, state_, and u_.

```
456                                     {
457   for (int n=0; n<Nd_; ++n)
458     u_[n].makePhysical(t);
459   state_ = Physical;
460 }
```

Here is the call graph for this function:

7.5.4.23 void BasisFunc::makePhysical ()

References Nd_, Ny_, Physical, state_, and u_.

Referenced by cross(), dot(), dotgrad(), and makePhysical().

```
425                         {
426   ChebyTransform t(Ny_);
427   for (int n=0; n<Nd_; ++n)
428     u_[n].makePhysical(t);
429   state_ = Physical;
430 }
```

Here is the caller graph for this function:

7.5.4.24 void BasisFunc::makeSpectral (const ChebyTransform & *t*)

References makeSpectral(), Nd_, Spectral, state_, and u_.

```
451                                     {
452   for (int n=0; n<Nd_; ++n)
453     u_[n].makeSpectral(t);
454   state_ = Spectral;
455 }
```

Here is the call graph for this function:

7.5.4.25 void BasisFunc::makeSpectral ()

References Nd_, Ny_, Spectral, state_, and u_.

Referenced by cross(), curl(), dotgrad(), makemeen(), makeprtb(), makeSpectral(), makewave(), and makewobl().

```
419                         {
420   ChebyTransform t(Ny_);
421   for (int n=0; n<Nd_; ++n)
422     u_[n].makeSpectral(t);
423   state_ = Spectral;
424 }
```

Here is the caller graph for this function:

7.5.4.26 void BasisFunc::makeState (fieldstate *s*, const ChebyTransform & *t*)

References makeState(), Nd_, state_, and u_.

```
461                                                 {
462   for (int n=0; n<Nd_; ++n)
463     u_[n].makeState(s,t);
464   state_ = s;
465 }
```

Here is the call graph for this function:

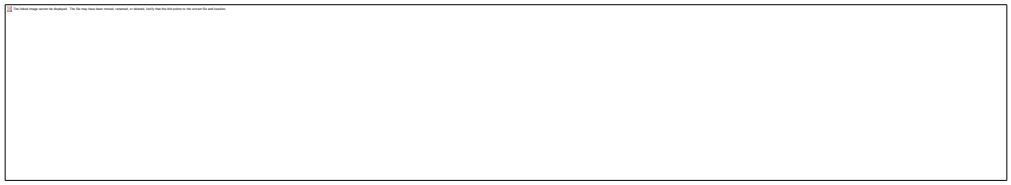
7.5.4.27 void BasisFunc::makeState (fieldstate *s*)

References Nd_, Ny_, state_, and u_.

Referenced by dot(), and makeState().

```
431                                 {
432   ChebyTransform t(Ny_);
433   for (int n=0; n<Nd_; ++n)
434     u_[n].makeState(s,t);
435   state_ = s;
436 }
```

Here is the caller graph for this function:



7.5.4.28 int BasisFunc::Nd () const [inline]

References Nd_.

Referenced by FlowField::addProfile(), chebyDist2(), chebyInnerProduct(), chebyNorm(), chebyNorm2(), L2Dist2(), L2InnerProduct(), L2Norm(), L2Norm2(), and operator==().

```
207 {return Nd_;}
```

Here is the caller graph for this function:

7.5.4.29 int BasisFunc::Ny () const [inline]

References Ny_.

Referenced by FlowField::congruent(), cross(), curl(), divergence(), dot(), dotgrad(), gradient(), and ydiff().

```
208 {return Ny_;}
```

Here is the caller graph for this function:

7.5.4.30 BasisFunc & BasisFunc::operator*= (Complex *c*)

References Nd_, and u_.

```
521                                     {
522   for (int n=0; n<Nd_; ++n)
523     u_[n] *= c;
524   return *this;
525 }
```

7.5.4.31 BasisFunc & BasisFunc::operator*= (Real *c*)

References Nd_, and u_.

```
516                                    {
```

```
517   for (int n=0; n<Nd_; ++n)
518     u_[n] *= c;
519   return *this;
520 }
```

7.5.4.32 BasisFunc & BasisFunc::operator*= (const BasisFunc & *g*)

References Nd_, and u_.

```
500                                         {
501   for (int n=0; n<Nd_; ++n)
502     u_[n] *= phi.u_[n];
503   return *this;
504 }
```

7.5.4.33 BasisFunc & BasisFunc::operator+= (const BasisFunc & *g*)

References Nd_, and u_.

```
506                                         {
507   for (int n=0; n<Nd_; ++n)
508     u_[n] += phi.u_[n];
509   return *this;
510 }
```

7.5.4.34 BasisFunc & BasisFunc::operator-= (const BasisFunc & *g*)

References Nd_, and u_.

```
511                                         {
512   for (int n=0; n<Nd_; ++n)
513     u_[n] -= phi.u_[n];
514   return *this;
515 }
```

7.5.4.35 ComplexChebyCoeff & BasisFunc::operator[] (int *i*)

References Nd_, and u_.

```
487                                       {
488   assert(i>=0 && i<Nd_);
489   return u_[i];
490 }
```

7.5.4.36 const ComplexChebyCoeff & BasisFunc::operator[] (int *i*) const

References Nd_, and u_.

```
483                                                 {
484   assert(i>=0 && i<Nd_);
485   return u_[i];
486 }
```

7.5.4.37 void BasisFunc::randomize (bool *adiri*, bool *bdiri*, Real *decay* = 0.7)

References a_, b_, diff(), I(), kx_, kz_, Lx_, Lz_, Nd_, Ny_, pi, ComplexChebyCoeff::randomize(), setState(), ComplexChebyCoeff::setToZero(), setToZero(), Spectral, u(), u_, ubcFix(), v(), vbcFix(), and w().

```
268                                                 {
269   assert(Nd_ == 3);
270
271   setToZero();
272   setState(Spectral);
273
274   ComplexChebyCoeff u(Ny_, a_, b_, Spectral);
275   ComplexChebyCoeff v(Ny_, a_, b_, Spectral);
276   ComplexChebyCoeff w(Ny_, a_, b_, Spectral);
277   ComplexChebyCoeff vy(Ny_, a_, b_, Spectral);
278
279
280   // Make a u,v part
281   if (kx_ == 0) {
282     u.randomize(decay);
283     ubcFix(u, adiri, bdiri);
284   }
285   else {
286     v.randomize(decay);
287     vbcFix(v, adiri, bdiri);
288     diff(v, u);
289     u *= I*Lx_/(2*pi*kx_);
290   }
291   u_[0] += u;
292   u_[1] += v;
293
294   u.setToZero();
295   v.setToZero();
296   w.setToZero();
297
298   // Make a v,w, part
```

```
299   if (kz_ == 0) {
300     w.randomize(decay);
301     ubcFix(w, adiri, bdiri);
302   }
303   else {
304     v.randomize(decay);
305     vbcFix(v, adiri, bdiri);
306     diff(v, w);
307     w *= I*Lz_/(2*pi*kz_);
308   }
309   u_[2] += w;
310   u_[1] += v;
311
312   if (kx_ == 0 && kz_ == 0) {
313     u_[0].im.setToZero();
314     u_[1].re.setToZero(); // should already be zero
315     u_[2].im.setToZero();
316   }
317 }
```

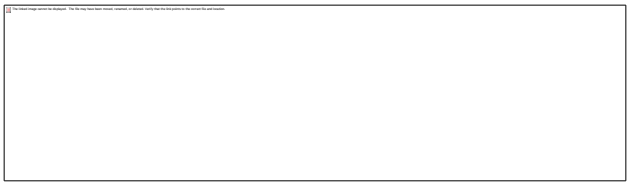
Here is the call graph for this function:

7.5.4.38 void BasisFunc::reflect (const BasisFunc & *f*)

References a(), a_, b(), b_, Even, Nd_, Odd, Physical, Spectral, state_, and u_.

```
328                                     {
329   assert((a_+b_)/2 == phi.a() && b_ <= phi.b() && b_ > phi.a());
330   assert(phi.state_ == Spectral);
331   for (int n=0; n<Nd_; n += 2)
332     u_[n].reflect(phi.u_[n], Odd);
333   for (int n=1; n<Nd_; n += 2)
334     u_[n].reflect(phi.u_[n], Even);
335   state_ = Physical;
336 }
```

Here is the call graph for this function:



7.5.4.39 void BasisFunc::resize (int *Ny*)

References Nd_, Ny_, and u_.

```
338                         {
339   if (N != Ny_) {
340     Ny_ = N;
341     for (int n=0; n<Nd_; ++n)
342       u_[n].resize(Ny_);
343   }
344 }
```

7.5.4.40 Real BasisFunc::rmsval () const [inline]

References lambda_.

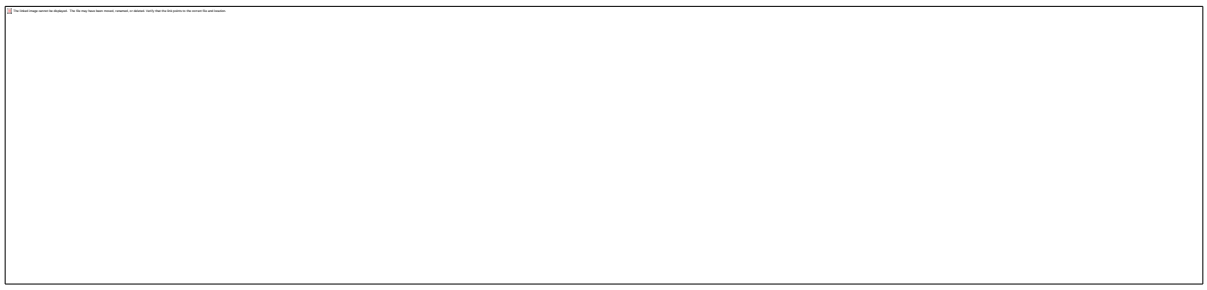
```
215 {return lambda_;}
```

7.5.4.41 void BasisFunc::save (const std::string & *filebase*, fieldstate *s* = Physical) const

References a_, b_, Im(), kx_, kz_, lambda_, Lx_, Lz_, ChebyCoeff::makeState(), Nd_, Ny_, Re(), REAL_DIGITS, state_, and u_.

```
203                                                 {
204   fieldstate origstate = state_;
205   ((BasisFunc&) *this).makeState(savestate);
206
207   string filename(filebase);
208   filename += string(".bf");
209   ofstream os(filename.c_str());
210   os << scientific << setprecision(REAL_DIGITS);
211   char sp=' ';
212   os << '%'
213      <<sp<< Nd_
214      <<sp<< Ny_
215      <<sp<< kx_
216      <<sp<< kz_
217      <<sp<< Lx_
218      <<sp<< Lz_
219      <<sp<< a_
220      <<sp<< b_
221      <<sp<< state_
222      <<sp<< lambda_
223      << '\n';
224
225   for (int ny=0; ny<Ny_; ++ny) {
226     for (int n=0; n<Nd_; ++n)
227       os << Re(u_[n][ny]) <<sp<< Im(u_[n][ny]) <<sp;
228     os << '\n';
229   }
230   os.close();
231
232   ((BasisFunc&) *this).makeState(origstate);
233 }
```

Here is the call graph for this function:



7.5.4.42 void BasisFunc::setBounds (Real *Lx*, Real *Lz*, Real *a*, Real *b*)

References Lx_, Lz_, Nd_, and u_.

Referenced by dotgrad().

```
346                                         {
347   Lx_ = Lx;
348   Lz_ = Lz;
349   for (int n=0; n<Nd_; ++n)
350     u_[n].setBounds(a,b);
351 }
```

Here is the caller graph for this function:

7.5.4.43 void BasisFunc::setkxkz (int *kx*, int *kz*)

References kx_, and kz_.

Referenced by dotgrad().

```
352                                         {
353   kx_=kx;
354   kz_=kz;
355 }
```

Here is the caller graph for this function:

7.5.4.44 void BasisFunc::setRmsval (Real *lamda*)

References lambda_.

```
363                                         {
364   lambda_ = lambda;
365 }
```

7.5.4.45 void BasisFunc::setState (fieldstate *s*)

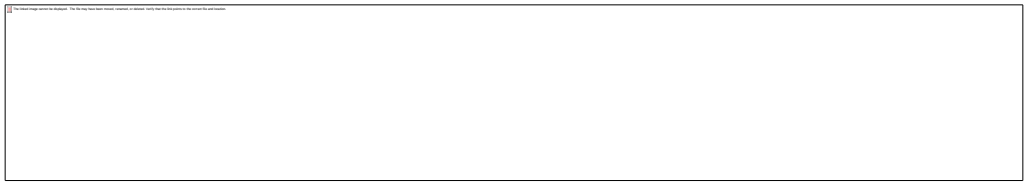
References Nd_, Physical, Spectral, state_, and u_.

Referenced by dotgrad(), and randomize().

```
356                                         {
357   assert(s == Spectral || s == Physical);
```

```
358   for (int n=0; n<Nd_; ++n)
359     u_[n].setState(s);
360   state_ = s;
361 }
```

Here is the caller graph for this function:



7.5.4.46 void BasisFunc::setToZero ()

References Nd_, and u_.

Referenced by curl(), dotgrad(), and randomize().

```
367                         {
368   for (int n=0; n<Nd_; ++n)
369     u_[n].setToZero();
370 }
```

Here is the caller graph for this function:

7.5.4.47 fieldstate BasisFunc::state () const [inline]

References state_.

Referenced by FlowField::addProfile(), FlowField::congruent(), cross(), curl(), divDist2(), divergence(), divNorm2(), dot(), dotgrad(), gradient(), and ydiff().

```
217 {return state_;}
```

Here is the caller graph for this function:

7.5.4.48 ComplexChebyCoeff & BasisFunc::u ()

References u_.

```
471 {return u_[0];}
```

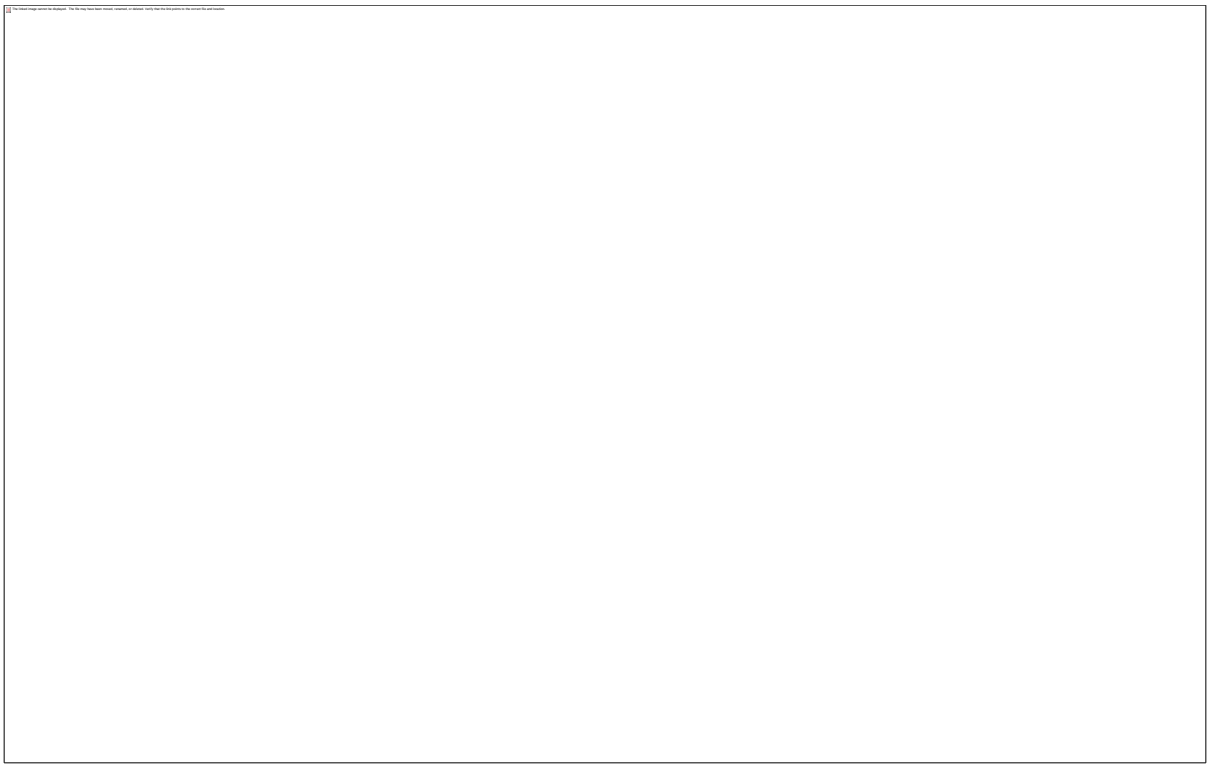
7.5.4.49 const ComplexChebyCoeff & BasisFunc::u () const

References u_.

Referenced by bcDist2(), bcNorm2(), divDist2(), divNorm2(), dotgrad(), FlowField::dudy_a(), FlowField::dudy_b(), laplacian(), makemeen(), makeprtb(), makeroll(), makewave(), makewobl(), randomize(), and ydiff().

```
468 {return u_[0];}
```

Here is the caller graph for this function:



7.5.4.50 ComplexChebyCoeff & BasisFunc::v ()

References u_.

```
472 {return u_[1];}
```

7.5.4.51 const ComplexChebyCoeff & BasisFunc::v () const

References u_.

Referenced by bcDist2(), bcNorm2(), divDist2(), divNorm2(), dotgrad(), laplacian(), makeprtb(), makeroll(), makewave(), makewobl(), randomize(), and ydiff().

```
469 {return u_[1];}
```

Here is the caller graph for this function:

7.5.4.52 ComplexChebyCoeff & BasisFunc::w ()

References u_.

```
473 {return u_[2];}
```

7.5.4.53 const ComplexChebyCoeff & BasisFunc::w () const

References u_.

Referenced by bcDist2(), bcNorm2(), divDist2(), divNorm2(), dotgrad(), laplacian(), makeprtb(), makeroll(), makewave(), makewobl(), randomize(), and ydiff().

```
470 {return u_[2];}
```

Here is the caller graph for this function:

7.5.5 Member Data Documentation

7.5.5.1 Real BasisFunc::a_ [protected]

Referenced by a(), BasisFunc(), binaryDump(), binaryLoad(), geometryCongruent(), interpolate(), randomize(), reflect(), and save().

7.5.5.2 Real BasisFunc::b_ [protected]

Referenced by b(), BasisFunc(), binaryDump(), binaryLoad(), geometryCongruent(), interpolate(), randomize(), reflect(), and save().

7.5.5.3 int BasisFunc::kx_ [protected]

Referenced by BasisFunc(), binaryDump(), binaryLoad(), congruent(), conjugate(), kx(), randomize(), save(), and setkxkz().

7.5.5.4 int BasisFunc::kz_ [protected]

Referenced by BasisFunc(), binaryDump(), binaryLoad(), congruent(), conjugate(), kz(), randomize(), save(), and setkxkz().

7.5.5.5 Real BasisFunc::lambda_ [protected]

Referenced by BasisFunc(), binaryDump(), binaryLoad(), lambda(), rmsval(), save(), and setRmsval().

7.5.5.6 Real BasisFunc::Lx_ [protected]

Referenced by BasisFunc(), binaryDump(), binaryLoad(), geometryCongruent(), Lx(), randomize(), save(), and setBounds().

7.5.5.7 Real BasisFunc::Lz_ [protected]

Referenced by BasisFunc(), binaryDump(), binaryLoad(), geometryCongruent(), Lz(), randomize(), save(), and setBounds().

7.5.5.8 int BasisFunc::Nd_ [protected]

Referenced by BasisFunc(), bcNorm(), binaryDump(), binaryLoad(), chebyfft(), component(), conjugate(), fill(), geometryCongruent(), ichebyfft(), interpolate(), makePhysical(), makeSpectral(), makeState(), Nd(), operator*=(), operator+=(), operator-=(), operator[](), randomize(), reflect(), resize(), save(), setBounds(), setState(), and setToZero().

7.5.5.9 int BasisFunc::Ny_ [protected]

Referenced by BasisFunc(), binaryDump(), binaryLoad(), chebyfft(), geometryCongruent(), ichebyfft(), makePhysical(), makeSpectral(), makeState(), Ny(), randomize(), resize(), and save().

7.5.5.10 fieldstate BasisFunc::state_ [protected]

Referenced by BasisFunc(), binaryDump(), binaryLoad(), chebyfft(), congruent(), ichebyfft(), interpolate(), makePhysical(), makeSpectral(), makeState(), reflect(), save(), setState(), and state().

7.5.5.11 array<ComplexChebyCoeff> BasisFunc::u_ [protected]

Referenced by BasisFunc(), bcNorm(), binaryDump(), binaryLoad(), chebyfft(), component(), conjugate(), fill(), ichebyfft(), interpolate(), makePhysical(), makeSpectral(), makeState(), operator*=(), operator+=(), operator-=(), operator[](), randomize(), reflect(), resize(), save(), setBounds(), setState(), setToZero(), u(), v(), and w().

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

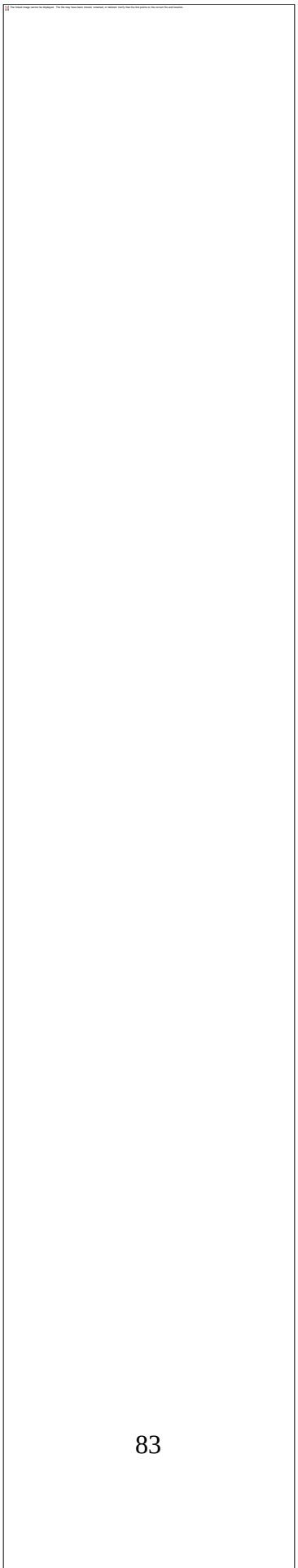
- _cuda/channelflow-0.9.22/channelflow/basisfunc.h
- _cuda/channelflow-0.9.22/channelflow/basisfunc.cpp

7.5.5.13

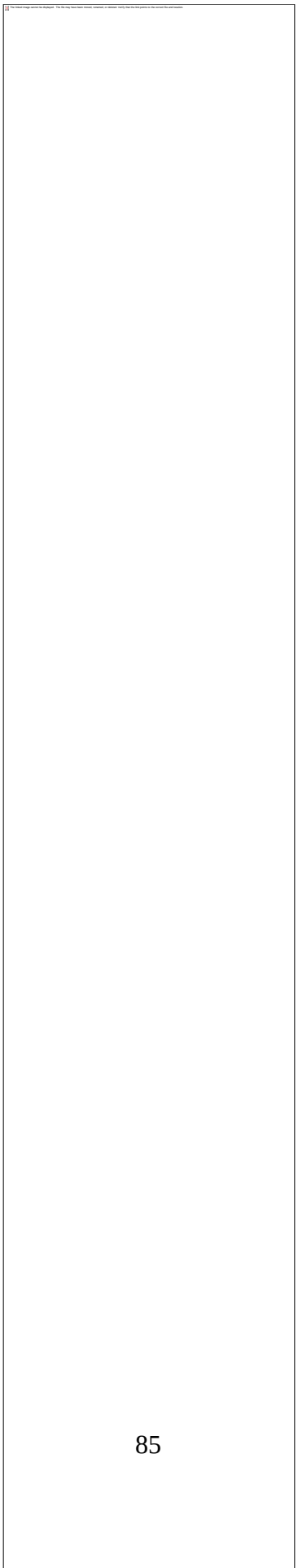
7.6 ChebyCoeff Class Reference

#include <chebyshev.h>

Inheritance diagram for ChebyCoeff:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

Collaboration diagram for ChebyCoeff:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.6.1 Public Member Functions

- ChebyCoeff ()
- ChebyCoeff (int N, Real a, Real b, fieldstate s=Spectral)
- *Null constructor.* ChebyCoeff (const Vector &v, Real a, Real b, fieldstate s=Spectral)
- ChebyCoeff (int N, const ChebyCoeff &g)
- ChebyCoeff (const std::string &filebase)
- ~ChebyCoeff ()
- void save (const std::string &filebase, fieldstate s=Physical) const
- void binaryDump (std::ostream &os) const
- void binaryLoad (std::istream &is)
- void randomize (Real decay, bool a_dirichlet=false, bool b_dirichlet=false)
- void setBounds (Real a, Real b)
- void setState (fieldstate s)
- void setToZero ()
- void fill (const ChebyCoeff &g)
- void interpolate (const ChebyCoeff &g)
- void reflect (const ChebyCoeff &g, parity p)
- Real eval_a () const
- Real eval_b () const
- Real eval (Real x) const
- ChebyCoeff eval (const Vector &x) const
- void eval (const Vector &x, ChebyCoeff &g) const
- Real a () const
- Real b () const
- Real L () const
- int N () const
- int numModes () const
- fieldstate state () const
- Real mean () const
- ChebyCoeff & operator*= (Real c)
- ChebyCoeff & operator+= (const ChebyCoeff &g)
- ChebyCoeff & operator-= (const ChebyCoeff &g)
- ChebyCoeff & operator*= (const ChebyCoeff &g)
- void chebyfft ()
- void ichebyfft ()
- void makeSpectral ()
- void makePhysical ()
- void makeState (fieldstate s)
- void chebyfft (const ChebyTransform &t)
- void ichebyfft (const ChebyTransform &t)
- void makeSpectral (const ChebyTransform &t)
- void makePhysical (const ChebyTransform &t)

- void makeState (fieldstate s, const ChebyTransform &t)
- bool congruent (const ChebyCoeff &g) const

7.6.2 Private Attributes

- Real a_
- Real b_
- fieldstate state_

7.6.3 Friends

- class ChebyTransform
- void swap (ChebyCoeff &f, ChebyCoeff &g)

7.6.4 Constructor & Destructor Documentation

7.6.4.1 ChebyCoeff::ChebyCoeff ()

```
171   :
172   Vector(),
173   a_(0),
174   b_(0),
175   state_(Spectral)
176 {}
```

7.6.4.2 ChebyCoeff::ChebyCoeff (int *N*, Real *a*, Real *b*, fieldstate *s* = Spectral)

Null constructor.

References a_, and b_.

```
179   :
180   Vector(N),
181   a_(a),
182   b_(b),
183   state_(s)
184 {
185   assert( b_ > a_);
186 }
```

7.6.4.3 ChebyCoeff::ChebyCoeff (const Vector & *v*, Real *a*, Real *b*, fieldstate *s* = Spectral)

References a_, and b_.

```
189   :
190   Vector(v),
191   a_(a),
192   b_(b),
193   state_(s)
194 {
195   assert( b_ > a_);
196 }
```

7.6.4.4 ChebyCoeff::ChebyCoeff (int *N*, const ChebyCoeff & *g*)

References a_, b_, Vector::data_, lesser(), Vector::N_, Spectral, and state_.

```
199   :
200   Vector(N),
201   a_(u.a_),
202   b_(u.b_),
203   state_(u.state_)
204 {
205   assert(b_ > a_);
206   assert(state_ == Spectral);
207   int Ncommon = lesser(N, u.N_);
208   int i; // MSVC++ FOR-SCOPE BUG
209   for (i=0; i<Ncommon; ++i)
210     data_[i] = u.data_[i];
211   for (i=Ncommon; i<N_; ++i)
212     data_[i] = 0.0;
213
214 }
```

Here is the call graph for this function:

7.6.4.5 ChebyCoeff::ChebyCoeff (const std::string & *filebase*)

References a_, b_, N(), Vector::resize(), and state_.

```
217   :
218   Vector(0),
219   a_(0),
```

```
220   b_(0),
221   state_(Spectral)
222 {
223   string filename(filebase);
224   filename += string(".asc");
225   ifstream is(filename.c_str());
226   if (!is.good()) {
227     cerr << "ChebyCoeff::ChebyCoeff(filebase) : can't open file " << filename << endl;
228     abort();
229   }
230
231   // Read in header. Form is "%N a b s"
232   char c;
233   int N;
234   is >> c;
235   if (c != '%') {
236     string message("ChebyCoeff(filebase): bad header in file ");
237     message += filename;
238     cerr << message << endl;
239     assert(false);
240   }
241   is >> N >> a_ >> b_ >> state_;
242   resize(N);
243   assert(is.good());
244   for (int i=0; i<N; ++i) {
245     is >> (*this)[i];
246     assert(is.good());
247   }
248
249   is.close();
250 }
```

Here is the call graph for this function:

7.6.4.6 ChebyCoeff::~ChebyCoeff ()

```
264 {}
```

7.6.5 Member Function Documentation

7.6.5.1 Real ChebyCoeff::a () const [inline]

References a_.

Referenced by ComplexChebyCoeff::a(), TurbStats::bulkReynolds(), TurbStats::centerlineReynolds(), chebyDist2(), chebyInnerProduct(), chebyNorm2(), convectionNL(), diff(), diff2(), divergenceNL(), integrate(), L1Dist(), L1Norm(), L2InnerProduct(), L2Norm2(), linearizedNL(), TurbStats::msave(), reflect(), rotationalNL(), ComplexChebyCoeff::save(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), ubcFix(), TurbStats::uu(), vbcFix(), and TurbStats::yplus().

```
341 {return a_;}
```

Here is the caller graph for this function:

7.6.5.2 Real ChebyCoeff::b () const [inline]

References b_.

Referenced by ComplexChebyCoeff::b(), TurbStats::bulkReynolds(), TurbStats::centerlineReynolds(), chebyDist2(), chebyInnerProduct(), chebyNorm2(), convectionNL(), diff(), diff2(), divergenceNL(), integrate(), L1Dist(), L1Norm(), L2InnerProduct(), L2Norm2(), linearizedNL(), TurbStats::msave(), reflect(), rotationalNL(), ComplexChebyCoeff::save(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), ubcFix(), TurbStats::uu(), vbcFix(), and TurbStats::yplus().

```
342 {return b_;}
```

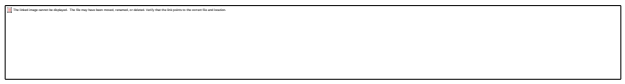
Here is the caller graph for this function:

7.6.5.3 void ChebyCoeff::binaryDump (std::ostream & *os*) const

References a_, b_, Vector::data_, Vector::N_, state_, and write().
Referenced by ComplexChebyCoeff::binaryDump().

```
532                                     {
533   write(os, N_);
534   write(os, a_);
535   write(os, b_);
536   write(os, state_);
537   for (int i=0; i<N_; ++i)
538     write(os, data_[i]);
539 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

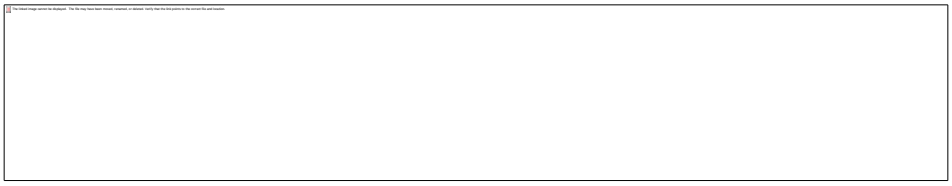
7.6.5.4 void ChebyCoeff::binaryLoad (std::istream & *is*)

References a_, b_, Vector::data_, Vector::N_, read(), Vector::resize(), and state_.
Referenced by ComplexChebyCoeff::binaryLoad().

```
540                                     {
541   if (!is.good()) {
542     cerr << "ChebyCoeff::binaryLoad(istream& is) : input error\n";
543     abort();
544   }
545   int newN_;
546   read(is, newN_);
547   read(is, a_);
548   read(is, b_);
549   read(is, state_);
550   resize(newN_);
551   for (int i=0; i<N_; ++i) {
552     if (!is.good()) {
553       cerr << "ChebyCoeff::binaryLoad(istream& is) : input error\n";
554       abort();
```

```
555     }
556     read(is, data_[i]);
557   }
558 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.5.5 void ChebyCoeff::chebyfft (const ChebyTransform & *t*)

References ChebyTransform::cosfftw_plan_, Vector::data_, ChebyTransform::N(), Vector::N_, Physical, Spectral, and state_.

```
292                                         {
293   assert(t.N() == N_);
294   assert(state_ == Physical);
295   if (N_ < 2) {
296     state_ = Spectral;
297     return;
298   }
299
300   fftw_execute_r2r(t.cosfftw_plan_, data_, data_);
301
302   // Factors of 1/2 appear on 0th and (N-1)th coeff because of reln btwn
303   // chebyshev and cosine transform. Normalization factor nrm=1/(N-1) is
304   // needed because FFTW does unnormalized transforms. After this loop,
305   // data_[n] equals the coefficient of T_n.
306   Real nrm = 1.0/(N_-1);
307   data_[0] *= 0.5*nrm;
308   for (int n=1; n<N_-1; ++n)
309     data_[n] *= nrm;
310   data_[N_-1] *= 0.5*nrm;
311
312   state_ = Spectral;
313   return;
314 }
```

Here is the call graph for this function:

7.6.5.6 void ChebyCoeff::chebyfft ()

References Vector::N_.

Referenced by ComplexChebyCoeff::chebyfft(), makeSpectral(), and makeState().

```
266                         {
267   ChebyTransform t(N_);
268   chebyfft(t);
269 }
```

Here is the caller graph for this function:

7.6.5.7 bool ChebyCoeff::congruent (const ChebyCoeff & *g*) const

References a_, b_, Vector::N_, and state_.

Referenced by ComplexChebyCoeff::congruent(), L1Dist(), LinfDist(), operator*=(), operator+=(), operator-=(), and operator==().

```
594                                         {
595   return (v.N_==N_  && v.a_==a_ && v.b_==b_ &&  v.state_==state_)
596     ? true : false;
597 }
```

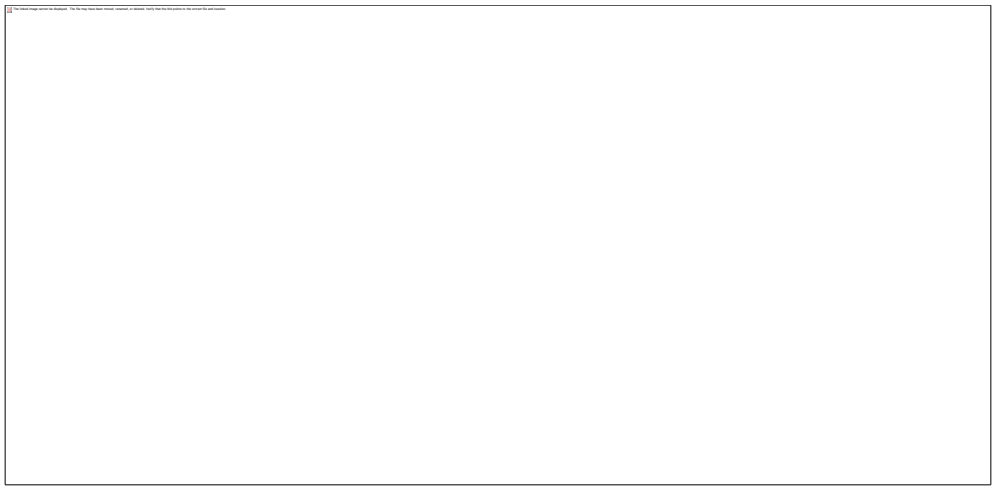
Here is the caller graph for this function:

7.6.5.8 void ChebyCoeff::eval (const Vector & *x*, ChebyCoeff & *g*) const

References a_, b_, Vector::data_, Vector::length(), N(), Vector::N_, Physical, Vector::resize(), setBounds(), setState(), Spectral, and state_.

```
484                                                {
485   assert(state_ == Spectral);
486   int N=x.length();
487   if (f.length() != N)
488     f.resize(N);
489   f.setBounds(a_, b_);
490   f.setState(Physical);
491
492   int M=N_;
493   for (int i=0; i<N; ++i) {
494     Real y = (2*x[i]-a_-b_)/(b_-a_);
495     Real y2 = 2*y;
496     Real d=0.0;
497     Real dd=0.0;
498     for (int j=M-1; j>0; --j) {
499       Real sv=d;
500       d = y2*d - dd + data_[j];
501       dd=sv;
502     }
503     f[i] = y*d - dd + data_[0]; // NR has 0.5*c[0], but that gives wrong results!
504   }
505 }
```

Here is the call graph for this function:



7.6.5.9 ChebyCoeff ChebyCoeff::eval (const Vector & *x*) const

References a_, b_, eval(), Vector::N_, and Physical.

```
477                                         {
478   ChebyCoeff f(N_, a_, b_, Physical);
479   eval(x, f);
480   return f;
481 }
```

Here is the call graph for this function:

7.6.5.10 Real ChebyCoeff::eval (Real *x*) const

References a_, b_, Vector::data_, Vector::N_, Spectral, and state_.

Referenced by TurbStats::centerlineReynolds(), ComplexChebyCoeff::eval(), eval(), interpolate(), and reflect().

```
461                                 {
462   assert(state_ == Spectral);
463   if (N_==0)
464     return NAN;
465   Real y = (2*x-a_-b_)/(b_-a_);
466   Real y2 = 2*y;
467   Real d=0.0;
468   Real dd=0.0;
469   for (int j=N_-1; j>0; --j) {
470     Real sv=d;
```

```
471     d = y2*d - dd + data_[j];
472     dd=sv;
473   }
474   return (y*d - dd + data_[0]); // NR has 0.5*c[0], but that gives wrong results!
475 }
```

Here is the caller graph for this function:

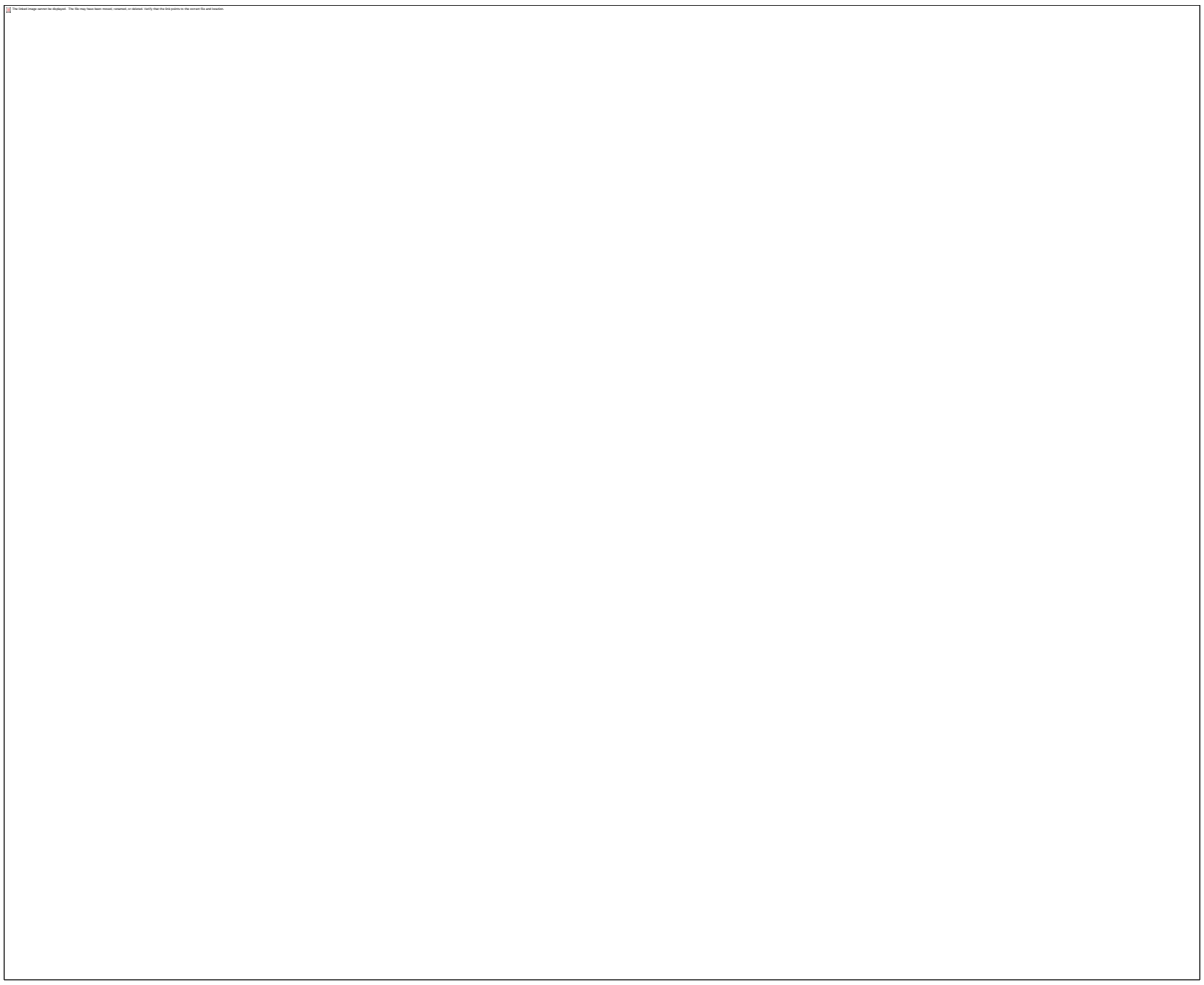
7.6.5.11 Real ChebyCoeff::eval_a () const

References Vector::data_, Vector::N_, Spectral, and state_.

Referenced by DNSAlgorithm::DNSAlgorithm(), FlowField::dudy_a(), ComplexChebyCoeff::eval_a(), TauSolver::influenceCorrection(), L1Dist(), L1Norm(), randomize(), PoissonSolver::solve(), TauSolver::TauSolver(), ubcFix(), TurbStats::ustar(), vbcFix(), HelmholtzSolver::verify(), and TauSolver::verify_P_and_v().

```
448                         {
449   if (N_==0)
450     return NAN;
451   if (state_ == Spectral) {
452     Real sum=0.0;
453     for (int n=N_-1; n>=0; --n)
454       sum += data_[n] * ((n%2 == 0) ? 1 : -1);
455     return sum;
456   }
457   else
458     return data_[N_-1];
459 }
```

Here is the caller graph for this function:



7.6.5.12 Real ChebyCoeff::eval_b () const

References Vector::data_, Vector::N_, Spectral, and state_.

Referenced by DNSAlgorithm::DNSAlgorithm(), FlowField::dudy_b(), ComplexChebyCoeff::eval_b(), TauSolver::influenceCorrection(), L1Dist(), L1Norm(), randomize(), PoissonSolver::solve(), TauSolver::TauSolver(), ubcFix(), TurbStats::ustar(), vbcFix(), HelmholtzSolver::verify(), and TauSolver::verify_P_and_v().

```
435                         {
436   if (N_==0)
437     return NAN;
438   if (state_ == Spectral) {
439     Real sum = 0.0;
440     for (int n=N_-1; n>=0; --n)
441       sum += data_[n];
442     return sum;
443   }
444   else
```

```
445     return data_[0];
446 }
```

Here is the caller graph for this function:

7.6.5.13 void ChebyCoeff::fill (const ChebyCoeff & *g*)

References Vector::data_, lesser(), Vector::N_, Spectral, and state_.

Referenced by ComplexChebyCoeff::fill().

```
379                                 {
380   assert(v.state_ == Spectral);
381   assert(state_ == Spectral);
382   int Ncommon = lesser(N_, v.N_);
383   int i; // MSVC++ FOR-SCOPE BUG
384   for (i=0; i<Ncommon; ++i)
385     data_[i] = v.data_[i];
386   for (i=Ncommon; i<N_; ++i)
387     data_[i] = 0.0;
388 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

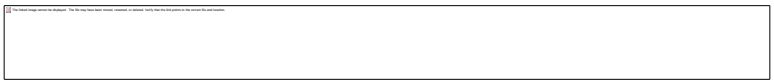
7.6.5.14 void ChebyCoeff::ichebyfft (const ChebyTransform & *t*)

References ChebyTransform::cosfftw_plan_, Vector::data_, ChebyTransform::N(), Vector::N_, Physical, Spectral, and state_.

```
316                                 {
317   assert(t.N() == N_);
318   assert(state_ == Spectral);
319   if (N_ < 2) {
320     state_ = Physical;
321     return;
322   }
323
324   // Factors of 2 undo the 1/2's introduced in ::chebyfft.
325   data_[0]    *= 2.0;
326   data_[N_-1] *= 2.0;
327   fftw_execute_r2r(t.cosfftw_plan_, data_, data_);
328   for (int n=0; n<N_; ++n)
```

```
329     data_[n] *= 0.5;
330
331   state_ = Physical;
332   return;
333 }
```

Here is the call graph for this function:



7.6.5.15 void ChebyCoeff::ichebyfft ()

References Vector::N_.

Referenced by ComplexChebyCoeff::ichebyfft(), makePhysical(), and makeState().

```
270                         {
271   ChebyTransform t(N_);
272   ichebyfft(t);
273 }
```

Here is the caller graph for this function:

7.6.5.16 void ChebyCoeff::interpolate (const ChebyCoeff & *g*)

References a_, b_, Vector::data_, eval(), Vector::N_, Physical, pi, Spectral, and state_.

Referenced by ComplexChebyCoeff::interpolate().

```
389                                     {
```

```
390   assert(a_ >= v.a_ && b_ <= v.b_);
391   assert(v.state_ == Spectral);
392   state_ = Physical;
393   Real piN = pi/(N_-1);
394   Real width = (b_-a_)/2;
395   Real center = (b_+a_)/2;
396   for (int n=0; n<N_; ++n)
397     data_[n] = v.eval(center + width*cos(n*piN));
398   //makeSpectral();
399 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.6.5.17 Real ChebyCoeff::L () const [inline]

References a_, and b_.

Referenced by diff(), and ComplexChebyCoeff::L().

```
343 {return b_-a_;}
```

Here is the caller graph for this function:

7.6.5.18 void ChebyCoeff::makePhysical (const ChebyTransform & *t*)

References ichebyfft(), Spectral, and state_.

```
338                                     {
339   if (state_ == Spectral)
340     ichebyfft(t);
341 }
```

Here is the call graph for this function:

7.6.5.19 void ChebyCoeff::makePhysical ()

References Vector::N_.

Referenced by changeBaseFlow(), DNSAlgorithm::DNSAlgorithm(), energy(), L1Dist(), L1Norm(), linearizedNL(), LinfDist(), LinfNorm(), ComplexChebyCoeff::makePhysical(), reflect(), rotationalNL(), TurbStats::TurbStats(), and TurbStats::ustar().

```
278                         {
279   ChebyTransform t(N_);
280   makePhysical(t);
281 }
```

Here is the caller graph for this function:

7.6.5.20 void ChebyCoeff::makeSpectral (const ChebyTransform & *t*)

References chebyfft(), Physical, and state_.

```
334                                         {
335   if (state_ == Physical)
336     chebyfft(t);
337 }
```

Here is the call graph for this function:

7.6.5.21 void ChebyCoeff::makeSpectral ()

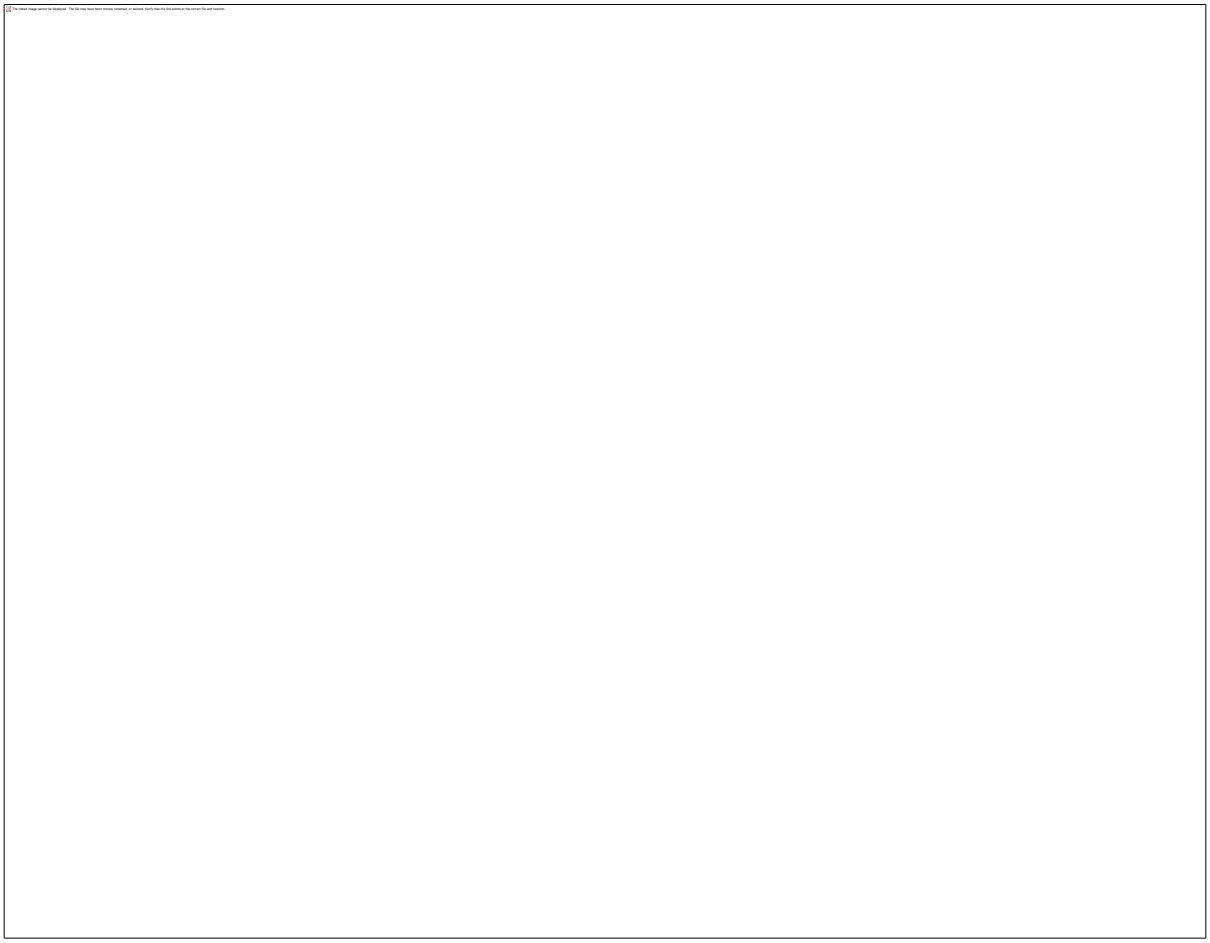
References Vector::N_.

Referenced by TurbStats::bulkReynolds(), TurbStats::centerlineReynolds(), convectionNL(), DNSAlgorithm::DNSAlgorithm(), L1Dist(), L1Norm(), legendre(), linearizedNL(), makeroll(), ComplexChebyCoeff::makeSpectral(), PressureSolver::PressureSolver(), reflect(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), and TurbStats::ustar().

```
274                         {
275   ChebyTransform t(N_);
276   makeSpectral(t);
```

```
277 }
```

Here is the caller graph for this function:

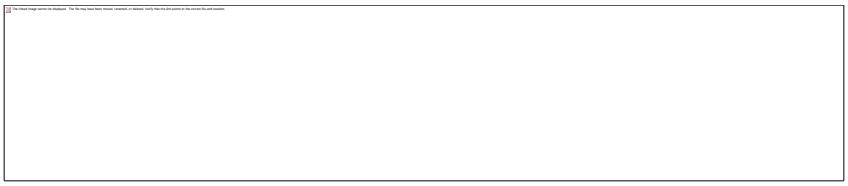


7.6.5.22 void ChebyCoeff::makeState (fieldstate *s*, const ChebyTransform & *t*)

References chebyfft(), ichebyfft(), Physical, and state_.

```
342                                                 {
343   if (state_ != s) {       // need to change state?
344     if (state_==Physical)  // state is Physical; change to Spectral
345       chebyfft(t);
346     else
347       ichebyfft(t);  // state is Spectral; change to Physical
348   }
349 }
```

Here is the call graph for this function:



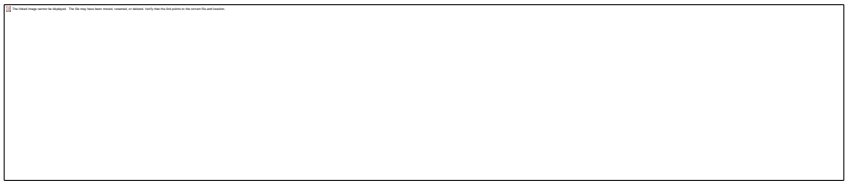
7.6.5.23 void ChebyCoeff::makeState (fieldstate *s*)

References chebyfft(), ichebyfft(), Vector::N_, Physical, and state_.

Referenced by changeBaseFlow(), convectionNL(), divergenceNL(), energy(), linearizedNL(), ComplexChebyCoeff::makeState(), randomize(), rotationalNL(), ComplexChebyCoeff::save(), BasisFunc::save(), skewsymmetricNL(), skewsymmetricNL_GPU(), and skewsymmetricNL_THREAD().

```
282                                 {
283   if (state_ != s) {       // need to change state?
284     ChebyTransform t(N_);
285     if (state_==Physical)  // state is Physical; change to Spectral
286       chebyfft(t);
287     else
288       ichebyfft(t);  // state is Spectral; change to Physical
289   }
290 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.5.24 Real ChebyCoeff::mean () const

References Vector::data_, Vector::N_, Spectral, and state_.

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), TurbStats::bulkReynolds(), DNSAlgorithm::DNSAlgorithm(), integrate(), ComplexChebyCoeff::mean(), HelmholtzSolver::residual(), HelmholtzSolver::solve(), and HelmholtzSolver::verify().

```
507                         {
508   assert(state_ == Spectral);
509   Real sum = data_[0];
510   for (int n=2; n<N_; n+=2)
511     sum -= data_[n]/(n*n-1); // *2
512   return sum;               // /2
513 }
```

Here is the caller graph for this function:

7.6.5.25 int ChebyCoeff::N () const [inline]

Reimplemented from Vector.

References Vector::N_.

Referenced by TurbStats::bulkReynolds(), TurbStats::centerlineReynolds(), FlowField::CFLfactor(), ChebyCoeff(), diff(), eval(), L1Dist(), L1Norm(), linearizedNL(), LinfDist(), LinfNorm(), ComplexChebyCoeff::N(), FlowField::operator+=(), FlowField::operator-=(), PressureSolver::PressureSolver(), rotationalNL(), and TurbStats::ustar().

```
344 {return N_;}
```

Here is the caller graph for this function:

7.6.5.26 int ChebyCoeff::numModes () const [inline]

References Vector::N_.

Referenced by changeBaseFlow(), chebyDist2(), ComplexChebyCoeff::chebyfft(), chebyInnerProduct(), chebyNorm2(), convectionNL(), diff(), divergenceNL(), energy(), ComplexChebyCoeff::ichebyfft(), integrate(), L2InnerProduct(), L2Norm2(), ComplexChebyCoeff::makePhysical(), ComplexChebyCoeff::makeSpectral(), ComplexChebyCoeff::makeState(), TurbStats::msave(), ComplexChebyCoeff::numModes(), ComplexChebyCoeff::operator*=(), operator==(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), TauSolver::tauDist(), TauSolver::tauNorm(), TurbStats::TurbStats(), TurbStats::uu(), and TurbStats::yplus().

```
345 {return N_;}
```

Here is the caller graph for this function:

7.6.5.27 ChebyCoeff & ChebyCoeff::operator*= (const ChebyCoeff & *g*)

References congruent(), Vector::data_, Vector::N_, Physical, and state_.

```
586                                             {
587   assert(congruent(a));
588   assert(state_ == Physical);
589   for (int i=0; i<N_; ++i)
590     data_[i] *= a.data_[i];
591   return *this;
592 }
```

Here is the call graph for this function:

7.6.5.28 ChebyCoeff & ChebyCoeff::operator*= (Real *c*)

Reimplemented from Vector.

References Vector::data_, and Vector::N_.

```
560                                     {
561   for (int i=0; i<N_; ++i)
562     data_[i] *= c;
563   return *this;
564 }
```

7.6.5.29 ChebyCoeff & ChebyCoeff::operator+= (const ChebyCoeff & *g*)

Reimplemented from Vector.

References congruent(), Vector::data_, and Vector::N_.

```
566                                             {
567   assert(congruent(a));
568   for (int i=0; i<N_; ++i)
569     data_[i] += a.data_[i];
570   return *this;
571 }
```

Here is the call graph for this function:

7.6.5.30 ChebyCoeff & ChebyCoeff::operator-= (const ChebyCoeff & *g*)

Reimplemented from Vector.

References congruent(), Vector::data_, and Vector::N_.

```
573                                          {
574   assert(congruent(a));
575   for (int i=0; i<N_; ++i)
576     data_[i] -= a.data_[i];
577   return *this;
578 }
```

Here is the call graph for this function:

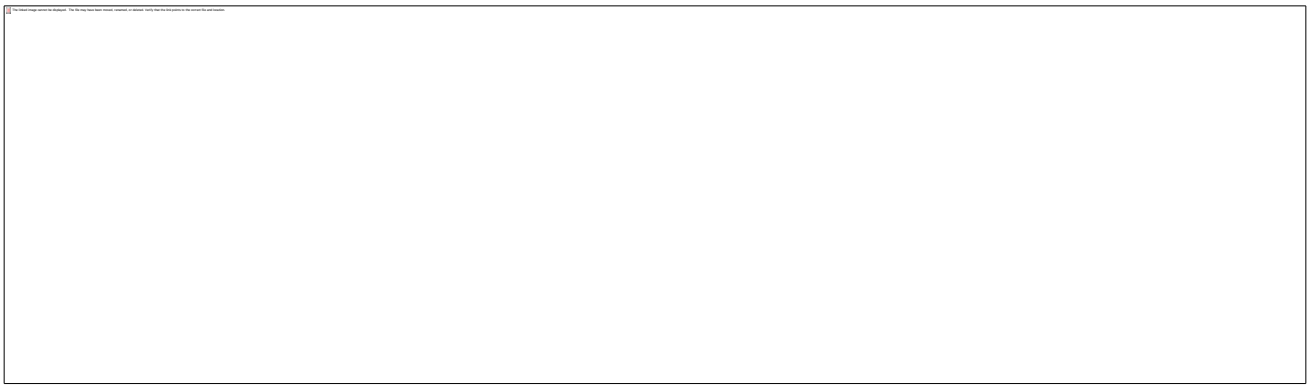
7.6.5.31 void ChebyCoeff::randomize (Real *decay*, bool *a_dirichlet* = false, bool *b_dirichlet* = false)

References Vector::data_, eval_a(), eval_b(), makeState(), Vector::N_, randomReal(), Spectral, and state_.

Referenced by ComplexChebyCoeff::randomize().

```
352                                                  {
353   fieldstate startState = state_;
354   state_ = Spectral;
355   Real pow_decay_n = 1.0;
356   for (int n=0; n<N_; ++n) {
357     data_[n] = pow_decay_n * randomReal();
358     pow_decay_n *= decay;
359   }
360   if (N_ == 1 && (a_dirichlet || b_dirichlet))
361     data_[0] = 0.0;
362   else if (N_ >= 2 && (a_dirichlet && b_dirichlet)) {
363     data_[1] -= 0.5*(eval_b() - eval_a());
364     data_[0] -= 0.5*(eval_b() + eval_a());
365   }
366   else if (N_ >= 2 && a_dirichlet)
367     data_[0] -= eval_a();
368   else if (N_ >= 2 && b_dirichlet)
369     data_[0] -= eval_b();
370
371   makeState(startState);
372 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

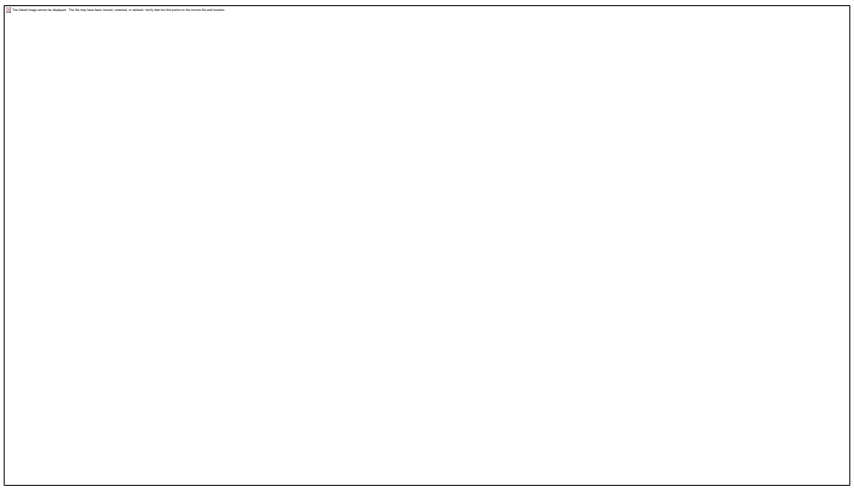
7.6.5.32 void ChebyCoeff::reflect (const ChebyCoeff & *g*, parity *p*)

References a(), a_, b(), b_, Vector::data_, eval(), makePhysical(), makeSpectral(), Vector::N_, Odd, Physical, pi, Spectral, and state_.

Referenced by ComplexChebyCoeff::reflect().

```
401                                         {
402   assert((a_+b_)/2 == v.a() && b_ <= v.b() && b_ > v.a());
403   assert(v.state_ == Spectral);
404   state_ = Physical;
405   Real piN = pi/(N_-1);
406   Real width = (b_-a_)/2;
407   Real center = (b_+a_)/2;
408   int N2=N_/2;
409   int N1=N_-1;
410   int sign = (p==Odd) ? -1 : 1;
411   for (int n=0; n<N2; ++n) {
412     Real tmp = v.eval(center + width*cos(n*piN));
413     data_[n] = tmp;
414     data_[N1-n] = sign*tmp;
415   }
416   ChebyTransform t(N_);
417   makeSpectral(t);
418   for (int n=2*N_/3; n<N_; ++ n)
419     data_[n] = 0.0;
420   makePhysical(t);
421 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.5.33 void ChebyCoeff::save (const std::string & *filebase*, fieldstate *s* = Physical) const

References a_, b_, Vector::data_, Vector::N_, REAL_DIGITS, REAL_IOWIDTH, and state_.

```
517                                                 {
518   fieldstate origstate = state_;
519   ((ChebyCoeff&) *this).makeState(savestate);
520
521   string filename(filebase);
522   filename += string(".asc");
523   ofstream os(filename.c_str());
524   os << scientific << setprecision(REAL_DIGITS);
525   os << "% " << N_ << ' ' << a_ << ' ' << b_ << ' ' << state_ << '\n';
526   for (int i=0; i<N_; ++i)
527     os << setw(REAL_IOWIDTH) << data_[i] << '\n';
528   os.close();
529   ((ChebyCoeff&) *this).makeState(origstate);
530 }
```

7.6.5.34 void ChebyCoeff::setBounds (Real *a*, Real *b*)

References a_, and b_.

Referenced by ComplexChebyCoeff::ComplexChebyCoeff(), diff(), eval(), integrate(), ComplexChebyCoeff::setBounds(), ubcFix(), and vbcFix().

```
423                                 {
424   a_ = a;
425   b_ = b;
426   assert(b_ > a_);
427 }
```

Here is the caller graph for this function:

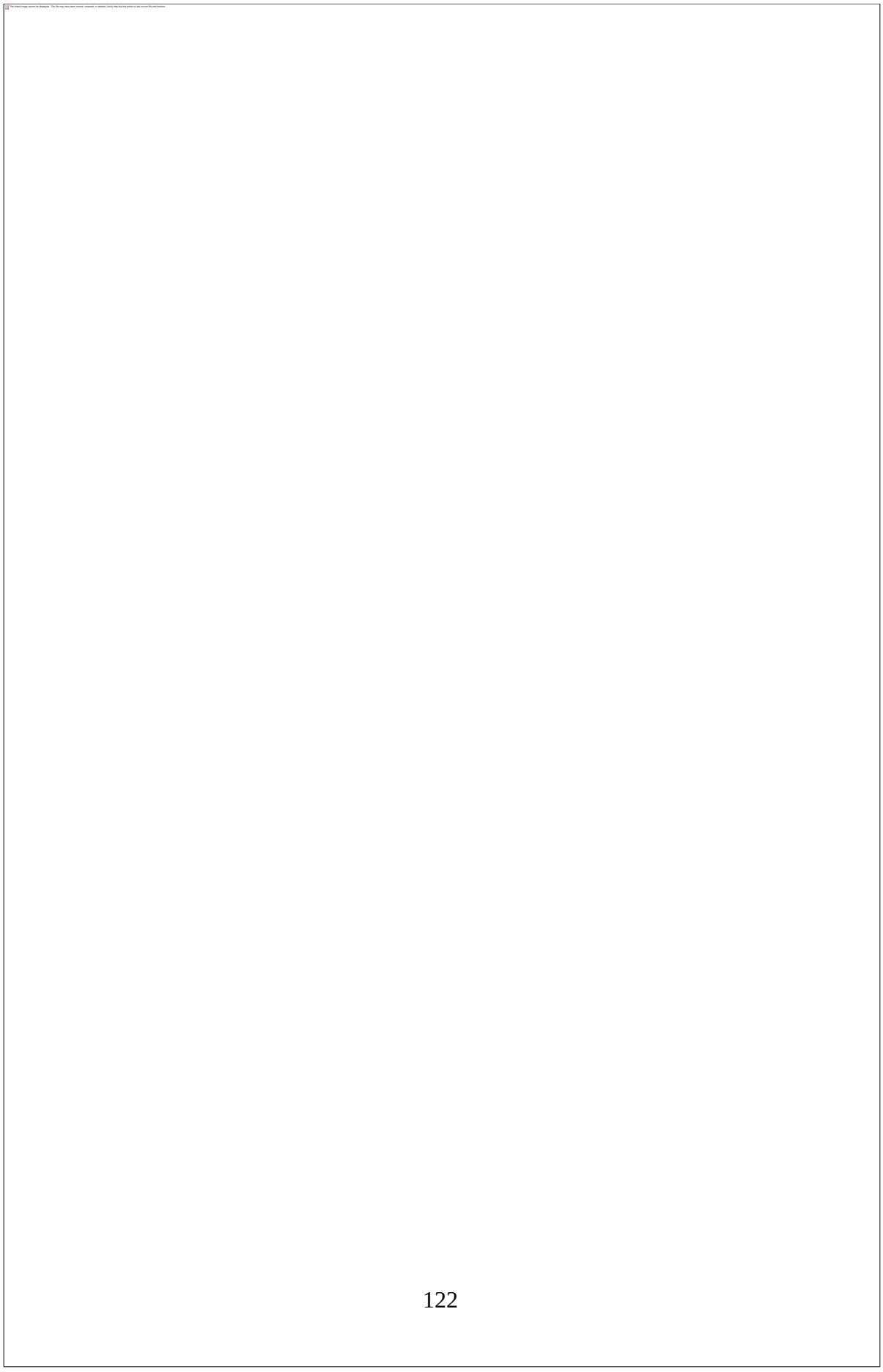
7.6.5.35 void ChebyCoeff::setState (fieldstate *s*)

References Physical, Spectral, and state_.

Referenced by ComplexChebyCoeff::ComplexChebyCoeff(), diff(), DNSAlgorithm::DNSAlgorithm(), eval(), integrate(), makeroll(), ComplexChebyCoeff::setState(), and HelmholtzSolver::solve().

```
429                                {
430   assert(s == Physical || s == Spectral);
431   state_=s;
432 }
```

Here is the caller graph for this function:


The mixed image named No Mystery. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.6.5.36 void ChebyCoeff::setToZero ()

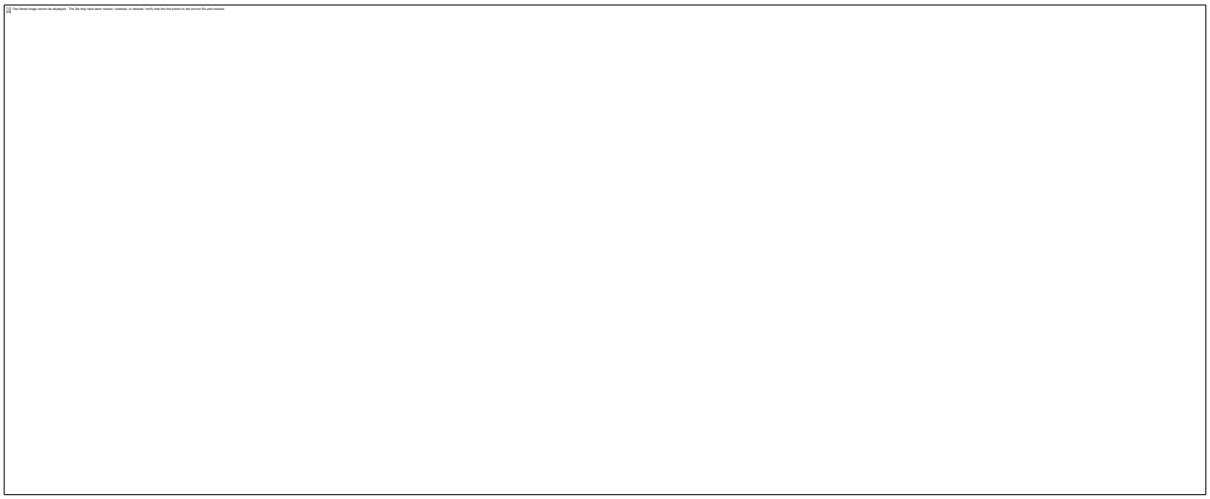
Reimplemented from Vector.

References Vector::data_, and Vector::N_.

Referenced by chebyProfile(), makeroll(), randomProfile(), TurbStats::reset(), ComplexChebyCoeff::setToZero(), and HelmholtzSolver::solve().

```
374                         {
375   for (int i=0; i<N_; ++i)
376     data_[i] = 0.0;
377 }
```

Here is the caller graph for this function:



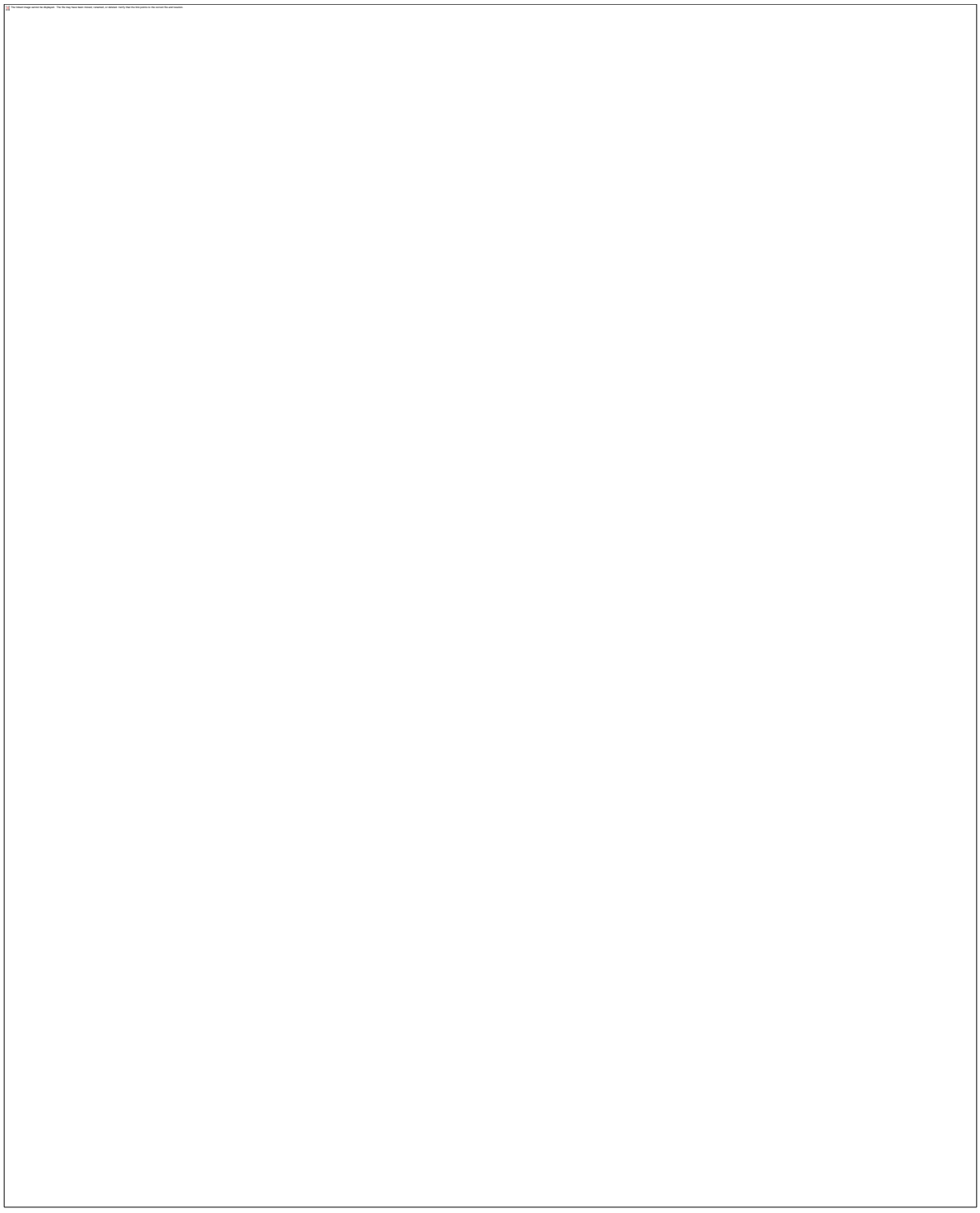
7.6.5.37 fieldstate ChebyCoeff::state () const [inline]

References state_.

Referenced by FlowField::addProfile(), FlowField::CFLfactor(), changeBaseFlow(), chebyDist2(), chebyInnerProduct(), chebyNorm2(), convectionNL(), diff(), divergenceNL(), energy(), integrate(), L1Dist(), L2InnerProduct(), L2Norm2(), linearizedNL(), LinfDist(), LinfNorm(), ComplexChebyCoeff::operator*=(), FlowField::operator+=(), FlowField::operator-=(), rotationalNL(), ComplexChebyCoeff::save(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), HelmholtzSolver::solve(), ComplexChebyCoeff::state(), and TurbStats::TurbStats().

```
346 {return state_;}
```

Here is the caller graph for this function:



7.6.6 Friends And Related Function Documentation

7.6.6.1 friend class ChebyTransform [friend]

7.6.6.2 void swap (ChebyCoeff & *f*, ChebyCoeff & *g*) [friend]

Reimplemented from Vector.

7.6.7 Member Data Documentation

7.6.7.1 Real ChebyCoeff::a_ [private]

Referenced by a(), binaryDump(), binaryLoad(), ChebyCoeff(), congruent(), eval(), interpolate(), L(), reflect(), save(), setBounds(), and swap().

7.6.7.2 Real ChebyCoeff::b_ [private]

Referenced by b(), binaryDump(), binaryLoad(), ChebyCoeff(), congruent(), eval(), interpolate(), L(), reflect(), save(), setBounds(), and swap().

7.6.7.3 fieldstate ChebyCoeff::state_ [private]

Referenced by binaryDump(), binaryLoad(), ChebyCoeff(), chebyfft(), congruent(), eval(), eval_a(), eval_b(), fill(), ichebyfft(), interpolate(), makePhysical(), makeSpectral(), makeState(), mean(), operator*=(), randomize(), reflect(), save(), setState(), state(), and swap().

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

- _cuda/channelflow-0.9.22/channelflow/chebyshev.h
- /_cuda/channelflow-0.9.22/channelflow/chebyshev.cpp

7.6.7.5

7.7 ChebyTransform Class Reference

#include <chebyshev.h>

7.7.1 Public Member Functions

- ChebyTransform (int N, int fftw_flags=FFTW_ESTIMATE)
- ChebyTransform (const ChebyTransform &t)
- ChebyTransform & operator= (const ChebyTransform &t)
- ~ChebyTransform ()
- int N () const
- int length () const

7.7.2 Private Attributes

- int N_
- int flags_
- fftw_plan cosfftw_plan_

7.7.3 Friends

- class ChebyCoeff
- class ComplexChebyCoeff

7.7.4 Constructor & Destructor Documentation

7.7.4.1 ChebyTransform::ChebyTransform (int *N*, int *fftw_flags* = FFTW_ESTIMATE)

References cosfftw_plan_, flags_, and N_.

```
1307   : N_(N),
1308     flags_(flags | FFTW_DESTROY_INPUT),
1309     cosfftw_plan_(0)
1310 {
1311   assert(N_ >0);
1312
1313   // Build an FFTW plan that can be applied to a number of different arrays,
1314   // using the FFTW guru interface. See FFTW guru documentation.
1315   Real* tmp = (Real*) fftw_malloc(N_*sizeof(Real));
1316   fftw_r2r_kind fftw_kind = FFTW_REDFT00;
1317   cosfftw_plan_  = fftw_plan_r2r_1d(N,tmp,tmp,fftw_kind,flags_);
1318   fftw_free(tmp);
1319 }
```

7.7.4.2 ChebyTransform::ChebyTransform (const ChebyTransform & *t*)

References cosfftw_plan_, flags_, and N_.

```
1322   :
1323   N_(),
1324   flags_(t.flags_ | FFTW_DESTROY_INPUT),
1325   cosfftw_plan_(0)
1326 {
1327   assert(N_ >0);
1328
1329   // Build an FFTW plan that can be applied to a number of different arrays,
1330   // using the FFTW guru interface. See FFTW guru documentation.
1331   Real* tmp = (Real*) fftw_malloc(N_*sizeof(Real));
1332   fftw_r2r_kind fftw_kind = FFTW_REDFT00;
1333   cosfftw_plan_ = fftw_plan_r2r_1d(N_, tmp, tmp, fftw_kind, flags_);
1334   fftw_free(tmp);
1335 }
```

7.7.4.3 ChebyTransform::~ChebyTransform ()

References cosfftw_plan_.

```
1357                              {
1358   if (cosfftw_plan_)
1359     fftw_destroy_plan(cosfftw_plan_);
1360 }
```

7.7.5 Member Function Documentation

7.7.5.1 int ChebyTransform::length () const [inline]

References N_.

```
379 { return N_;}
```

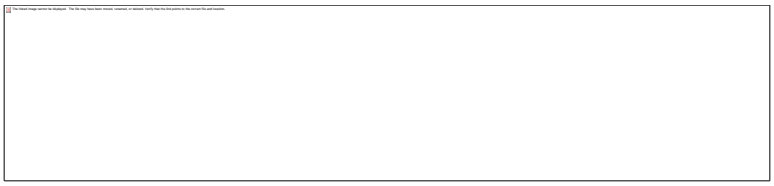
7.7.5.2 int ChebyTransform::N () const [inline]

References N_.

Referenced by ChebyCoeff::chebyfft(), and ChebyCoeff::ichebyfft().

```
378 { return N_;}
```

Here is the caller graph for this function:



7.7.5.3 ChebyTransform & ChebyTransform::operator= (const ChebyTransform & *t*)

References cosfftw_plan_, flags_, and N_.

```
1337                                                {
1338   if (&t == this)
1339     return *this;
1340
1341   if (cosfftw_plan_)
1342     fftw_destroy_plan(cosfftw_plan_);
1343
1344   N_ = t.N_;
1345   flags_ = t.flags_;
1346
1347   assert(N_ >0);
1348
1349   Real* tmp = (Real*) fftw_malloc(N_*sizeof(Real));
1350   fftw_r2r_kind fftw_kind = FFTW_REDFT00;
1351   cosfftw_plan_ = fftw_plan_r2r_1d(N_, tmp, tmp, fftw_kind,flags_);
1352   fftw_free(tmp);
1353
1354   return *this;
1355 }
```

7.7.6 Friends And Related Function Documentation

7.7.6.1 friend class ChebyCoeff [friend]

7.7.6.2 friend class ComplexChebyCoeff [friend]

7.7.7 Member Data Documentation

7.7.7.1 fftw_plan ChebyTransform::cosfftw_plan_ [private]

Referenced by ChebyCoeff::chebyfft(), ChebyTransform(), ChebyCoeff::ichebyfft(), operator=(), and ~ChebyTransform().

7.7.7.2 int ChebyTransform::flags_ [private]

Referenced by ChebyTransform(), and operator=().

7.7.7.3 int ChebyTransform::N_ [private]

Referenced by ChebyTransform(), length(), N(), and operator=().

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

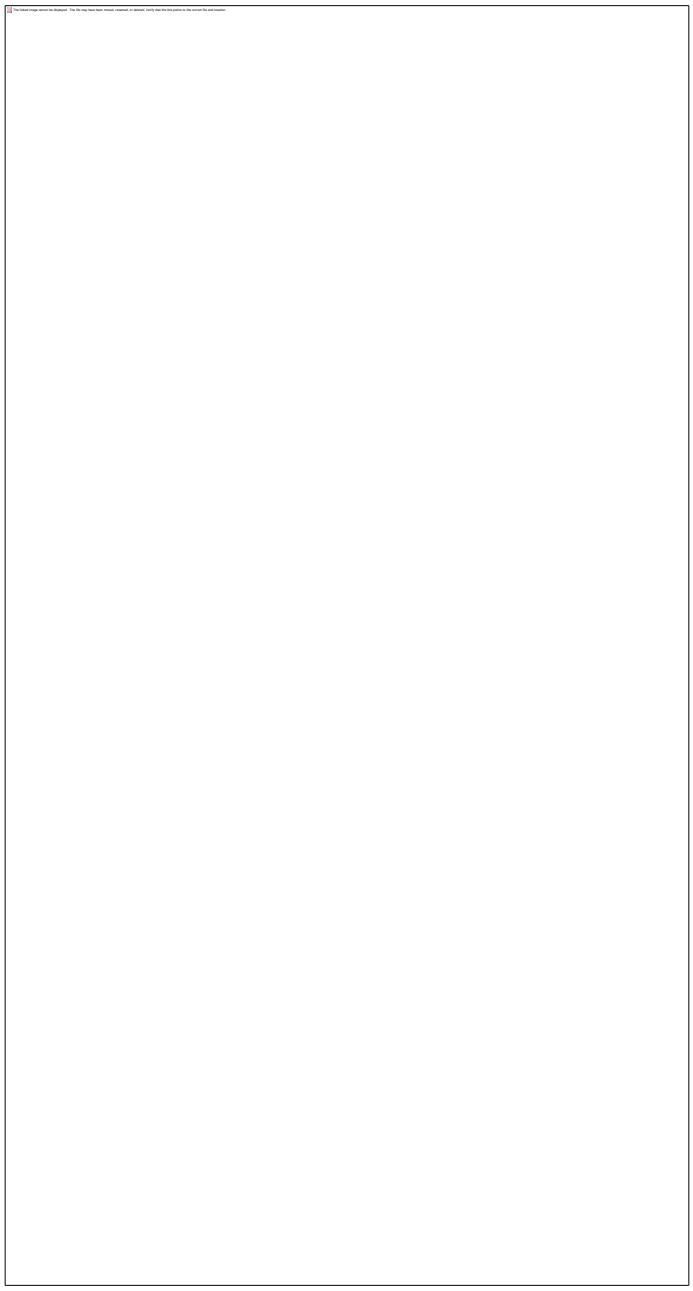
- /_cuda/channelflow-0.9.22/channelflow/chebyshev.h
- /_cuda/channelflow-0.9.22/channelflow/chebyshev.cpp

7.7.7.5

7.8 cmndline Class Reference

#include <dns.h>

Collaboration diagram for cmndline:



7.8.1 Public Member Functions

- void parse (int _argc, char **_argv)

- void show ()
- HowToRun getRun ()
- void nThreads (int k)
- int nThreads ()

7.8.2 Static Public Member Functions

- static cmndline * GetInstance ()

7.8.3 Private Member Functions

- cmndline ()

7.8.4 Private Attributes

- vector< string > m_args
- HowToRun m_run
- int m_nmbThreads

7.8.5 Static Private Attributes

- static cmndline * pInstance = NULL

7.8.6 Constructor & Destructor Documentation

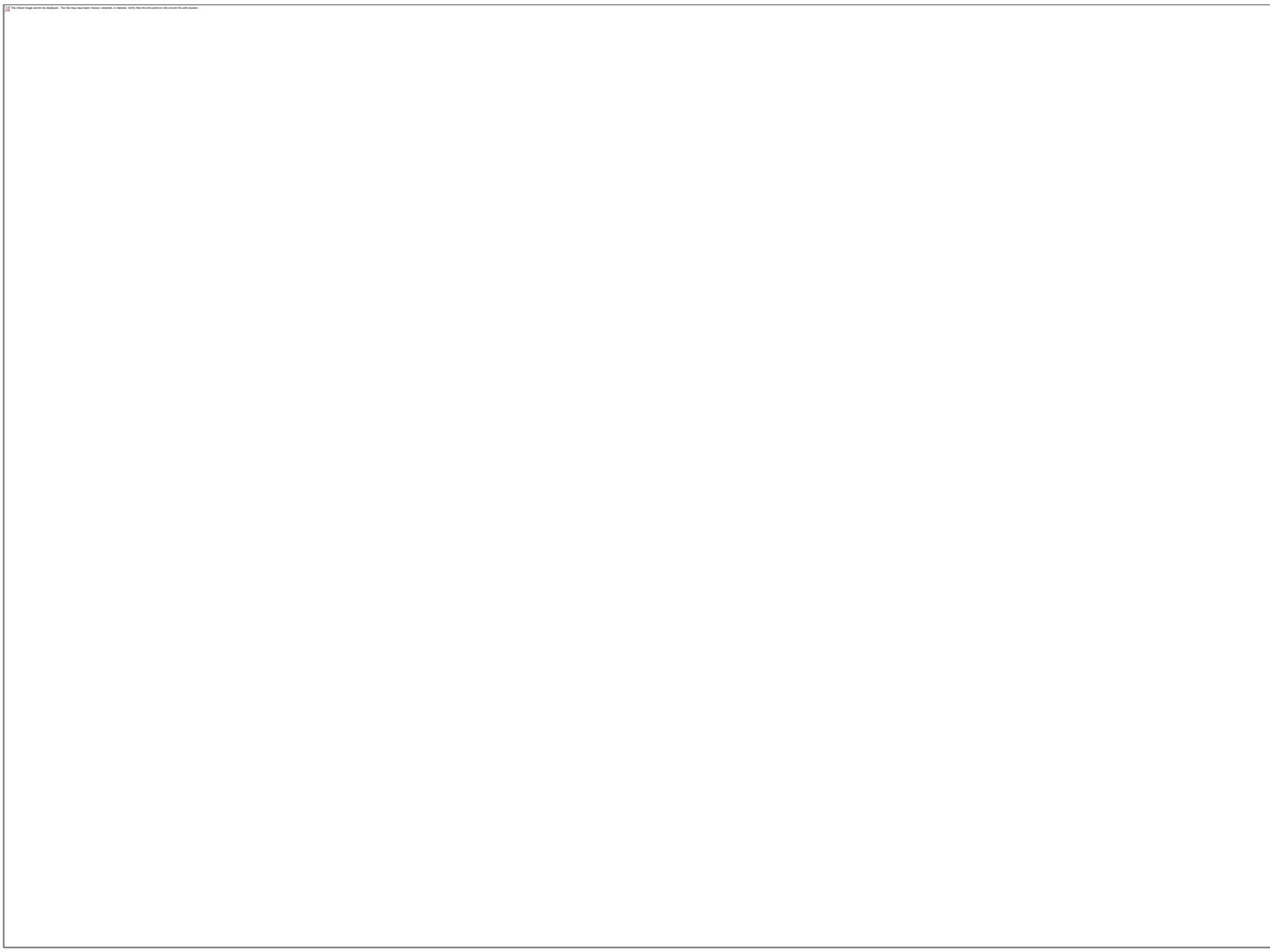
7.8.6.1 cmndline::cmndline () [private]

References m_nmbThreads.

Referenced by GetInstance().

```
37 {
38     m_nmbThreads = 0;
39 }
```

Here is the caller graph for this function:



7.8.7 Member Function Documentation

7.8.7.1 cmndline * cmndline::GetInstance () [static]

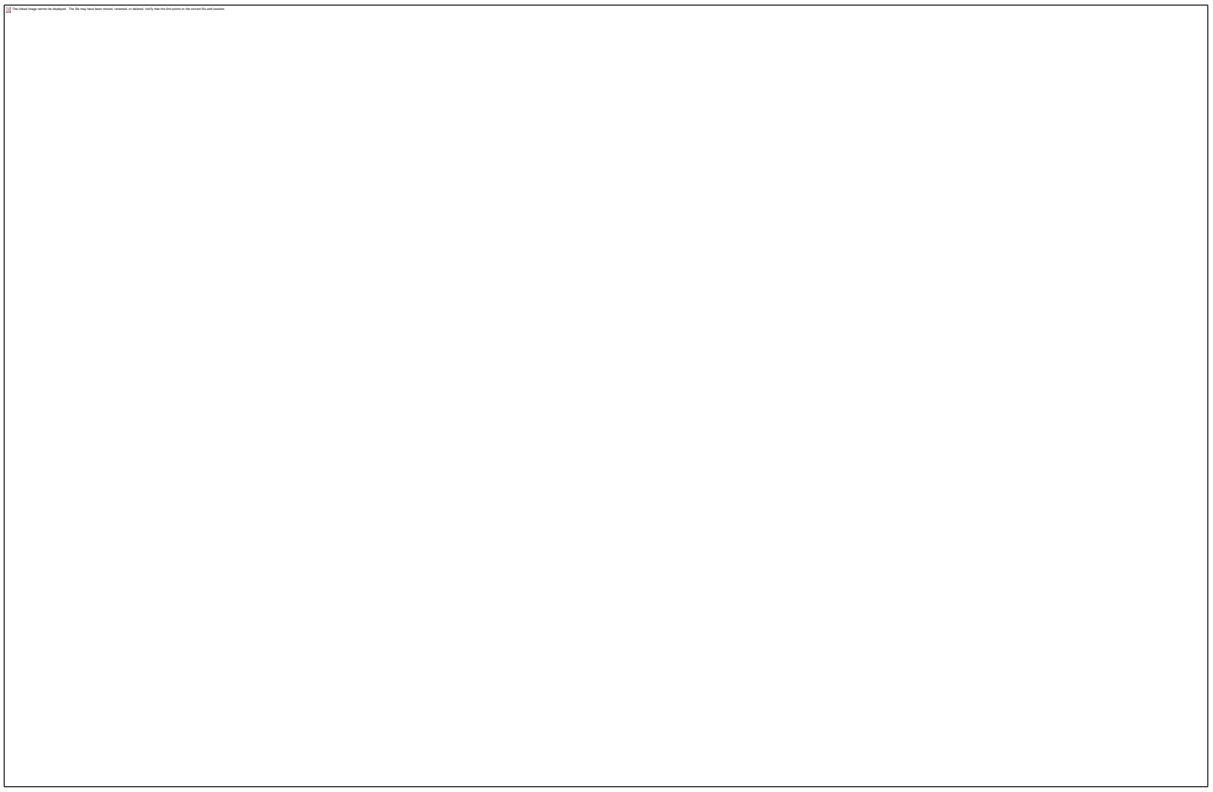
References cmndline(), and pInstance.

Referenced by navierstokesNL().

```
42 {
43         if (pInstance== NULL)
44         {
45                 pInstance = new cmndline();
46         }
47                 return pInstance;
48 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



7.8.7.2 HowToRun cmndline::getRun () [inline]

References m_run.

Referenced by navierstokesNL().

```
77 {return m_run;}
```

Here is the caller graph for this function:

7.8.7.3 int cmndline::nThreads () [inline]

References m_nmbThreads.
Referenced by parse().

```
79 { return m_nmbThreads; }
```

Here is the caller graph for this function:

7.8.7.4 void cmndline::nThreads (int *k*) [inline]

References m_nmbThreads.

```
78 { m_nmbThreads = k; }
```

7.8.7.5 void cmndline::parse (int *_argc*, char ** *_argv*)

References CPU, GPU, m_args, m_nmbThreads, m_run, nThreads(), and THREAD.

```
51 {
52
53       for(int k =0 ; k < _argc; k++)
54       {
55          m_args.push_back(_argv[k]);
56          //m_args.push_back(str);
57       }
58
59       m_run = CPU;
60       if(m_args.size() > 2)
61       {
62             std::vector<string>::const_iterator it;
63             it = find (m_args.begin(), m_args.end(), "-run");
64             if(it !=  m_args.end())
65             {
66
67                   it++;
68                   if(*it == "cpu" ) m_run = CPU;
69                   if(*it == "gpu" ) m_run = GPU;
70                   if(*it == "thread" || *it == "t")
71                   {
72                     m_run = THREAD;
73                     it++;
74                     if(it !=  m_args.end())
75                     {
76                       string tmp = *it;
77                       int nThreads = atoi(tmp.c_str());
78                       m_nmbThreads = nThreads;
79                       cout << "argv ="<< nThreads << endl;
80                     }
81
82                   }
83
84
85             }
86       }
87 }
```

Here is the call graph for this function:

7.8.7.6 void cmndline::show ()

References m_args.

```
90 {
91  std::vector<string>::const_iterator i;
92    for(i=m_args.begin(); i!=m_args.end(); ++i){
93        std::cout<<(*i)<<std::endl;
94     }
95 }
```

7.8.8 Member Data Documentation

7.8.8.1 vector<string> cmndline::m_args [private]

Referenced by parse(), and show().

7.8.8.2 int cmndline::m_nmbThreads [private]

Referenced by cmndline(), nThreads(), and parse().

7.8.8.3 HowToRun cmndline::m_run [private]

Referenced by getRun(), and parse().

7.8.8.4 cmndline * cmndline::pInstance = NULL [static, private]

Referenced by GetInstance().

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

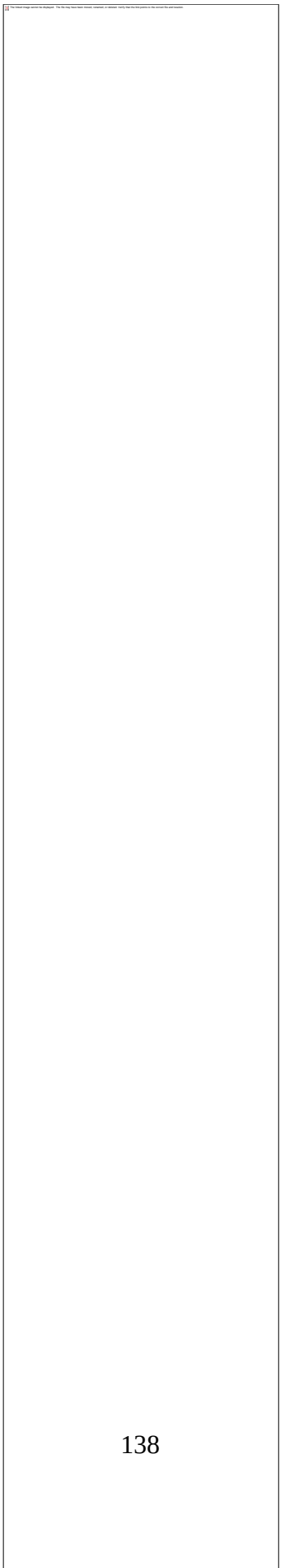
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.8.8.6

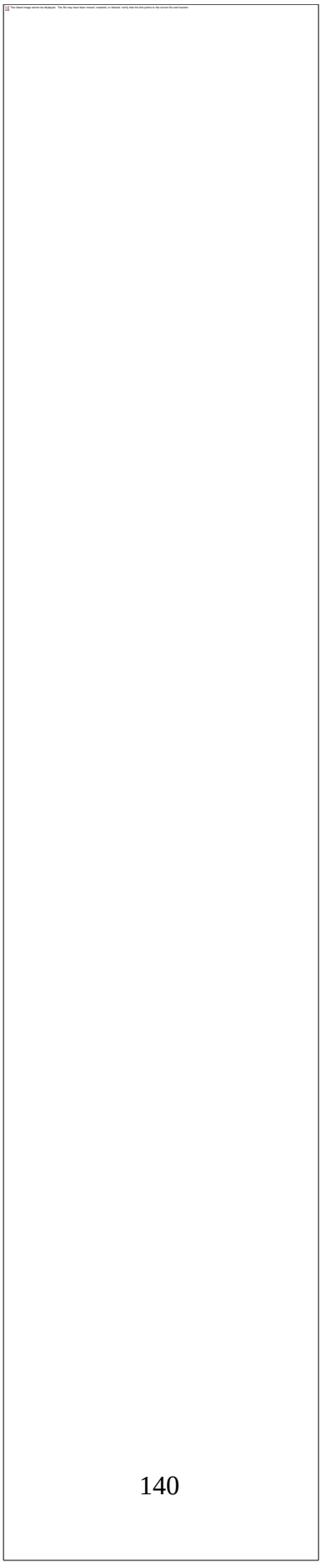
7.9 CNABstyleDNS Class Reference

#include <dns.h>

Inheritance diagram for CNABstyleDNS:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

Collaboration diagram for CNABstyleDNS:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.9.1 Public Member Functions

- CNABstyleDNS ()
- CNABstyleDNS (const CNABstyleDNS &dns)
- CNABstyleDNS (FlowField &u, const ChebyCoeff &Ubase, Real nu, Real dt, const DNSFlags &flags, Real t=0)
- ~CNABstyleDNS ()
- CNABstyleDNS & operator= (const CNABstyleDNS &dns)
- virtual void advance (FlowField &u, FlowField &q, int nSteps=1)
- virtual void reset_dt (Real dt)
- virtual bool push (const FlowField &u)
- virtual bool full () const

7.9.2 Private Member Functions

- virtual DNSAlgorithm * clone () const

7.9.3 Private Attributes

- int Nsubsteps_
- bool full_
- FlowField fj1_
- FlowField fj_
- array< Real > alpha_
- array< Real > beta_
- array< Real > gamma_
- array< Real > zeta_
- TauSolver *** tausolver_

7.9.4 Constructor & Destructor Documentation

7.9.4.1 CNABstyleDNS::CNABstyleDNS ()

Referenced by clone().

```
1716   :
1717   DNSAlgorithm(),
1718   full_(false),
1719   fj1_(),
1720   fj_()
1721 {}
```

Here is the caller graph for this function:

7.9.4.2 CNABstyleDNS::CNABstyleDNS (const CNABstyleDNS & *dns*)

References DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, Nsubsteps_, and tausolver_.

```
1724  :
1725  DNSAlgorithm(dns),
1726  Nsubsteps_(dns.Nsubsteps_),
1727  full_(dns.full_),
1728  fj1_(dns.fj1_),
1729  fj_(dns.fj_),
1730  alpha_(dns.alpha_),
1731  beta_(dns.beta_),
1732  gamma_(dns.gamma_),
1733  zeta_(dns.zeta_)
1734 {
1735  // Allocate memory for [Nsubsteps x Mx_ x Mz_] Tausolver arrays
1736  // and copy tausolvers from dns argument
1737  tausolver_ = new TauSolver**[Nsubsteps_];    // new #1
1738  for (int j=0; j<Nsubsteps_; ++j) {
1739    tausolver_[j] = new TauSolver*[Mx_];       // new #2
1740    for (int mx=0; mx<Mx_; ++mx) {
1741      tausolver_[j][mx] = new TauSolver[Mz_];  // new #3
1742      for (int mz=0; mz<Mz_; ++mz)
1743        tausolver_[j][mx][mz] = dns.tausolver_[j][mx][mz];
1744
1745    }
1746  }
1747 }
```

7.9.4.3 CNABstyleDNS::CNABstyleDNS (FlowField & *u*, const ChebyCoeff & *Ubase*, Real *nu*, Real *dt*, const DNSFlags & *flags*, Real *t* = 0)

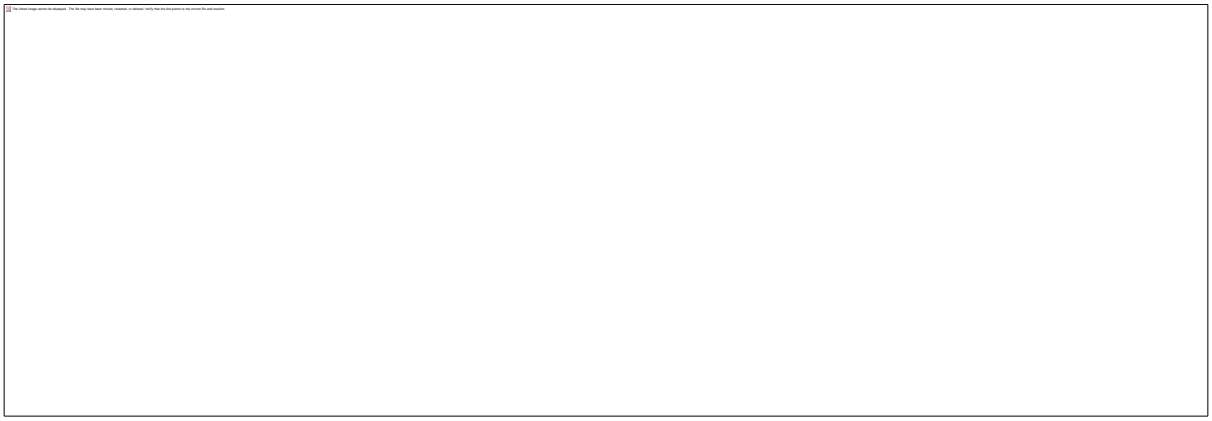
References alpha_, beta_, CNAB2, DNSAlgorithm::dt_, fj1_, fj_, full_, gamma_, DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, DNSAlgorithm::Ninitsteps_, Nsubsteps_, DNSAlgorithm::order_, reset_dt(), array< T >::resize(), FlowField::setToZero(), SMRK2, tausolver_, DNSFlags::timestepping, and zeta_.

```
1751  :
1752  DNSAlgorithm(u,Ubase,nu,dt,flags,t),
1753  full_(false),
1754  fj1_(u),
```

```
1755  fj_(u)
1756 {
1757  fj1_.setToZero();
1758  fj_.setToZero();
1759
1760  TimeStepMethod algorithm = flags.timestepping;
1761  switch (algorithm) {
1762  case CNAB2:
1763   order_ = 2;
1764   Nsubsteps_ = 1;
1765   Ninitsteps_ = 1;
1766   full_ = false;
1767   alpha_.resize(Nsubsteps_);
1768   beta_.resize(Nsubsteps_);
1769   gamma_.resize(Nsubsteps_);
1770   zeta_.resize(Nsubsteps_);
1771   alpha_[0] = 0.5;
1772   beta_[0]  = 0.5;
1773   gamma_[0] = 1.5;
1774   zeta_[0]  = -0.5;
1775   break;
1776  case SMRK2:
1777   order_ = 2;
1778   Nsubsteps_ = 3;
1779   Ninitsteps_ = 0;
1780   full_ = true;
1781   alpha_.resize(Nsubsteps_);
1782   beta_.resize(Nsubsteps_);
1783   gamma_.resize(Nsubsteps_);
1784   zeta_.resize(Nsubsteps_);
1785   alpha_[0] = 29.0/96.0;  alpha_[1] = -3.0/40.0;  alpha_[2] = 1.0/6.0;
1786   beta_[0]  = 37.0/160.0; beta_[1] =  5.0/24.0;  beta_[2]  = 1.0/6.0;
1787   gamma_[0] = 8.0/15.0;   gamma_[1] =  5.0/12.0;  gamma_[2] = 3.0/4.0;
1788   zeta_[0]  = 0.0;        zeta_[1] = -17.0/60.0;  zeta_[2] = -5.0/12.0;
1789   break;
1790  default:
1791   cerr << "CNABstyleDNS::CNABstyleDNS(un,Ubase,nu,dt,flags,t0)\n"
1792        << "error: flags.timestepping == " << algorithm
1793        << " is not a CNAB-style algorithm." << endl;
1794   exit(1);
1795  }
1796
1797  // Allocate memory for [Nsubsteps x Mx_ x Mz_] Tausolver array
1798  tausolver_ = new TauSolver**[Nsubsteps_];    // new #1
1799  for (int j=0; j<Nsubsteps_; ++j) {
1800   tausolver_[j] = new TauSolver*[Mx_];       // new #2
```

```
1801     for (int mx=0; mx<Mx_; ++mx)
1802       tausolver_[j][mx] = new TauSolver[Mz_];  // new #3
1803   }
1804   reset_dt(dt_);
1805 }
```

Here is the call graph for this function:



7.9.4.4 CNABstyleDNS::~CNABstyleDNS ()

References DNSAlgorithm::Mx_, Nsubsteps_, and tausolver_.

```
1807                          {
1808   if (tausolver_) {
1809     for (int j=0; j<Nsubsteps_; ++j) {
1810       for (int mx=0; mx<Mx_; ++mx)
1811         delete[] tausolver_[j][mx];  // undo new #3
1812       delete[] tausolver_[j];        // undo new #2
1813     }
1814     delete[] tausolver_;             // undo new #1
1815   }
1816   tausolver_ = 0;
1817 }
```

7.9.5 Member Function Documentation

7.9.5.1 void CNABstyleDNS::advance (FlowField & *u*, FlowField & *q*, int *nSteps* = 1) [virtual]

Implements DNSAlgorithm.

References ComplexChebyCoeff::add(), alpha_, beta_, DNSAlgorithm::cfl_, FlowField::CFLfactor(), FlowField::cmplx(), DNSFlags::constraint, DNSFlags::dealias_xz(), diff(), diff2(), DNSAlgorithm::dPdxAct_, DNSAlgorithm::dPdxRef_, DNSAlgorithm::dt_, FlowField::Dx(), FlowField::Dz(), fj1_, fj_, DNSAlgorithm::flags_, gamma_, DNSAlgorithm::isAliasedMode(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), Vector::length(), DNSAlgorithm::Lx_, DNSAlgorithm::Lz_, ChebyCoeff::mean(), DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, navierstokesNL(), DNSFlags::nonlinearity, Nsubsteps_, DNSAlgorithm::nu_, DNSAlgorithm::Nx_, DNSAlgorithm::Ny_, DNSAlgorithm::Nyd_, DNSAlgorithm::Nz_, pi, DNSAlgorithm::Pk_, PressureGradient, PrintAll, PrintTicks, PrintTime, DNSAlgorithm::Pyk_, Re(), ComplexChebyCoeff::re, DNSAlgorithm::Rxk_, DNSAlgorithm::Ryk_, DNSAlgorithm::Rzk_, ComplexChebyCoeff::set(), FlowField::setDealiased(), ComplexChebyCoeff::setToZero(), TauSolver::solve(), square(), swap(), DNSAlgorithm::t_, tausolver_, DNSAlgorithm::tmp_, DNSAlgorithm::Ubase_, DNSAlgorithm::Ubaseyy_, DNSAlgorithm::UbulkAct_, DNSAlgorithm::UbulkBase_, DNSAlgorithm::UbulkRef_, DNSAlgorithm::uk_, DNSFlags::verbosity, DNSAlgorithm::vk_, DNSAlgorithm::wk_, and zeta_.

```
1874                                                {
1875   const int kxmax = tmp_.kxmax();
1876   const int kzmax = tmp_.kzmax();
1877
1878   for (int n=0; n<Nsteps; ++n) {
1879     for (int j=0; j<Nsubsteps_; ++j) {
1880
1881       // Store substeps uj,qj in un,qn; reflect this in notation
1882       FlowField& uj(un);
1883       FlowField& qj(qn);
1884
1885       swap(fj_, fj1_);
1886        //cout << "IG call navierstokesNL 6" <<endl;
1887
1888       navierstokesNL(un, Ubase_, fj_, tmp_, flags_.nonlinearity);
1889
1890       // Set convenience variables
1891       Real a_b = alpha_[j]/beta_[j];
1892       Real g_b = gamma_[j]/beta_[j];
1893       Real z_b = zeta_[j]/beta_[j];
1894       Real anu_b = a_b*nu_;
1895       Real anu = alpha_[j]*nu_;
1896
1897       // Update each Fourier mode with time-stepping algorithm
1898       for (int mx=0; mx<Mx_; ++mx) {
1899         const int kx = uj.kx(mx);
1900
1901         for (int mz=0; mz<Mz_; ++mz) {
1902           const int kz = uj.kz(mz);
```

```
1903
1904        // Zero out the aliased modes
1905        if ((kx == kxmax || kz == kzmax) ||
1906          flags_.dealias_xz() && isAliasedMode(kx,kz)) {
1907          for (int ny=0; ny<Nyd_; ++ny) {
1908            uj.cmplx(mx,ny,mz,0) = 0.0;
1909            uj.cmplx(mx,ny,mz,1) = 0.0;
1910            uj.cmplx(mx,ny,mz,2) = 0.0;
1911            qj.cmplx(mx,ny,mz,0) = 0.0;
1912          }
1913          break;
1914        }
1915
1916        Rxk_.setToZero();
1917        Ryk_.setToZero();
1918        Rzk_.setToZero();
1919
1920        // Goal is to compute
1921        // R = a/b nu uj" + [1/(b dt)- a/b nu kappa2] uj - a/b grad qj
1922        //     - g/b fj - z/b fj1
1923        //
1924        //   = a/b nu uj" + [1/(nu a dt)- kappa2] a/b nu uj
1925        //     - a/b grad qj - g/b fj - z/b fj1
1926
1927        // Extract relevant Fourier modes of uj and qj for computations
1928        // set (uk,vk,wk) to a/b nu uj, Pk to a/b qj
1929
1930        for (int ny=0; ny<Nyd_; ++ny) {
1931          uk_.set(ny, anu_b*uj.cmplx(mx,ny,mz,0));
1932          vk_.set(ny, anu_b*uj.cmplx(mx,ny,mz,1));
1933          wk_.set(ny, anu_b*uj.cmplx(mx,ny,mz,2));
1934          Pk_.set(ny, a_b*qj.cmplx(mx,ny,mz,0));
1935        }
1936
1937        // (1) Put a/b nu uj" into in R. (Pyk_ is used as tmp workspace)
1938        diff2(uk_, Rxk_, Pyk_);
1939        diff2(vk_, Ryk_, Pyk_);
1940        diff2(wk_, Rzk_, Pyk_);
1941
1942        // (2) Put a/b qj' into Pyk (compute y-comp of pressure gradient).
1943        diff(Pk_, Pyk_);
1944
1945        // (3) Add [1/(nu a dt)- kappa2] a/b nu uj - a/b grad qj
1946        //         - g/b fj - z/b fj1 to R, completing calculation of R
1947        const Real kappa2 = 4*pi*pi*(square(kx/Lx_) + square(kz/Lz_));
1948        const Real c = 1.0/(anu*dt_) - kappa2;
```

```
1949        const Complex Dx = un.Dx(mx);
1950        const Complex Dz = un.Dz(mz);
1951        for (int ny=0; ny<Nyd_; ++ny) {
1952          Rxk_.add(ny, c*uk_[ny] - Dx*Pk_[ny]
1953                - g_b*fj_.cmplx(mx,ny,mz,0) - z_b*fj1_.cmplx(mx,ny,mz,0));
1954          Ryk_.add(ny, c*vk_[ny] - Pyk_[ny]
1955                - g_b*fj_.cmplx(mx,ny,mz,1) - z_b*fj1_.cmplx(mx,ny,mz,1));
1956          Rzk_.add(ny, c*wk_[ny] - Dz*Pk_[ny]
1957                - g_b*fj_.cmplx(mx,ny,mz,2) - z_b*fj1_.cmplx(mx,ny,mz,2));
1958        }
1959
1960        // Solve the tau solutions
1961        if (kx!=0 || kz!=0)
1962          tausolver_[j][mx][mz].solve(uk_,vk_,wk_,Pk_, Rxk_,Ryk_,Rzk_);
1963
1964        else { // kx,kz == 0,0
1965          // Rx has additional terms, nu Uyy at both t=j and t=j+1
1966          Real a_b = alpha_[j]/beta_[j];
1967          Real c = nu_*(a_b+1.0);
1968          if (Ubaseyy_.length() > 0)
1969            for (int ny=0; ny<Ny_; ++ny)
1970              Rxk_.re[ny] += c*Ubaseyy_[ny];
1971
1972          if (flags_.constraint == PressureGradient) {
1973            // dPdx is supplied, put dPdx at both t=j and t=j+1 on RHS
1974            Rxk_.re[0] -= a_b*dPdxAct_ + dPdxRef_;
1975
1976            // Solve the tau equations
1977            tausolver_[j][mx][mz].solve(uk_, vk_, wk_, Pk_, Rxk_, Ryk_, Rzk_);
1978
1979            // Bulk velocity is free variable on LHS solved by tau eqn
1980            UbulkAct_ = UbulkBase_ + uk_.re.mean();
1981            dPdxAct_ = dPdxRef_;
1982          }
1983          else { // const bulk velocity
1984            // Add the previous time-step's -dPdx to the RHS. The next
1985            // timestep's dPdx term appears on LHS as unknown.
1986            Rxk_.re[0] -= a_b*dPdxAct_;
1987
1988            // Use tausolver with additional variable and constraint:
1989            // free variable: dPdxAct at next time-step,
1990            // constraint:    UbulkBase + mean(u) = UbulkRef.
1991            tausolver_[j][mx][mz].solve(uk_, vk_, wk_, Pk_, dPdxAct_,
1992                                        Rxk_, Ryk_, Rzk_,
1993                                        UbulkRef_ - UbulkBase_);
1994            UbulkAct_ = UbulkBase_ + uk_.re.mean(); // should == UbulkRef_
```

```
1995          }
1996          // for kx=kz=0, constant term of pressure is arbitrary 3/19/05
1997          // Pk_.set(0, Complex(0.0, 0.0));
1998        }
1999
2000        // Load solutions back into the external 3d data arrays.
2001        // Because of FFTW complex symmetries
2002        // The 0,0 mode must be real.
2003        // For Nx even, the kxmax,0 mode must be real
2004        // For Nz even, the 0,kzmax mode must be real
2005        // For Nx,Nz even, the kxmax,kzmax mode must be real
2006        if ((kx == 0 && kz == 0) ||
2007            (Nx_%2 == 0 && kx == kxmax && kz == 0) ||
2008            (Nz_%2 == 0 && kz == kzmax && kx == 0) ||
2009            (Nx_%2 == 0 && Nz_%2 == 0 && kx == kxmax && kz == kzmax)) {
2010
2011          for (int ny=0; ny<Nyd_; ++ny) {
2012            un.cmplx(mx,ny,mz,0) = Complex(Re(uk_[ny]), 0.0);
2013            un.cmplx(mx,ny,mz,1) = Complex(Re(vk_[ny]), 0.0);
2014            un.cmplx(mx,ny,mz,2) = Complex(Re(wk_[ny]), 0.0);
2015            qn.cmplx(mx,ny,mz,0) = Complex(Re(Pk_[ny]), 0.0);
2016          }
2017        }
2018        // The normal case, for general kx,kz
2019        else
2020          for (int ny=0; ny<Nyd_; ++ny) {
2021            un.cmplx(mx,ny,mz,0) = uk_[ny];
2022            un.cmplx(mx,ny,mz,1) = vk_[ny];
2023            un.cmplx(mx,ny,mz,2) = wk_[ny];
2024            qn.cmplx(mx,ny,mz,0) = Pk_[ny];
2025          }
2026
2027        // And now set the y-aliased modes to zero.
2028        for (int ny=Nyd_; ny<Ny_; ++ny) {
2029          un.cmplx(mx,ny,mz,0) = 0.0;
2030          un.cmplx(mx,ny,mz,1) = 0.0;
2031          un.cmplx(mx,ny,mz,2) = 0.0;
2032          qn.cmplx(mx,ny,mz,0) = 0.0;
2033        }
2034      }
2035    }
2036  }
2037  t_ += dt_;
2038  if (flags_.verbosity == PrintTime || flags_.verbosity == PrintAll)
2039    cout << t_ << ' ' << flush;
2040  else if (flags_.verbosity == PrintTicks)
```

```
2041       cout << '.' << flush;
2042     }
2043 
2044     cfl_ = un.CFLfactor(Ubase_);
2045     cfl_ *= flags_.dealias_xz() ? 2.0*pi/3.0*dt_ : pi*dt_;
2046 
2047     // If using dealiasing, set flag in FlowField that compactifies binary IO
2048     un.setDealiased(flags_.dealias_xz());
2049     qn.setDealiased(flags_.dealias_xz());
2050 
2051     if (flags_.verbosity == PrintTime || flags_.verbosity == PrintAll)
2052       cout << endl;
2053 
2054     return;
2055 }
```

Here is the call graph for this function:

7.9.5.2 DNSAlgorithm * CNABstyleDNS::clone () const [private, virtual]

Implements DNSAlgorithm.

References CNABstyleDNS().

```
1819                                    {
1820   return new CNABstyleDNS(*this);
1821 }
```

Here is the call graph for this function:

7.9.5.3 bool CNABstyleDNS::full () const [virtual]

Reimplemented from DNSAlgorithm.

References full_.

```
1870                                {
1871   return full_;
1872 }
```

7.9.5.4 CNABstyleDNS& CNABstyleDNS::operator= (const CNABstyleDNS & *dns*)

Reimplemented from DNSAlgorithm.

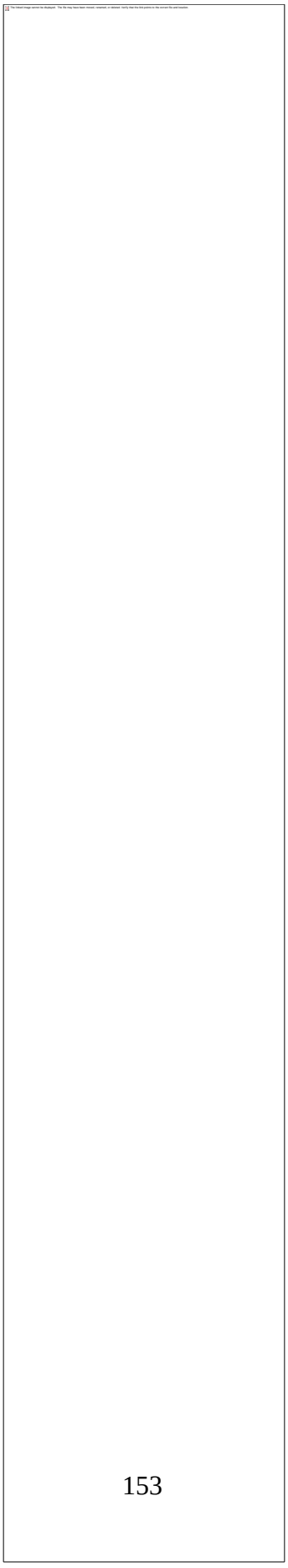
7.9.5.5 bool CNABstyleDNS::push (const FlowField & *u*) [virtual]

Reimplemented from DNSAlgorithm.

References DNSAlgorithm::dt_, fj1_, fj_, DNSAlgorithm::flags_, full_, navierstokesNL(), DNSFlags::nonlinearity, swap(), DNSAlgorithm::t_, DNSAlgorithm::tmp_, and DNSAlgorithm::Ubase_.

```
1861                                    {
1862   swap(fj_, fj1_);
1863    //cout << "IG call navierstokesNL 5" <<endl;
1864   navierstokesNL(un, Ubase_, fj_, tmp_, flags_.nonlinearity);
1865   t_ += dt_;
1866   full_ = true;
1867   return full_;
1868 }
```

Here is the call graph for this function:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.9.5.6 void CNABstyleDNS::reset_dt (Real *dt*) [virtual]

Implements DNSAlgorithm.

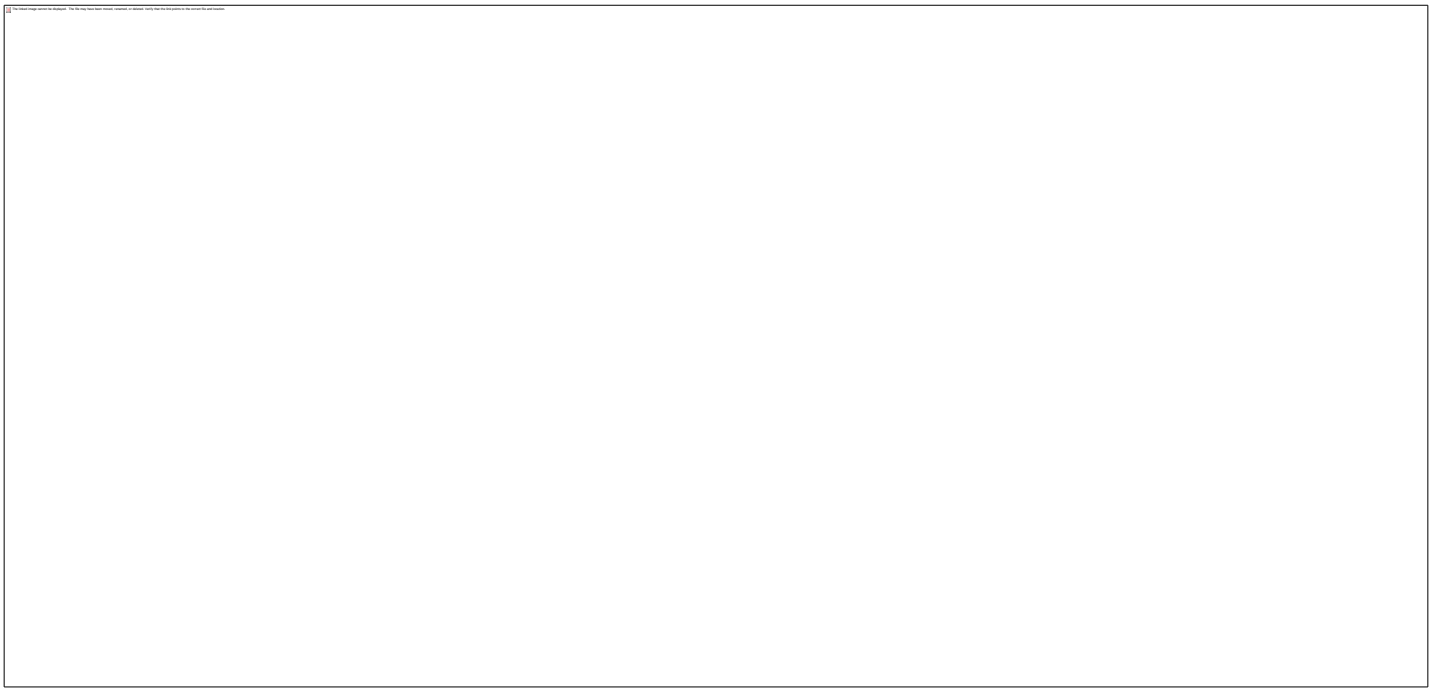
References DNSAlgorithm::a_, DNSAlgorithm::b_, beta_, DNSAlgorithm::cfl_, CNAB2, DNSFlags::dealias_xz(), DNSAlgorithm::dt_, DNSAlgorithm::flags_, full_, DNSAlgorithm::isAliasedMode(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), DNSAlgorithm::Lx_, DNSAlgorithm::Lz_, DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, Nsubsteps_, DNSAlgorithm::nu_, DNSAlgorithm::Nyd_, pi, square(), DNSFlags::taucorrection, tausolver_, DNSFlags::timestepping, and DNSAlgorithm::tmp_.

Referenced by CNABstyleDNS().

```
1824                                {
1825   cfl_ *= dt/dt_;
1826   //nu_ = nu;
1827   dt_ = dt;
1828   const Real c = 4.0*square(pi)*nu_;
1829   const int kxmax = tmp_.kxmax();
1830   const int kzmax = tmp_.kzmax();
1831
1832   // This loop replaces the TauSolver objects at tausolver_[i][mx][mz]
1833   // with new TauSolver objects configured with appropriate parameters
1834   for (int j=0; j<Nsubsteps_; ++j) {
1835     for (int mx=0; mx<Mx_; ++mx) {
1836       int kx = tmp_.kx(mx);
1837       for (int mz=0; mz<Mz_; ++mz) {
1838         int kz = tmp_.kz(mz);
1839
1840         Real lambda = 1.0/(beta_[j]*dt_)+c*(square(kx/Lx_)+square(kz/Lz_));
1841
1842         if ((kx != kxmax) && (kz != kzmax) &&
1843             (!flags_.dealias_xz() || !isAliasedMode(kx,kz)))
1844
1845           tausolver_[j][mx][mz] =
1846             TauSolver(kx, kz, Lx_, Lz_, a_, b_, lambda, nu_, Nyd_,
1847                       flags_.taucorrection);
1848       }
1849     }
1850   }
1851   // For some forms of CNABstyle, need to reinitialize again
1852   switch (flags_.timestepping) {
1853   case CNAB2:
1854     full_ = false;
1855     break;
1856   default:
1857     ;
```

```
1858   }
1859 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.9.6 Member Data Documentation

7.9.6.1 array<Real> CNABstyleDNS::alpha_ [private]

Referenced by advance(), and CNABstyleDNS().

7.9.6.2 array<Real> CNABstyleDNS::beta_ [private]

Referenced by advance(), CNABstyleDNS(), and reset_dt().

7.9.6.3 FlowField CNABstyleDNS::fj1_ [private]

Referenced by advance(), CNABstyleDNS(), and push().

7.9.6.4 FlowField CNABstyleDNS::fj_ [private]

Referenced by advance(), CNABstyleDNS(), and push().

7.9.6.5 bool CNABstyleDNS::full_ [private]

Referenced by CNABstyleDNS(), full(), push(), and reset_dt().

7.9.6.6 array<Real> CNABstyleDNS::gamma_ [private]

Referenced by advance(), and CNABstyleDNS().

7.9.6.7 int CNABstyleDNS::Nsubsteps_ [private]

Referenced by advance(), CNABstyleDNS(), reset_dt(), and ~CNABstyleDNS().

7.9.6.8 TauSolver*** CNABstyleDNS::tausolver_ [private]

Referenced by advance(), CNABstyleDNS(), reset_dt(), and ~CNABstyleDNS().

7.9.6.9 array<Real> CNABstyleDNS::zeta_ [private]

Referenced by advance(), and CNABstyleDNS().

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

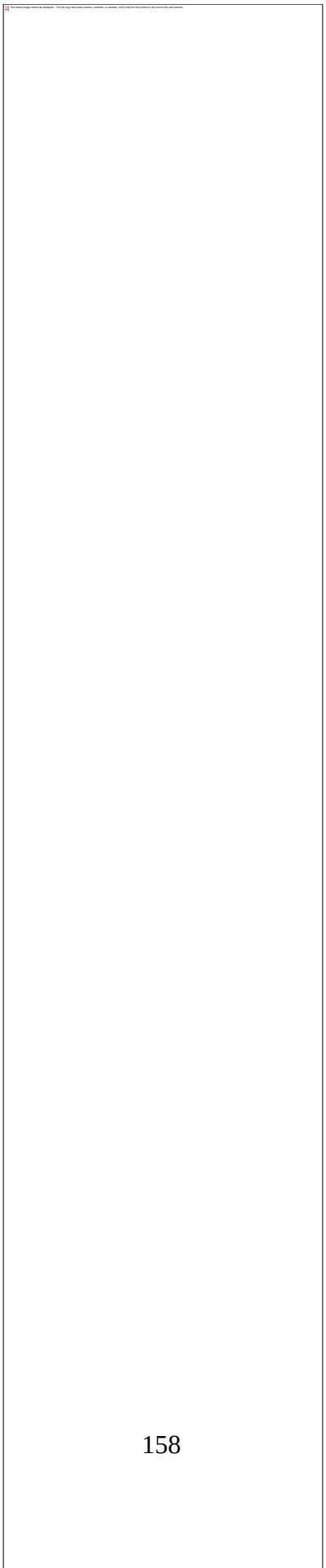
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.9.6.11

7.10 ComplexChebyCoeff Class Reference

#include <chebyshev.h>

Collaboration diagram for ComplexChebyCoeff:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.10.1 Public Member Functions

- ComplexChebyCoeff ()
- ComplexChebyCoeff (int N, Real a, Real b, fieldstate s)
- ComplexChebyCoeff (int N, const ComplexChebyCoeff &f)
- ComplexChebyCoeff (const ChebyCoeff &re, const ChebyCoeff &im)
- ComplexChebyCoeff (const std::string &filename)
- ComplexChebyCoeff (std::istream &is)
- void resize (int N)
- void randomize (Real decay, bool a_dirichlet=false, bool b_dirichlet=false)
- void setToZero ()
- void setBounds (Real a, Real b)
- void setState (fieldstate s)
- void fill (const ComplexChebyCoeff &g)
- void interpolate (const ComplexChebyCoeff &g)
- void reflect (const ComplexChebyCoeff &g, parity p)
- Complex eval_a () const
- Complex eval_b () const
- Complex eval (Real x) const
- Complex mean () const
- Real a () const
- Real b () const
- Real L () const
- int N () const
- int length () const
- int numModes () const
- fieldstate state () const
- Complex operator[] (int n) const
- void set (int n, Complex c)
- void add (int n, Complex c)
- void sub (int n, Complex c)
- ComplexChebyCoeff & operator+= (const ComplexChebyCoeff &f)
- ComplexChebyCoeff & operator-= (const ComplexChebyCoeff &f)
- ComplexChebyCoeff & operator*= (Real c)
- ComplexChebyCoeff & operator*= (Complex c)
- ComplexChebyCoeff & operator*= (const ComplexChebyCoeff &c)
- void conjugate ()
- void save (const std::string &filebase, fieldstate s=Physical) const
- void binaryDump (std::ostream &os) const
- void binaryLoad (std::istream &is)
- bool congruent (const ComplexChebyCoeff &g) const
- void chebyfft ()
- void ichebyfft ()

- void makeSpectral ()
- void makePhysical ()
- void makeState (fieldstate s)
- void chebyfft (const ChebyTransform &t)
- void ichebyfft (const ChebyTransform &t)
- void makeSpectral (const ChebyTransform &t)
- void makePhysical (const ChebyTransform &t)
- void makeState (fieldstate s, const ChebyTransform &t)

7.10.2 Public Attributes

- ChebyCoeff re
- ChebyCoeff im

7.10.3 Friends

- void swap (ComplexChebyCoeff &f, ComplexChebyCoeff &g)

7.10.4 Constructor & Destructor Documentation

7.10.4.1 ComplexChebyCoeff::ComplexChebyCoeff ()

```
978   :
979   re(),
980   im()
981 {}
```

7.10.4.2 ComplexChebyCoeff::ComplexChebyCoeff (int *N*, Real *a*, Real *b*, fieldstate *s*)

```
984   :
985   re(N,a,b,s),
986   im(N,a,b,s)
987 {}
ComplexChebyCoeff::ComplexChebyCoeff(int N, const ComplexChebyCoeff& u)
```

7.10.4.3 ComplexChebyCoeff::ComplexChebyCoeff (int *N*, const ComplexChebyCoeff & *f*)

```
989   :
990   re(N, u.re),
991   im(N, u.im)
992 {}
```

7.10.4.4 ComplexChebyCoeff::ComplexChebyCoeff (const ChebyCoeff & *re*, const ChebyCoeff & *im*)

```
995   :
996   re(r),
997   im(i)
998 {}
```

7.10.4.5 ComplexChebyCoeff::ComplexChebyCoeff (const std::string & *filename*)

References a(), b(), im, N(), re, Vector::resize(), ChebyCoeff::setBounds(), and ChebyCoeff::setState().

```
1001   :
1002   re(),
1003   im()
1004 {
1005   string filename(filebase);
1006   filename += string(".asc");
1007 
1008   //ifstream sis(sizefile.c_str());
1009   ifstream is(filename.c_str());
1010 
1011   // Read in header. Form is "%N a b s"
1012   char c;
1013   int N;
1014   is >> c;
1015   if (c != '%') {
1016     string message("ComplexChebyCoeff(filebase): bad header in file ");
1017     message += filename;
1018     cerr << message << endl;
1019     assert(false);
1020   }
1021   Real a,b;
1022   fieldstate s;
1023   is >> N >> a >> b >> s;
1024   assert(is.good());
1025 
1026   re.resize(N);
1027   im.resize(N);
1028   re.setBounds(a,b);
1029   im.setBounds(a,b);
1030   re.setState(s);
```

```
1031   im.setState(s);
1032
1033   for (int i=0; i<N; ++i) {
1034     is >> re[i] >> im[i];
1035     assert(is.good());
1036   }
1037   is.close();
1038 }
```

Here is the call graph for this function:

7.10.4.6 ComplexChebyCoeff::ComplexChebyCoeff (std::istream & *is*)

7.10.5 Member Function Documentation

7.10.5.1 Real ComplexChebyCoeff::a () const [inline]

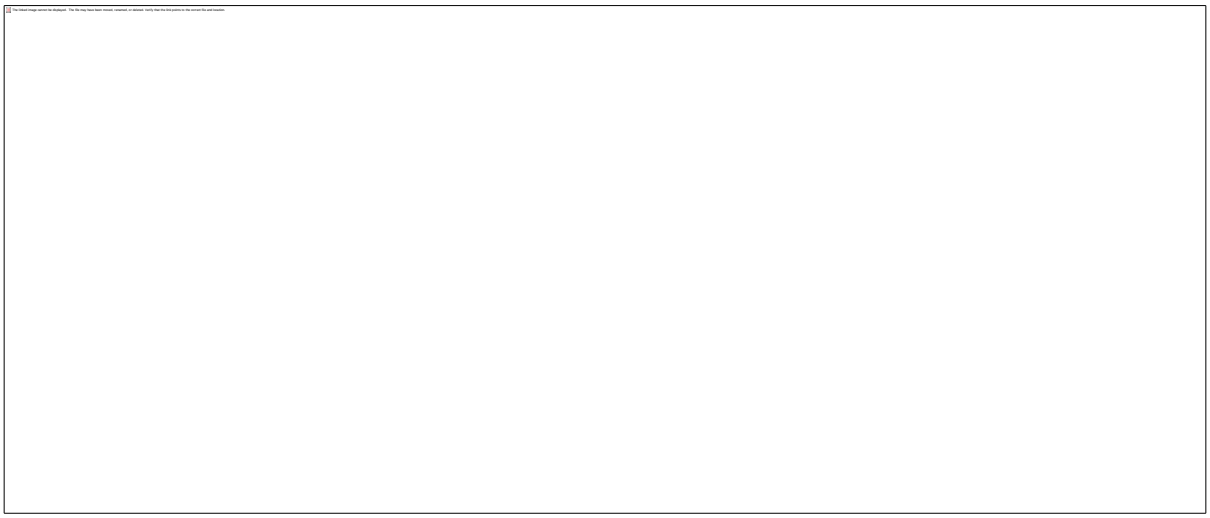
References ChebyCoeff::a(), and re.

Referenced by chebyInnerProduct(), chebyProfile(), ComplexChebyCoeff(), diff(), diff2(), gradient(), integrate(), L2InnerProduct(), randomProfile(), randomVprofile(), ubcFix(), and vbcFix().

```
348 {return re.a();}
```

Here is the call graph for this function:

Here is the caller graph for this function:



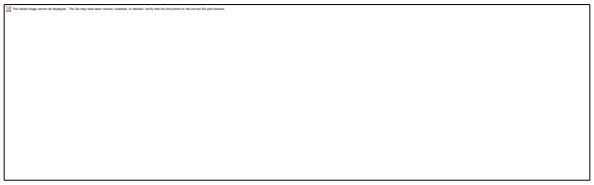
7.10.5.2 void ComplexChebyCoeff::add (int *n*, Complex *c*) [inline]

References Im(), im, Re(), and re.

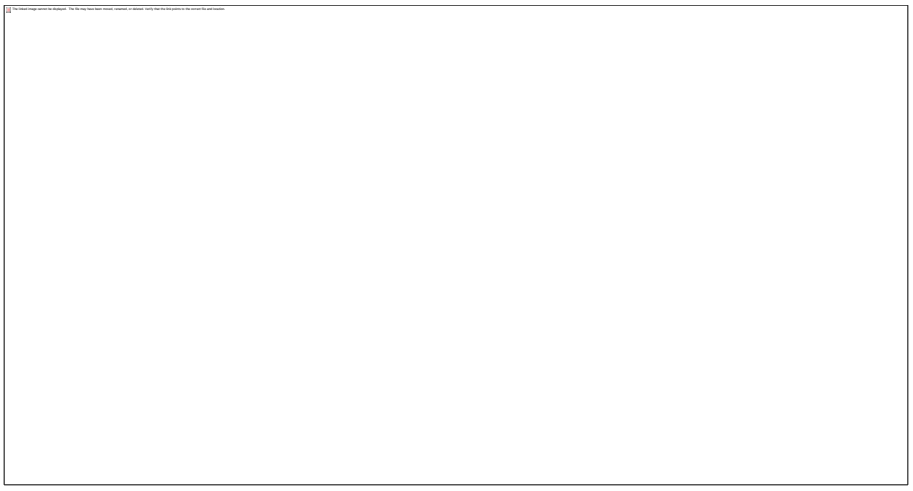
Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), and TauSolver::verify().

```
369                                 {
370   re[i] += Re(c);
371   im[i] += Im(c);
372 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.5.3 Real ComplexChebyCoeff::b () const [inline]

References ChebyCoeff::b(), and re.

Referenced by chebyInnerProduct(), chebyProfile(), ComplexChebyCoeff(), diff(), diff2(), gradient(), integrate(), L2InnerProduct(), randomProfile(), randomVprofile(), ubcFix(), and vbcFix().

```
349 {return re.b();}
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.10.5.4 void ComplexChebyCoeff::binaryDump (std::ostream & *os*) const

References ChebyCoeff::binaryDump(), im, and re.

```
1188                                                {
1189   re.binaryDump(os);
1190   im.binaryDump(os);
1191 }
```

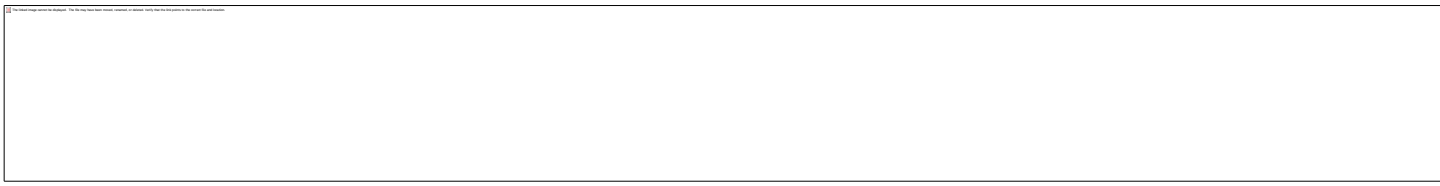
Here is the call graph for this function:

7.10.5.5 void ComplexChebyCoeff::binaryLoad (std::istream & *is*)

References ChebyCoeff::binaryLoad(), im, and re.

```
1192                                               {
1193   re.binaryLoad(is);
1194   im.binaryLoad(is);
1195 }
```

Here is the call graph for this function:

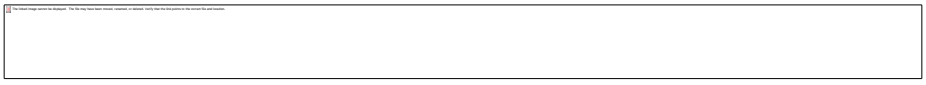


7.10.5.6 void ComplexChebyCoeff::chebyfft (const ChebyTransform & *t*)

References ChebyCoeff::chebyfft(), im, and re.

```
1075                                                {
1076   re.chebyfft(t);
1077   im.chebyfft(t);
1078 }
```

Here is the call graph for this function:



7.10.5.7 void ComplexChebyCoeff::chebyfft ()

References ChebyCoeff::numModes(), and re.

```
1053                                {
1054   ChebyTransform t(re.numModes());
1055   chebyfft(t);
1056 }
```

Here is the call graph for this function:

7.10.5.8 bool ComplexChebyCoeff::congruent (const ComplexChebyCoeff & *g*) const

References ChebyCoeff::congruent(), im, and re.

Referenced by operator*=().

```
1219                                                {
1220   assert(re.congruent(im));
1221   return (re.congruent(v.re) && im.congruent(v.im));
1222 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.10.5.9 void ComplexChebyCoeff::conjugate ()

References im.

```
1165                                {
1166   im *= -1.0;
1167 }
```

7.10.5.10 Complex ComplexChebyCoeff::eval (Real *x*) const

References ChebyCoeff::eval(), im, and re.

```
1131                                    {
1132   return Complex(re.eval(x), im.eval(x));
1133 }
```

Here is the call graph for this function:

7.10.5.11 Complex ComplexChebyCoeff::eval_a () const

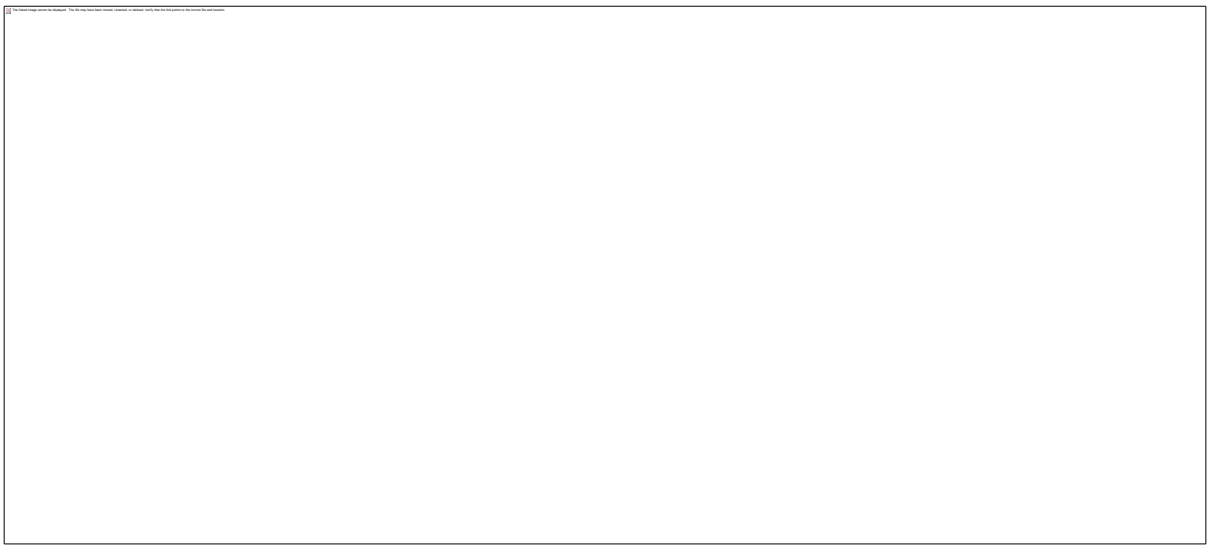
References ChebyCoeff::eval_a(), im, and re.

Referenced by bcDist2(), bcNorm2(), chebyProfile(), chebyUprofile(), chebyVprofile(), randomProfile(), randomUprofile(), randomVprofile(), PressureSolver::solve(), ubcFix(), vbcFix(), and TauSolver::verify().

```
1125                                    {
1126   return Complex(re.eval_a(), im.eval_a());
1127 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



7.10.5.12 Complex ComplexChebyCoeff::eval_b () const

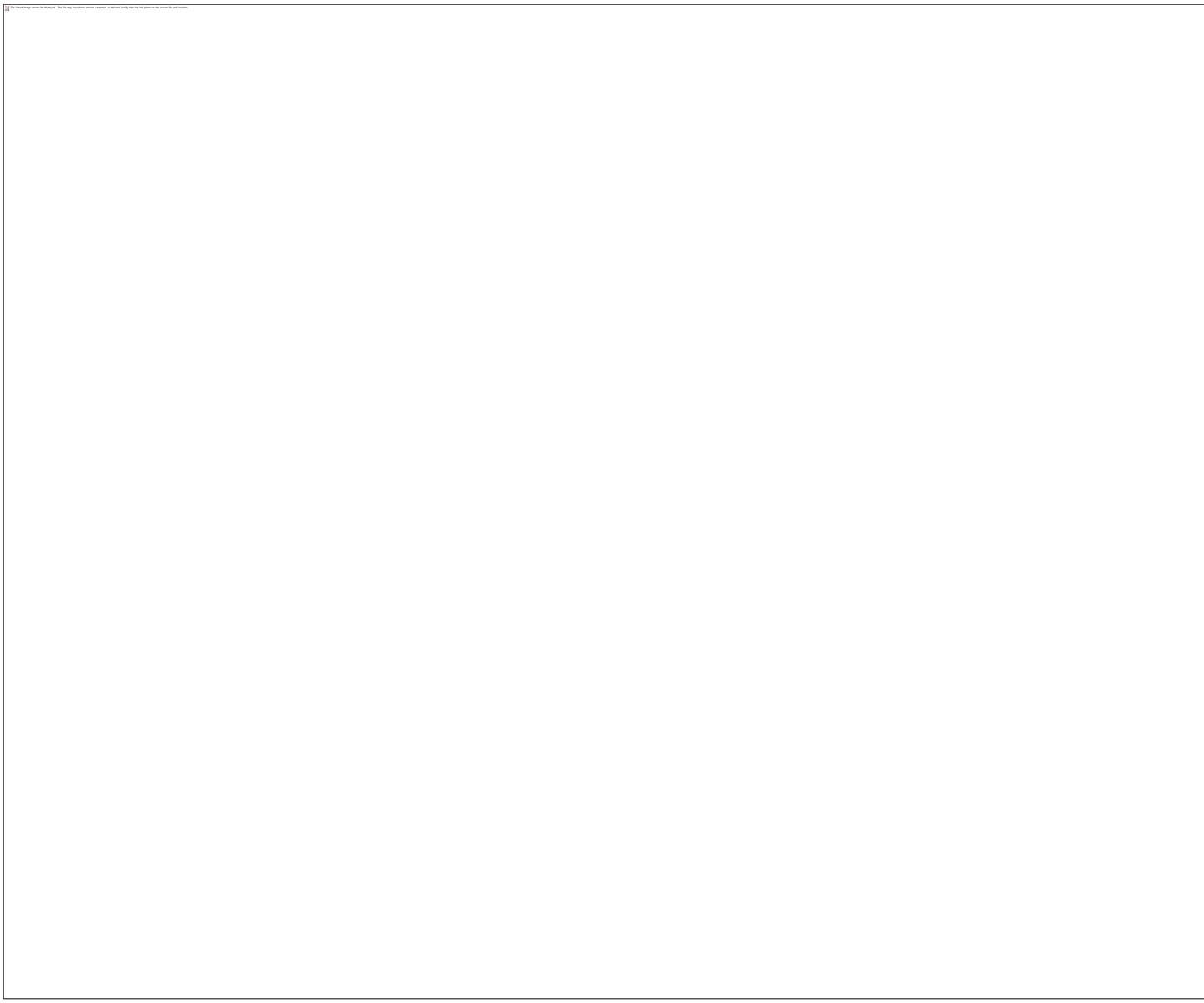
References ChebyCoeff::eval_b(), im, and re.

Referenced by bcDist2(), bcNorm2(), chebyProfile(), chebyUprofile(), chebyVprofile(), randomProfile(), randomUprofile(), randomVprofile(), PressureSolver::solve(), ubcFix(), vbcFix(), and TauSolver::verify().

```
1128                                {
1129   return Complex(re.eval_b(), im.eval_b());
1130 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



7.10.5.13 void ComplexChebyCoeff::fill (const ComplexChebyCoeff & *g*)

References ChebyCoeff::fill(), im, and re.

```
1101                                        {
1102   re.fill(v.re);
1103   im.fill(v.im);
1104 }
```

Here is the call graph for this function:

7.10.5.14 void ComplexChebyCoeff::ichebyfft (const ChebyTransform & *t*)

References ChebyCoeff::ichebyfft(), im, and re.

```
1079                                                {
1080   re.ichebyfft(t);
1081   im.ichebyfft(t);
1082 }
```

Here is the call graph for this function:

7.10.5.15 void ComplexChebyCoeff::ichebyfft ()

References ChebyCoeff::numModes(), and re.

```
1057                                {
1058   ChebyTransform t(re.numModes());
1059   ichebyfft(t);
1060 }
```

Here is the call graph for this function:

7.10.5.16 void ComplexChebyCoeff::interpolate (const ComplexChebyCoeff & *g*)

References im, ChebyCoeff::interpolate(), and re.

```
1105                                                 {
1106   re.interpolate(v.re);
1107   im.interpolate(v.im);
1108 }
```

Here is the call graph for this function:

7.10.5.17 Real ComplexChebyCoeff::L () const [inline]

References ChebyCoeff::L(), and re.

```
350 {return re.L();}
```

Here is the call graph for this function:

7.10.5.18 int ComplexChebyCoeff::length () const [inline]

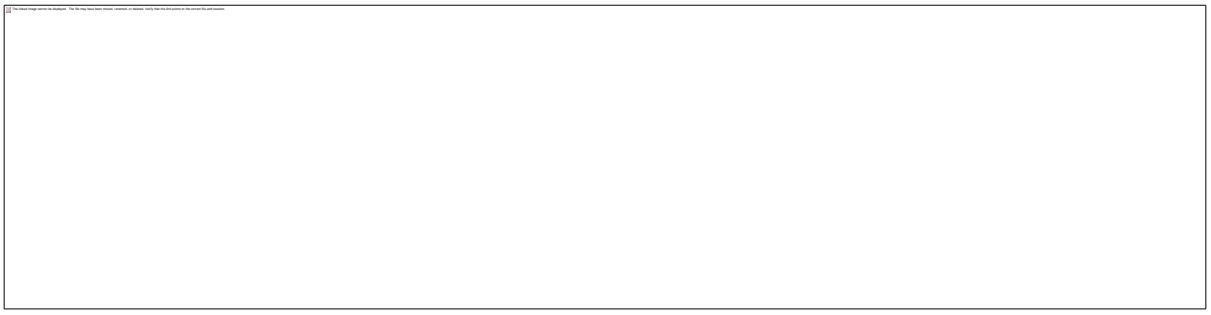
References Vector::length(), and re.

Referenced by chebyProfile(), chebyUprofile(), divcheck(), operator<<(), randomProfile(), randomUprofile(), and randomVprofile().

```
352 {return re.length();}
```

Here is the call graph for this function:

Here is the caller graph for this function:



7.10.5.19 void ComplexChebyCoeff::makePhysical (const ChebyTransform & *t*)

References im, ChebyCoeff::makePhysical(), and re.

```
1087                                        {
1088   re.makePhysical(t);
1089   im.makePhysical(t);
1090 }
```

Here is the call graph for this function:

7.10.5.20 void ComplexChebyCoeff::makePhysical ()

References ChebyCoeff::numModes(), and re.

Referenced by FlowField::saveProfile().

```
1065                                {
1066   ChebyTransform t(re.numModes());
1067   makePhysical(t);
1068 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.10.5.21 void ComplexChebyCoeff::makeSpectral (const ChebyTransform & *t*)

References im, ChebyCoeff::makeSpectral(), and re.

```
1083                                          {
1084   re.makeSpectral(t);
1085   im.makeSpectral(t);
1086 }
```

Here is the call graph for this function:

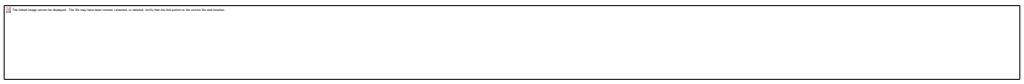
7.10.5.22 void ComplexChebyCoeff::makeSpectral ()

References ChebyCoeff::numModes(), and re.

Referenced by PressureSolver::solve().

```
1061                                {
1062   ChebyTransform t(re.numModes());
1063   makeSpectral(t);
1064 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.10.5.23 void ComplexChebyCoeff::makeState (fieldstate *s*, const ChebyTransform & *t*)

References im, ChebyCoeff::makeState(), and re.

```
1091                                                {
1092   re.makeState(s,t);
1093   im.makeState(s,t);
1094 }
```

Here is the call graph for this function:

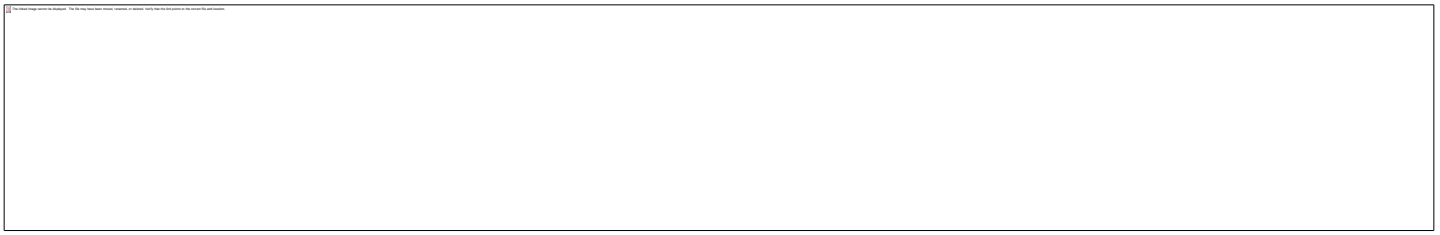
7.10.5.24 void ComplexChebyCoeff::makeState (fieldstate *s*)

References im, ChebyCoeff::makeState(), ChebyCoeff::numModes(), and re.

```
1069                                {
```

```
1070   ChebyTransform t(re.numModes());
1071   re.makeState(s,t);
1072   im.makeState(s,t);
1073 }
```

Here is the call graph for this function:



7.10.5.25 Complex ComplexChebyCoeff::mean () const

References im, ChebyCoeff::mean(), and re.

Referenced by TauSolver::verify().

```
1197                                {
1198   return Complex(re.mean(), im.mean());
1199 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.10.5.26 int ComplexChebyCoeff::N () const [inline]

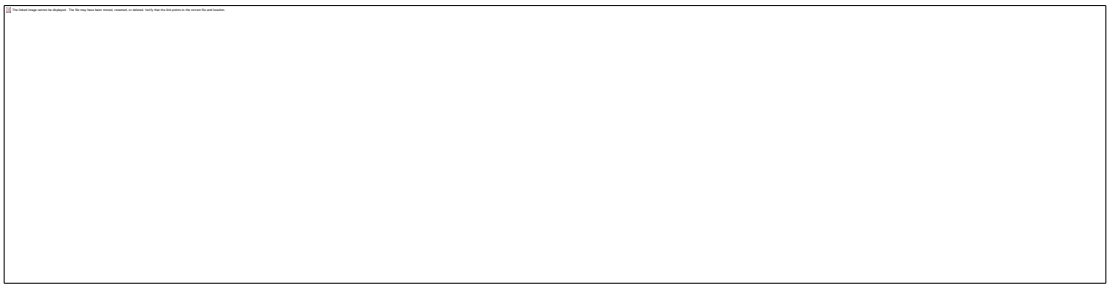
References ChebyCoeff::N(), and re.

Referenced by ComplexChebyCoeff(), FlowField::operator+=(), and FlowField::operator-=().

```
351 {return re.N();}
```

Here is the call graph for this function:

Here is the caller graph for this function:



7.10.5.27 int ComplexChebyCoeff::numModes () const [inline]

References ChebyCoeff::numModes(), and re.

Referenced by chebyInnerProduct(), chebyProfile(), diff(), diff2(), gradient(), integrate(), L2InnerProduct(), randomProfile(), randomVprofile(), TauSolver::tauDist(), and TauSolver::tauNorm().

```
353 {return re.numModes();}
```

Here is the call graph for this function:

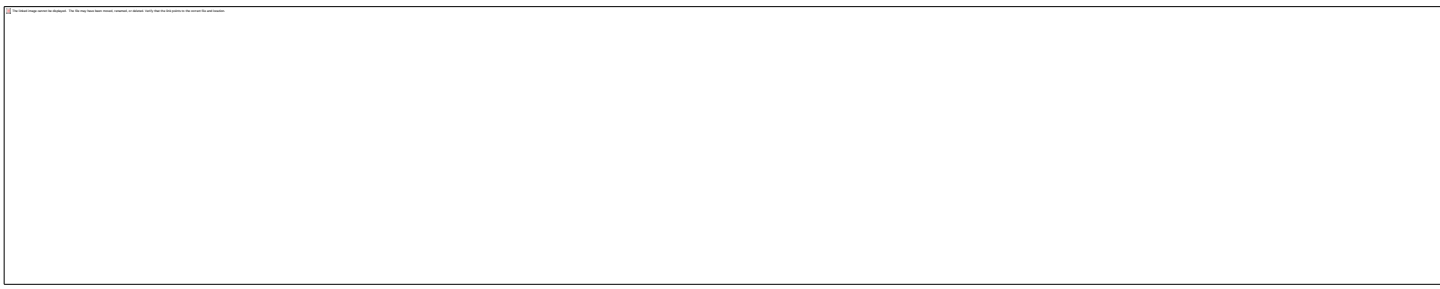
Here is the caller graph for this function:

7.10.5.28 ComplexChebyCoeff & ComplexChebyCoeff::operator*= (const ComplexChebyCoeff & *c*)

References congruent(), im, ChebyCoeff::numModes(), Physical, re, and ChebyCoeff::state().

```
1151                                                    {
1152   assert(congruent(u));
1153   assert(re.state() == Physical);
1154   assert(im.state() == Physical);
1155   Real r;
1156   Real i;
1157   for (int ny=0; ny<im.numModes(); ++ny) {
1158     r = re[ny]*u.re[ny] - im[ny]*u.im[ny];
1159     i = re[ny]*u.im[ny] + im[ny]*u.re[ny];
1160     re[ny] = r;
1161     im[ny] = i;
1162   }
1163   return *this;
1164 }
```

Here is the call graph for this function:



7.10.5.29 ComplexChebyCoeff & ComplexChebyCoeff::operator*= (Complex *c*)

References im, Im(), ChebyCoeff::numModes(), re, and Re().

```
1205                                                {
1206   Real cr = Re(c);
1207   Real ci = Im(c);
1208   Real ur;
1209   Real ui;
1210   for (int n=0; n<re.numModes(); ++n) {
1211     ur = re[n];
1212     ui = im[n];
1213     re[n] = ur*cr - ui*ci;
1214     im[n] = ur*ci + ui*cr;
1215   }
```

```
1216   return *this;
1217 }
```

Here is the call graph for this function:

7.10.5.30 ComplexChebyCoeff & ComplexChebyCoeff::operator*= (Real *c*)

References im, and re.

```
1145                                        {
1146   re *= c;
1147   im *= c;
1148   return *this;
1149 }
```

7.10.5.31 ComplexChebyCoeff & ComplexChebyCoeff::operator+= (const ComplexChebyCoeff & *f*)

References im, and re.

```
1134                                                {
1135   re += u.re;
1136   im += u.im;
1137   return *this;
1138 }
```

7.10.5.32 ComplexChebyCoeff & ComplexChebyCoeff::operator-= (const ComplexChebyCoeff & *f*)

References im, and re.

```
1139                                                {
1140   re -= u.re;
1141   im -= u.im;
1142   return *this;
1143 }
```

7.10.5.33 Complex ComplexChebyCoeff::operator[] (int *n*) const [inline]

References I(), im, and re.

```
362                                         {
363   return re[i] + I*im[i];
364 }
```

Here is the call graph for this function:

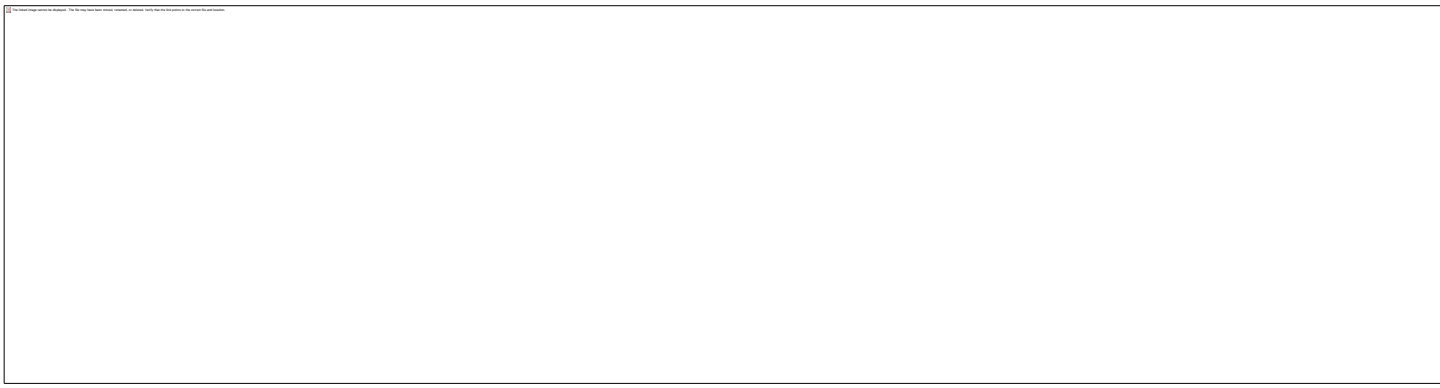
7.10.5.34 void ComplexChebyCoeff::randomize (Real *decay*, bool *a_dirichlet* = false, bool *b_dirichlet* = false)

References im, ChebyCoeff::randomize(), and re.

Referenced by BasisFunc::randomize().

```
1096                                                        {
1097   re.randomize(decay,a_dirichlet,b_dirichlet);
1098   im.randomize(decay,a_dirichlet,b_dirichlet);
1099 }
```

Here is the call graph for this function:



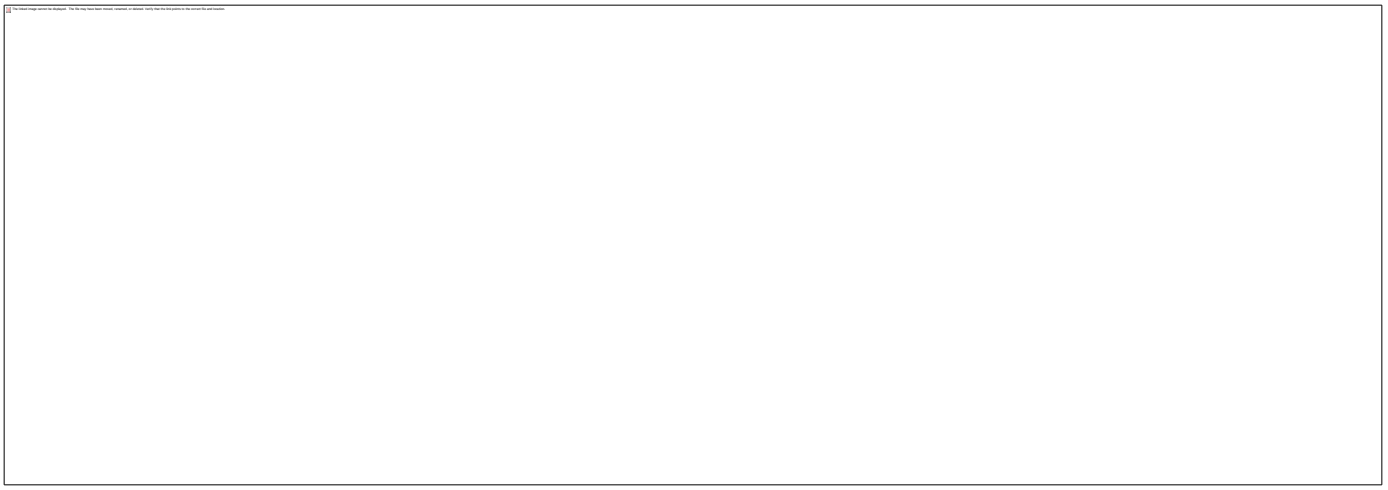
Here is the caller graph for this function:

7.10.5.35 void ComplexChebyCoeff::reflect (const ComplexChebyCoeff & *g*, parity *p*)

References im, re, and ChebyCoeff::reflect().

```
1109                                                    {
1110   re.reflect(v.re, p);
1111   im.reflect(v.im, p);
1112 }
```

Here is the call graph for this function:



7.10.5.36 void ComplexChebyCoeff::resize (int *N*)

References im, re, and Vector::resize().

```
1051 {re.resize(N); im.resize(N);}
```

Here is the call graph for this function:

7.10.5.37 void ComplexChebyCoeff::save (const std::string & *filebase*, fieldstate *s* = Physical) const

References ChebyCoeff::a(), ChebyCoeff::b(), im, Vector::length(), ChebyCoeff::makeState(), re, REAL_DIGITS, REAL_IOWIDTH, and ChebyCoeff::state().

```
1169                                                    {
1170   fieldstate origstate = re.state();
1171   ((ChebyCoeff&) *this).makeState(savestate);
```

```
1172
1173   string filename(filebase);
1174   filename += string(".asc");
1175   ofstream os(filename.c_str());
1176
1177   os << scientific << setprecision(REAL_DIGITS);
1178   os << "% " << re.length() << ' ' << re.a() << ' ' << re.b() << ' '
1179      << re.state() << '\n';
1180   for (int n=0; n<re.length(); ++n)
1181     os  << setw(REAL_IOWIDTH) << re[n] << ' '
1182         << setw(REAL_IOWIDTH) << im[n] << '\n';
1183   os.close();
1184
1185   ((ChebyCoeff&) *this).makeState(origstate);
1186 }
```

Here is the call graph for this function:

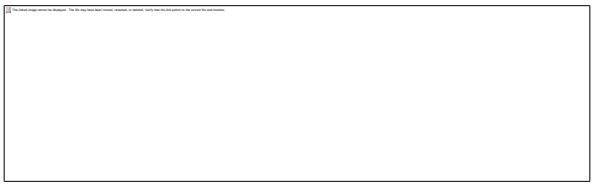
7.10.5.38 void ComplexChebyCoeff::set (int *n*, Complex *c*) [inline]

References Im(), im, Re(), and re.

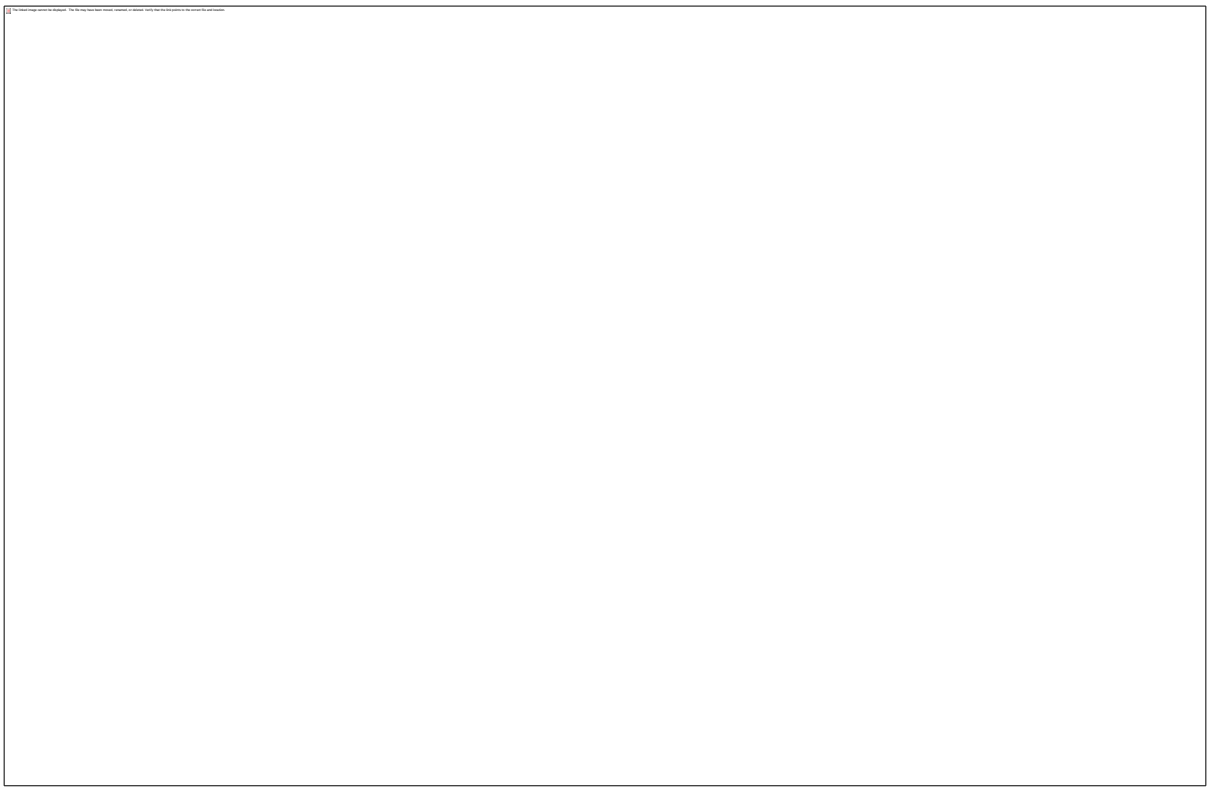
Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), chebyUprofile(), chebyVprofile(), FlowField::energy(), FlowField::profile(), randomUprofile(), randomVprofile(), skewsymmetricNL_task4::Run(), FlowField::saveDivSpectrum(), FlowField::saveSpectrum(), TauSolver::solve(), PressureSolver::solve(), and PoissonSolver::solve().

```
365                                     {
366   re[i] = Re(c);
367   im[i] = Im(c);
368 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.5.39 void ComplexChebyCoeff::setBounds (Real *a*, Real *b*)

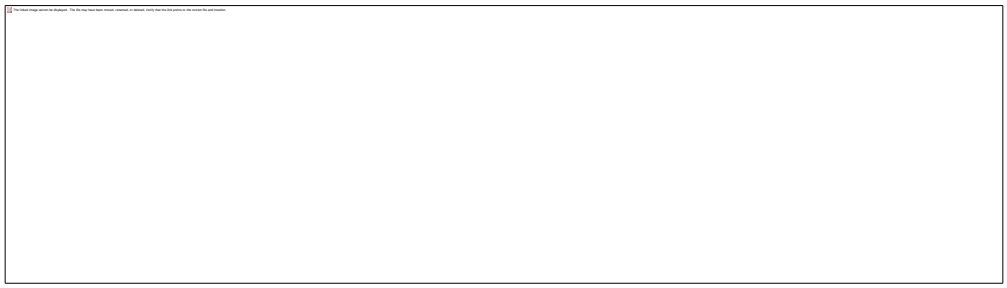
References im, re, and ChebyCoeff::setBounds().

Referenced by dotgrad(), ubcFix(), and vbcFix().

```
1118                                    {
1119   re.setBounds(a,b);
1120   im.setBounds(a,b);
1121 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



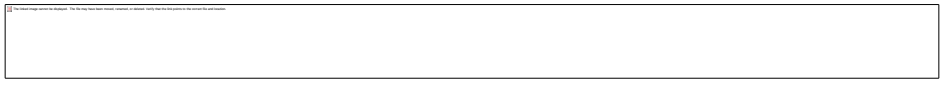
7.10.5.40 void ComplexChebyCoeff::setState (fieldstate s)

References im, re, and ChebyCoeff::setState().

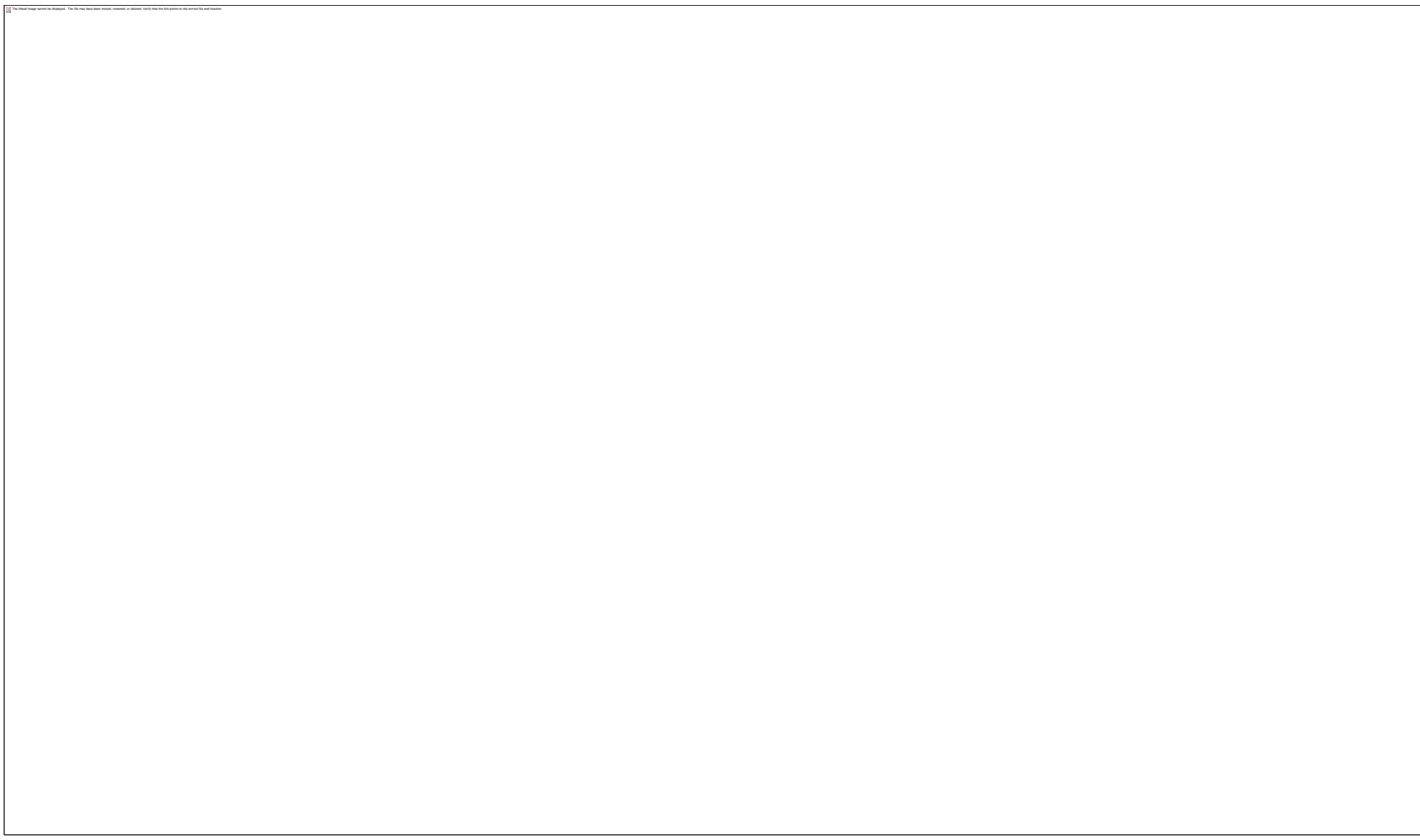
Referenced by chebyProfile(), chebyUprofile(), chebyVprofile(), dotgrad(), randomProfile(), randomUprofile(), randomVprofile(), and PressureSolver::solve().

```
1114                                    {
1115   re.setState(s);
1116   im.setState(s);
1117 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.5.41 void ComplexChebyCoeff::setToZero ()

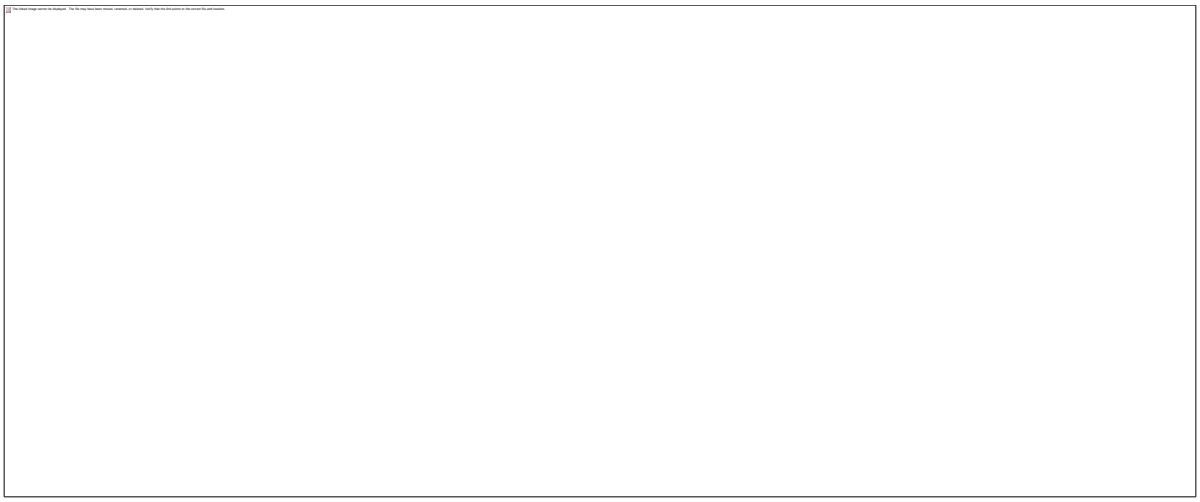
References im, re, and ChebyCoeff::setToZero().

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), chebyProfile(), chebyUprofile(), chebyVprofile(), dotgrad(), BasisFunc::randomize(), and randomProfile().

```
1100 {re.setToZero(); im.setToZero();}
```

Here is the call graph for this function:

Here is the caller graph for this function:



7.10.5.42 fieldstate ComplexChebyCoeff::state () const [inline]

References re, and ChebyCoeff::state().

Referenced by FlowField::addProfile(), chebyInnerProduct(), gradient(), L2InnerProduct(), FlowField::operator+=(), and FlowField::operator-=().

```
354 {return re.state();}
```

Here is the call graph for this function:

Here is the caller graph for this function:

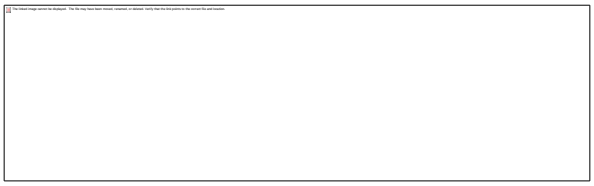
7.10.5.43 void ComplexChebyCoeff::sub (int *n*, Complex *c*) [inline]

References Im(), im, Re(), and re.

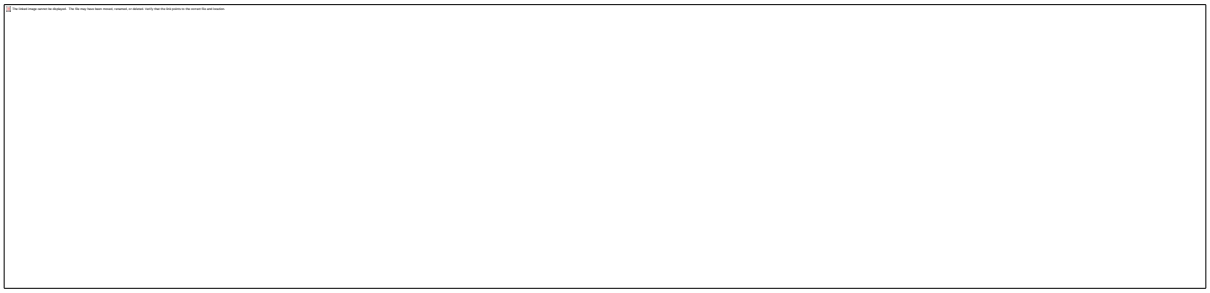
Referenced by chebyUprofile(), chebyVprofile(), randomUprofile(), randomVprofile(), ubcFix(), and vbcFix().

```
373                                 {
374   re[i] -= Re(c);
375   im[i] -= Im(c);
376 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.6 Friends And Related Function Documentation

7.10.6.1 void swap (ComplexChebyCoeff & *f*, ComplexChebyCoeff & *g*) [friend]

7.10.7 Member Data Documentation

7.10.7.1 ChebyCoeff ComplexChebyCoeff::im

Referenced by add(), PPEDNS::advance(), binaryDump(), binaryLoad(), chebyDist(), chebyDist2(), chebyfft(), chebyInnerProduct(), chebyNorm(), chebyNorm2(), chebyProfile(), ComplexChebyCoeff(), congruent(), conjugate(), diff(), diff2(), eval(), eval_a(), eval_b(), fill(), ichebyfft(), Im(), integrate(), interpolate(), L1Dist(), L1Norm(), L2Dist(), L2Dist2(),

L2InnerProduct(), L2Norm(), L2Norm2(), LinfDist(), LinfNorm(), makePhysical(), makeprtb(), makeroll(), makeSpectral(), makeState(), makewave(), makewobl(), mean(), operator*=(), operator+=(), operator-=(), operator<<(), operator==(), operator[](), randomize(), randomProfile(), reflect(), resize(), save(), set(), setBounds(), setState(), setToZero(), TauSolver::solve(), PoissonSolver::solve(), sub(), swap(), and TauSolver::verify().

7.10.7.2 ChebyCoeff ComplexChebyCoeff::re

Referenced by a(), add(), PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), b(), binaryDump(), binaryLoad(), chebyDist(), chebyDist2(), chebyfft(), chebyInnerProduct(), chebyNorm(), chebyNorm2(), ComplexChebyCoeff(), congruent(), diff(), diff2(), eval(), eval_a(), eval_b(), fill(), ichebyfft(), integrate(), interpolate(), L(), L1Dist(), L1Norm(), L2Dist(), L2Dist2(), L2InnerProduct(), L2Norm(), L2Norm2(), length(), LinfDist(), LinfNorm(), makemeen(), makePhysical(), makeprtb(), makeroll(), makeSpectral(), makeState(), makewave(), makewobl(), mean(), N(), numModes(), operator*=(), operator+=(), operator-=(), operator<<(), operator==(), operator[](), FlowField::profile(), randomize(), Re(), reflect(), resize(), save(), set(), setBounds(), setState(), setToZero(), TauSolver::solve(), PoissonSolver::solve(), state(), sub(), swap(), and TauSolver::verify().

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

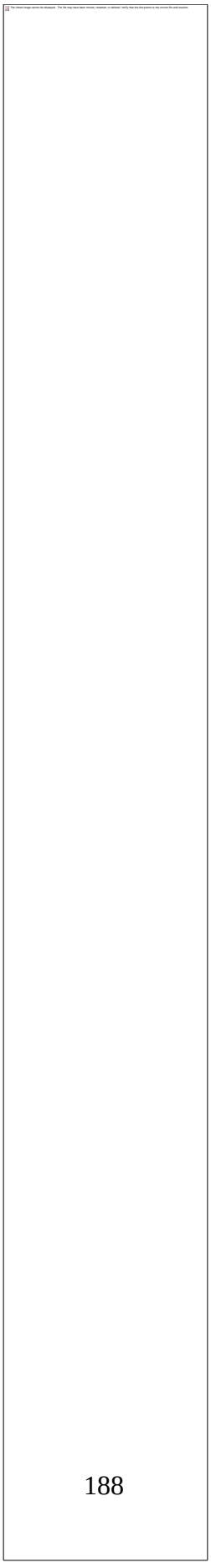
- /_cuda/channelflow-0.9.22/channelflow/chebyshev.h
- /_cuda/channelflow-0.9.22/channelflow/chebyshev.cpp

7.10.7.4

7.11 DNS Class Reference

#include <dns.h>

Collaboration diagram for DNS:


The mixed image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.11.1 Public Member Functions

- DNS ()
- DNS (const DNS &dns)
- DNS (FlowField &u, const ChebyCoeff &Ubase, Real nu, Real dt, const DNSFlags &flags, Real t=0.0)
- ~DNS ()
- DNS & operator= (const DNS &dns)
- void advance (FlowField &u, FlowField &q, int nSteps=1)
- void reset_dt (Real dt)
- void reset_time (Real t)
- void reset_dPdx (Real dPdx)
- void reset_Ubulk (Real Ubulk)
- bool push (const FlowField &u)
- bool full () const
- int order () const
- int Ninitsteps () const
- Real dt () const
- Real CFL () const
- Real time () const
- Real dPdx () const
- Real Ubulk () const
- Real dPdxRef () const
- Real UbulkRef () const
- const DNSFlags & flags () const
- TimeStepMethod timestepping () const

7.11.2 Private Member Functions

- DNSAlgorithm * newAlgorithm (FlowField &u, const ChebyCoeff &Ubase, Real nu, Real dt, const DNSFlags &flags, Real t)

7.11.3 Private Attributes

- DNSAlgorithm * main_algorithm_
- DNSAlgorithm * init_algorithm_

7.11.4 Constructor & Destructor Documentation

7.11.4.1 DNS::DNS ()

```
648   :
649   main_algorithm_(0),
650   init_algorithm_(0)
```

```
651 {}
```

7.11.4.2 DNS::DNS (const DNS & *dns*)

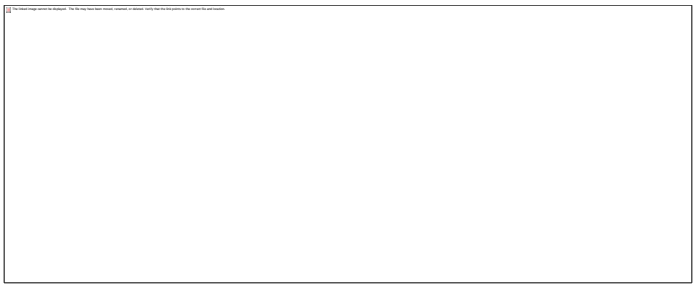
```
676   :
677   main_algorithm_(dns.main_algorithm_ ? dns.main_algorithm_->clone() : 0),
678   init_algorithm_(dns.init_algorithm_ ? dns.init_algorithm_->clone() : 0)
679 {}
```

7.11.4.3 DNS::DNS (FlowField & *u*, const ChebyCoeff & *Ubase*, Real *nu*, Real *dt*, const DNSFlags & *flags*, Real *t* = 0.0)

References DNSAlgorithm::full(), init_algorithm_, DNSFlags::initstepping, main_algorithm_, newAlgorithm(), DNSAlgorithm::Ninitsteps(), and DNSFlags::timestepping.

```
655   :
656   main_algorithm_(0),
657   init_algorithm_(0)
658 {
659   main_algorithm_ = newAlgorithm(u, Ubase, nu, dt, flags, t);
660
661   if (!main_algorithm_->full() &&
662       flags.initstepping != flags.timestepping) {
663     DNSFlags initflags = flags;
664     initflags.timestepping = flags.initstepping;
665     init_algorithm_ = newAlgorithm(u, Ubase, nu, dt, initflags, t);
666
667     // Safety check
668     if (init_algorithm_->Ninitsteps() != 0)
669       cerr << "DNS::DNS(u, Ubase, nu, dt, flags, t) :\n" << flags.initstepping
670            << " can't initialize " << flags.timestepping
671            << " since it needs initialization itself.\n";
672   }
673 }
```

Here is the call graph for this function:



7.11.4.4 DNS::~DNS ()

References init_algorithm_, and main_algorithm_.

```
718         {
719   delete main_algorithm_;
720   delete init_algorithm_;
721 }
```

7.11.5 Member Function Documentation

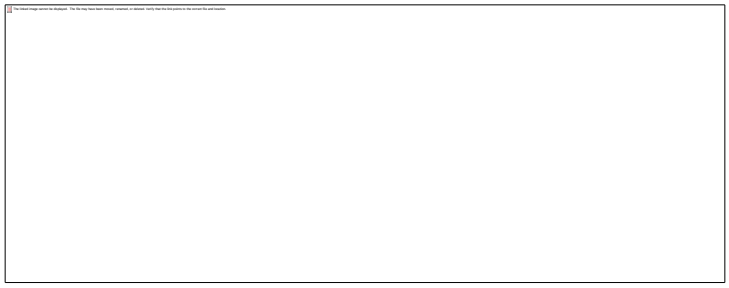
7.11.5.1 void DNS::advance (FlowField & *u*, FlowField & *q*, int *nSteps* = 1)

References DNSAlgorithm::advance(), DNSAlgorithm::full(), init_algorithm_, main_algorithm_, and DNSAlgorithm::push().

```
731                                         {
732   assert(main_algorithm_);
733
734   // Error check
735   if (!main_algorithm_->full() && !init_algorithm_) {
736     cerr << "DNS::advance(u,q,Nsteps) : the main algorithm is uninitialized,\n"
737          << "and the initialization algorithm is not set. This should not be\n"
738          << "possible. Please submit a bug report (see documentation)."
739          << endl;
740     exit(1);
741   }
742   int n=0;
743   while (!main_algorithm_->full() && n<Nsteps) {
744     init_algorithm_->advance(u,q,1);
745     main_algorithm_->push(u);
746     if (main_algorithm_->full()) {
747       delete init_algorithm_;
748       init_algorithm_ = 0;
749     }
```

```
750     ++n;
751   }
752   main_algorithm_->advance(u,q,Nsteps-n);
753 }
```

Here is the call graph for this function:



7.11.5.2 Real DNS::CFL () const

References DNSAlgorithm::CFL(), and main_algorithm_.

```
828                 {
829   assert(main_algorithm_);
830   return main_algorithm_->CFL();
831 }
```

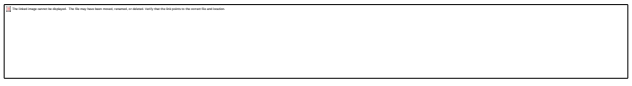
Here is the call graph for this function:

7.11.5.3 Real DNS::dPdx () const

References DNSAlgorithm::dPdx(), and main_algorithm_.

```
836                  {
837   assert(main_algorithm_);
838   return main_algorithm_->dPdx();
839 }
```

Here is the call graph for this function:



7.11.5.4 Real DNS::dPdxRef () const

References DNSAlgorithm::dPdxRef(), and main_algorithm_.

```
844                     {
845   assert(main_algorithm_);
846   return main_algorithm_->dPdxRef();
847 }
```

Here is the call graph for this function:

7.11.5.5 Real DNS::dt () const

References DNSAlgorithm::dt(), and main_algorithm_.

```
824                 {
825   assert(main_algorithm_);
826   return main_algorithm_->dt();
827 }
```

Here is the call graph for this function:

7.11.5.6 const DNSFlags & DNS::flags () const

References DNSAlgorithm::flags(), and main_algorithm_.

```
852                         {
853   assert(main_algorithm_);
854   return main_algorithm_->flags();
855 }
```

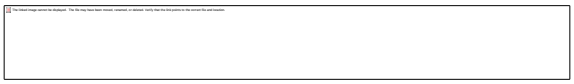
Here is the call graph for this function:

7.11.5.7 bool DNS::full () const

References DNSAlgorithm::full(), and main_algorithm_.

```
812                 {
813   assert(main_algorithm_);
814   return main_algorithm_->full();
815 }
```

Here is the call graph for this function:



7.11.5.8 DNSAlgorithm * DNS::newAlgorithm (FlowField & *u*, const ChebyCoeff & *Ubase*, Real *nu*, Real *dt*, const DNSFlags & *flags*, Real *t*) [private]

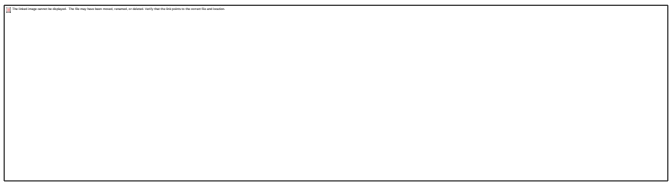
References CNAB2, CNFE1, CNRK2, PPEQ2, SBDF1, SBDF2, SBDF3, SBDF4, SMRK2, and DNSFlags::timestepping.

Referenced by DNS(), and reset_dt().

```
683                                 {
684 
685   printf("IG: newAlgorithm called\n");
686 
687   DNSAlgorithm* alg = 0;
688   switch (flags.timestepping) {
689   case CNFE1:
690   case SBDF1:
691   case SBDF2:
692   case SBDF3:
693   case SBDF4:
694     //printf("IG: new MultistepDNS called\n");
695     alg = new MultistepDNS(u, Ubase, nu, dt, flags, t);
696 
697     break;
698   case CNRK2:
699     alg = new RungeKuttaDNS(u, Ubase, nu, dt, flags, t);
700     break;
701   case SMRK2:
702   case CNAB2:
703     //printf("IG: new CNABstyleDNS called\n");
704 
705     alg = new CNABstyleDNS(u, Ubase, nu, dt, flags, t);
706     break;
707   case PPEQ2:
708     alg = new PPEDNS(u, Ubase, nu, dt, flags, t);
709     break;
```

```
710   default:
711     cerr << "DNS::newAlgorithm : algorithm " << flags.timestepping
712          << " is unimplemented" << endl;
713     exit(1);
714   }
715   return alg;
716 }
```

Here is the caller graph for this function:



7.11.5.9 int DNS::Ninitsteps () const

References main_algorithm_, and DNSAlgorithm::Ninitsteps().

```
820                         {
821   assert(main_algorithm_);
822   return main_algorithm_->Ninitsteps();
823 }
```

Here is the call graph for this function:

7.11.5.10 DNS & DNS::operator= (const DNS & *dns*)

References DNSAlgorithm::clone(), init_algorithm_, and main_algorithm_.

```
723                                 {
724   delete main_algorithm_;
725   delete init_algorithm_;
726   main_algorithm_ = dns.main_algorithm_ ? dns.main_algorithm_->clone() : 0;
727   init_algorithm_ = dns.init_algorithm_ ? dns.init_algorithm_->clone() : 0;
728   return *this;
729 }
```

Here is the call graph for this function:

7.11.5.11 int DNS::order () const

References main_algorithm_, and DNSAlgorithm::order().

```
816                 {
817   assert(main_algorithm_);
818   return main_algorithm_->order();
819 }
```

Here is the call graph for this function:

7.11.5.12 bool DNS::push (const FlowField & *u*)

References main_algorithm_, and DNSAlgorithm::push().

```
808                         {
809   assert(main_algorithm_);
810   return main_algorithm_->push(u);
811 }
```

Here is the call graph for this function:

7.11.5.13 void DNS::reset_dPdx (Real *dPdx*)

References main_algorithm_, and DNSAlgorithm::reset_dPdx().

```
796                         {
797   assert(main_algorithm_);
798   main_algorithm_->reset_dPdx(dPdx);
799 }
```

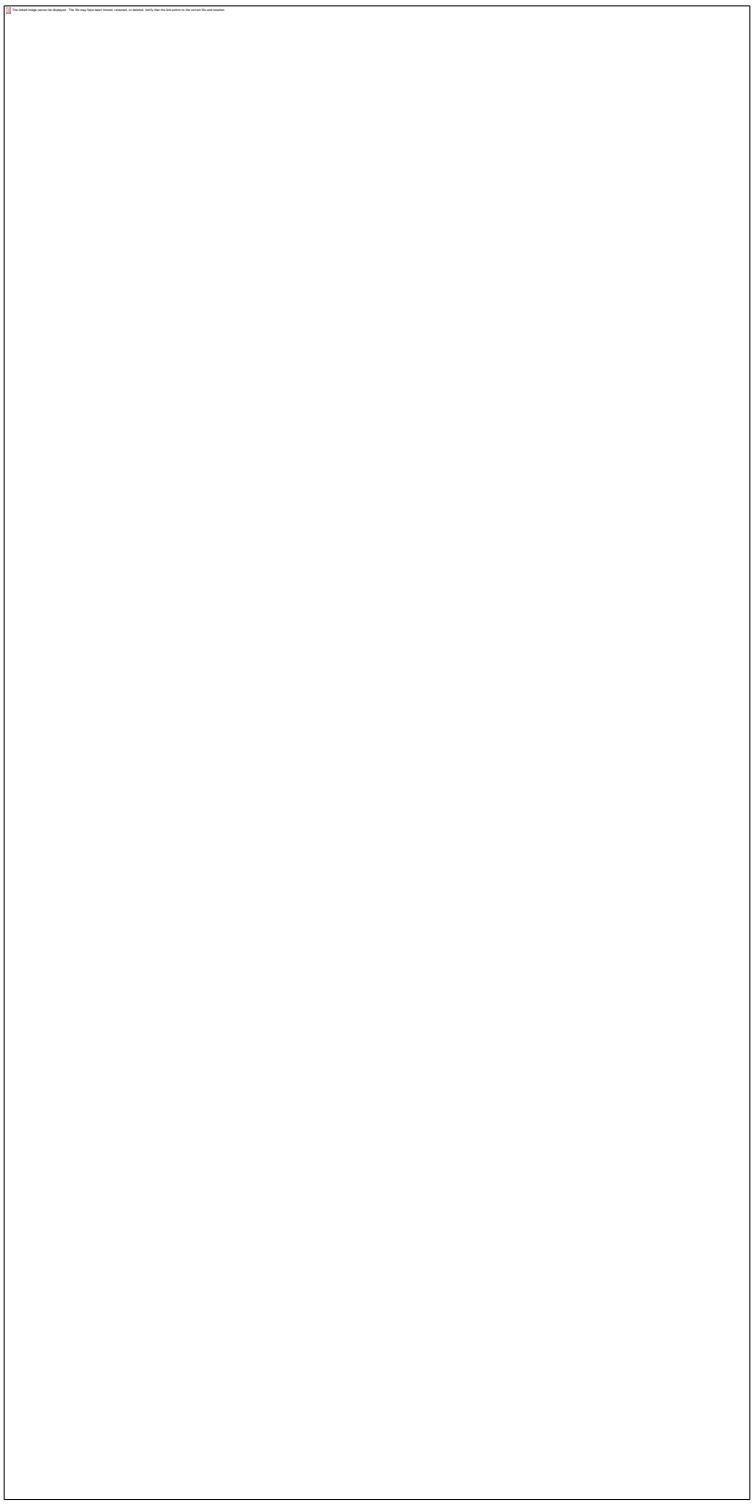
Here is the call graph for this function:

7.11.5.14 void DNS::reset_dt (Real *dt*)

References DNSAlgorithm::a(), DNSAlgorithm::b(), DNSAlgorithm::flags(), DNSAlgorithm::full(), init_algorithm_, DNSFlags::initstepping, DNSAlgorithm::Lx(), DNSAlgorithm::Lz(), main_algorithm_, newAlgorithm(), DNSAlgorithm::Ninitsteps(), DNSAlgorithm::nu(), DNSAlgorithm::Nx(), DNSAlgorithm::Ny(), DNSAlgorithm::Nz(), DNSAlgorithm::reset_dt(), DNSAlgorithm::time(), DNSFlags::timestepping, and DNSAlgorithm::Ubase().

```
755                     {
756   assert(main_algorithm_);
757   main_algorithm_->reset_dt(dt);
758
759   DNSFlags mainflags = main_algorithm_->flags();
760   if (!main_algorithm_->full() &&
761       mainflags.initstepping != mainflags.timestepping) {
762
763     DNSFlags initflags = mainflags;
764     initflags.timestepping = mainflags.initstepping;
765
766     // An initialization algorithm needs only u's parameters at construction,
767     // not its data, so we can construct it from a zero-valued u of right size
768     FlowField u(main_algorithm_->Nx(), main_algorithm_->Ny(),
769                 main_algorithm_->Nz(), 3,
770                 main_algorithm_->Lx(), main_algorithm_->Lz(),
771                 main_algorithm_->a(),  main_algorithm_->b());
772
773
774 //IG-s
775     if(init_algorithm_) delete init_algorithm_;
776 //IG-e
777     init_algorithm_ = newAlgorithm(u, main_algorithm_->Ubase(),
778                                    main_algorithm_->nu(), dt, initflags,
779                                    main_algorithm_->time());
780
781     // Safety check
782     if (init_algorithm_->Ninitsteps() != 0)
783       cerr << "DNS::DNS(u, Ubase, nu, dt, flags, t) :\n"
784            << mainflags.initstepping  << " can't initialize "
785            << mainflags.timestepping
786            << " since it needs initialization itself.\n";
787   }
788   // No need for safety check on init_algorithm, has already been ok'd in ctor
789 }
```

Here is the call graph for this function:

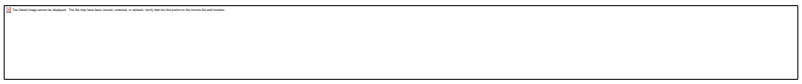


7.11.5.15 void DNS::reset_time (Real *t*)

References main_algorithm_, and DNSAlgorithm::reset_time().

```
792                         {
793   assert(main_algorithm_);
794   main_algorithm_->reset_time(t);
795 }
```

Here is the call graph for this function:



7.11.5.16 void DNS::reset_Ubulk (Real *Ubulk*)

References main_algorithm_, and DNSAlgorithm::reset_Ubulk().

```
800                             {
801   assert(main_algorithm_);
802   main_algorithm_->reset_Ubulk(Ubulk);
803 }
```

Here is the call graph for this function:

7.11.5.17 Real DNS::time () const

References main_algorithm_, and DNSAlgorithm::time().

```
832                     {
833   assert(main_algorithm_);
834 return main_algorithm_->time();
835 }
```

Here is the call graph for this function:

7.11.5.18 TimeStepMethod DNS::timestepping () const

References main_algorithm_, and DNSAlgorithm::timestepping().

```
856                                 {
```

```
857   assert(main_algorithm_);
858   return main_algorithm_->timestepping();
859 }
```

Here is the call graph for this function:

7.11.5.19 Real DNS::Ubulk () const

References main_algorithm_, and DNSAlgorithm::Ubulk().

```
840                     {
841   assert(main_algorithm_);
842   return main_algorithm_->Ubulk();
843 }
```

Here is the call graph for this function:

7.11.5.20 Real DNS::UbulkRef () const

References main_algorithm_, and DNSAlgorithm::UbulkRef().

```
848                     {  // the bulk velocity enforced during integ.
849   assert(main_algorithm_);
850   return main_algorithm_->UbulkRef();
851 }
```

Here is the call graph for this function:

7.11.6 Member Data Documentation

7.11.6.1 DNSAlgorithm* DNS::init_algorithm_ [private]

Referenced by advance(), DNS(), operator=(), reset_dt(), and ~DNS().

7.11.6.2 DNSAlgorithm* DNS::main_algorithm_ [private]

Referenced by advance(), CFL(), DNS(), dPdx(), dPdxRef(), dt(), flags(), full(), Ninitsteps(), operator=(), order(), push(), reset_dPdx(), reset_dt(), reset_time(), reset_Ubulk(), time(), timestepping(), Ubulk(), UbulkRef(), and ~DNS().

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

- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

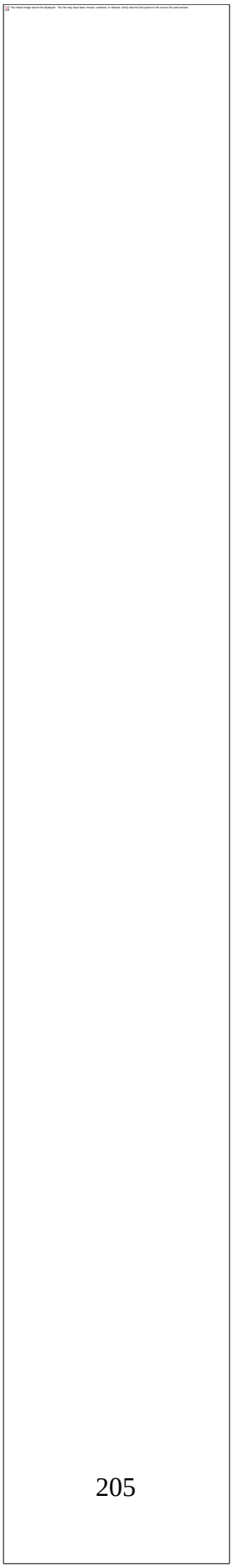
7.11.6.4

7.12 DNSAlgorithm Class Reference

#include <dns.h>

Inheritance diagram for DNSAlgorithm:

Collaboration diagram for DNSAlgorithm:


The mixed image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.12.1 Public Member Functions

- DNSAlgorithm ()
- DNSAlgorithm (FlowField &u, const ChebyCoeff &Ubase, Real nu, Real dt, const DNSFlags &flags, Real t=0)
- virtual ~DNSAlgorithm ()
- DNSAlgorithm & operator= (const DNSAlgorithm &dns)
- virtual void advance (FlowField &u, FlowField &q, int nSteps=1)=0
- virtual void reset_dt (Real dt)=0
- virtual bool push (const FlowField &u)
- virtual bool full () const
- void reset_time (Real t)
- void reset_dPdx (Real dPdx)
- void reset_Ubulk (Real Ubulk)
- int order () const
- int Ninitsteps () const
- int Nx () const
- int Ny () const
- int Nz () const
- int Mx () const
- int My () const
- int Mz () const
- Real Lx () const
- Real Lz () const
- Real a () const
- Real b () const
- Real nu () const
- Real dt () const
- Real CFL () const
- Real time () const
- Real dPdx () const
- Real Ubulk () const
- Real dPdxRef () const
- Real UbulkRef () const
- const DNSFlags & flags () const
- const ChebyCoeff & Ubase () const
- TimeStepMethod timestepping () const
- virtual DNSAlgorithm * clone () const =0

7.12.2 Protected Member Functions

- int kxmaxDealiased () const
- int kzmaxDealiased () const
- bool isAliasedMode (int kx, int kz) const

7.12.3 Protected Attributes

- int Nx_
- int Ny_
- int Nz_
- int Mx_
- int Mz_
- int Nyd_
- int kxd_max_
- int kzd_max_
- Real Lx_
- Real Lz_
- Real a_
- Real b_
- DNSFlags flags_
- int order_
- int Ninitsteps_
- Real nu_
- Real dt_
- Real t_
- Real cfl_
- Real dPdxRef_
- Real dPdxAct_
- Real UbulkRef_
- Real UbulkAct_
- Real UbulkBase_
- ChebyCoeff Ubase_
- ChebyCoeff Ubaseyy_
- FlowField tmp_
- ComplexChebyCoeff uk_
- ComplexChebyCoeff vk_
- ComplexChebyCoeff wk_
- ComplexChebyCoeff Pk_
- ComplexChebyCoeff Pyk_
- ComplexChebyCoeff Rxk_
- ComplexChebyCoeff Ryk_
- ComplexChebyCoeff Rzk_

7.12.4 Constructor & Destructor Documentation

7.12.4.1 DNSAlgorithm::DNSAlgorithm ()

```
874   :
875   Nx_(0),
```

```
876   Ny_(0),
877   Nz_(0),
878   Mx_(0),
879   Mz_(0),
880   Nyd_(0),
881   kxd_max_(0),
882   kzd_max_(0),
883   Lx_(0),
884   Lz_(0),
885   a_(0),
886   b_(0),
887   flags_(),
888   order_(0),
889   Ninitsteps_(0),
890   nu_(0),
891   dt_(0),
892   t_(0),
893   cfl_(0),
894   dPdxRef_(0),
895   dPdxAct_(0),
896   UbulkRef_(0),
897   UbulkAct_(0),
898   UbulkBase_(0),
899   Ubase_(),
900   Ubaseyy_(),
901   tmp_(),
902   uk_(),
903   vk_(),
904   wk_(),
905   Pk_(),
906   Pyk_(),
907   Rxk_(),
908   Ryk_(),
909   Rzk_()
910 {}
```

7.12.4.2 DNSAlgorithm::DNSAlgorithm (FlowField & *u*, const ChebyCoeff & *Ubase*, Real *nu*, Real *dt*, const DNSFlags & *flags*, Real *t* = 0)

References FlowField::a(), a_, Alternating, Alternating_, FlowField::assertState(), FlowField::b(), b_, BulkVelocity, cfl_, FlowField::CFLfactor(), FlowField::cmplx(), DNSFlags::constraint, Convection, DNSFlags::dealias_xz(), diff(), Divergence, dPdxAct_, dPdxRef_, dt_, ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), flags_, isAliasedMode(), FlowField::kx(), FlowField::kz(), Vector::length(), FlowField::Lx(), FlowField::Lz(),

ChebyCoeff::makePhysical(), ChebyCoeff::makeSpectral(), ChebyCoeff::mean(), Mx_, Mz_, DNSFlags::nonlinearity, FlowField::Nx(), FlowField::Ny(), Ny_, Nyd_, FlowField::Nz(), Physical, pi, Re(), FlowField::resize(), ChebyCoeff::setState(), SkewSymmetric, Spectral, tmp_, Ubase_, Ubaseyy_, UbulkAct_, UbulkBase_, UbulkRef_, and FlowField::vectorDim().

```
914   :
915   Nx_(u.Nx()),
916   Ny_(u.numYmodes()),
917   Nz_(u.Nz()),
918   Mx_(u.numXmodes()),
919   Mz_(u.numZmodes()),
920   Nyd_(flags.dealias_y() ? 2*(u.numYmodes()-1)/3 + 1 : u.numYmodes()),
921   //kxd_max_(flags.dealias_xz() ? 2*(u.kxmax()/3) : u.kxmax()),
922   //kzd_max_(flags.dealias_xz() ? 2*(u.kzmax()/3) : u.kzmax()),
923   kxd_max_(flags.dealias_xz() ? u.Nx()/3-1 : u.kxmax()),
924   kzd_max_(flags.dealias_xz() ? u.Nz()/3-1 : u.kzmax()),
925   Lx_(u.Lx()),
926   Lz_(u.Lz()),
927   a_(u.a()),
928   b_(u.b()),
929   flags_(flags),
930   order_(0),
931   Ninitsteps_(0),
932   nu_(nu),
933   dt_(dt),
934   t_(t0),
935   cfl_(0),
936   dPdxRef_(0),
937   dPdxAct_(0),
938   UbulkRef_(0),
939   UbulkAct_(0),
940   UbulkBase_(0),
941   Ubase_(Ubase),
942   Ubaseyy_(),
943   tmp_(u),
944   uk_(Nyd_,a_,b_,Spectral),
945   vk_(Nyd_,a_,b_,Spectral),
946   wk_(Nyd_,a_,b_,Spectral),
947   Pk_(Nyd_,a_,b_,Spectral),
948   Pyk_(Nyd_,a_,b_,Spectral),
949   Rxk_(Nyd_,a_,b_,Spectral),
950   Ryk_(Nyd_,a_,b_,Spectral),
951   Rzk_(Nyd_,a_,b_,Spectral)
952 {
953   //tmp_.setToZero();
954
955   u.assertState(Spectral, Spectral);
```

```
956   assert(u.vectorDim() == 3);
957
958   // Set the aliased modes to zero
959   Complex zero = 0.0;
960   if (flags_.dealias_xz()) {
961     for (int i=0; i<3; ++i)
962       for (int mx=0; mx<Mx_; ++mx)
963         for (int mz=0; mz<Mz_; ++mz) {
964           if (isAliasedMode(u.kx(mx), u.kz(mz)))
965             for (int ny=0; ny<Nyd_; ++ny)
966               u.cmplx(mx,ny,mz,i) = zero;
967           for (int ny=Nyd_; ny<Ny_; ++ny)
968             u.cmplx(mx,ny,mz,i) = zero;
969         }
970   }
971
972   // Calculate Ubasey_ and related quantities if nonzero. Require that
973   // base flow is quadratic (solves an equilibrium problem).
974   ChebyTransform trans(Ny_);
975   ChebyCoeff Ubasey;
976   if (Ubase_.length() != 0) {
977     Ubase_.makeSpectral(trans);
978     UbulkBase_ = Ubase_.mean();
979     Ubasey   = diff(Ubase_);
980     Ubaseyy_ = diff(Ubasey);
981     Ubase_.makePhysical(trans);
982     Ubasey.makePhysical(trans);
983     // keep Ubaseyy_ as spectral for calc of R in advance() method..
984   }
985   else
986     Ubase_.setState(Physical);
987
988   // These methods require a 9d (3x3) tmp flowfield
989   if (flags_.nonlinearity ==  Alternating ||
990       flags_.nonlinearity ==  Alternating_ ||
991       flags_.nonlinearity ==  Convection ||
992       flags_.nonlinearity ==  Divergence ||
993       flags_.nonlinearity ==  SkewSymmetric)
994     tmp_.resize(u.Nx(), u.Ny(), u.Nz(), 9, u.Lx(), u.Lz(), u.a(), u.b());
995
996   cfl_ = u.CFLfactor(Ubase_);
997   cfl_ *= flags_.dealias_xz() ? 2.0*pi/3.0*dt_ : pi*dt_;
998
999   // Determine actual Ubulk and dPdx from initial data Ubase + u.
1000    ChebyCoeff u00(Ny_,a_,b_,Spectral);
1001    for (int ny=0; ny<Ny_; ++ny)
```

```
1002    u00[ny] = Re(u.cmplx(0,ny,0,0));
1003   ChebyCoeff du00dy = diff(u00);
1004
1005   UbulkAct_ = UbulkBase_ + u00.mean();
1006   dPdxAct_  = nu*(du00dy.eval_b() - du00dy.eval_a())/(b_-a_);
1007   if (Ubase_.length() != 0)
1008     dPdxAct_  += nu*(Ubasey.eval_b() - Ubasey.eval_a())/(b_-a_);
1009
1010   // Set whichever reference value is being held const.
1011   if (flags_.constraint == BulkVelocity)
1012     UbulkRef_ = UbulkAct_;
1013   else
1014     dPdxRef_ = dPdxAct_;
1015 }
```

Here is the call graph for this function:

7.12.4.3 DNSAlgorithm::~DNSAlgorithm () [virtual]

```
871 {}
```

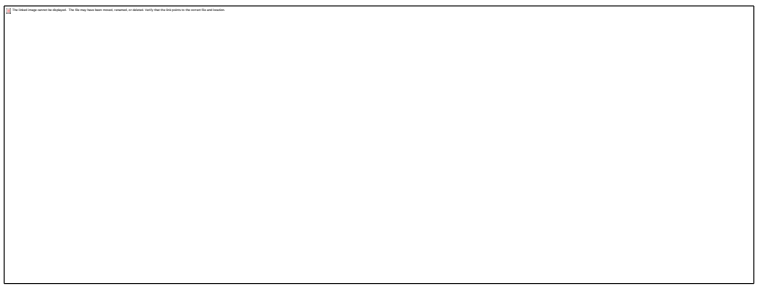
7.12.5 Member Function Documentation

7.12.5.1 Real DNSAlgorithm::a () const

References a_.
Referenced by PPEDNS::advance(), MultistepDNS::advance(), and DNS::reset_dt().

```
1040 {return a_;}
```

Here is the caller graph for this function:



7.12.5.2 virtual void DNSAlgorithm::advance (FlowField & *u*, FlowField & *q*, int *nSteps* = 1) [pure virtual]

Implemented in MultistepDNS, RungeKuttaDNS, CNABstyleDNS, and PPEDNS.
Referenced by DNS::advance().

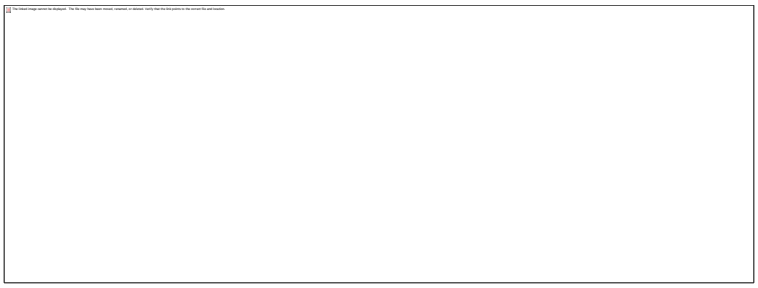
Here is the caller graph for this function:

7.12.5.3 Real DNSAlgorithm::b () const

References b_.
Referenced by PPEDNS::advance(), MultistepDNS::advance(), and DNS::reset_dt().

```
1041 {return b_;}
```

Here is the caller graph for this function:



7.12.5.4 Real DNSAlgorithm::CFL () const

References cfl_.
Referenced by DNS::CFL().
1044 {return cfl_;}

Here is the caller graph for this function:

7.12.5.5 virtual DNSAlgorithm* DNSAlgorithm::clone () const [pure virtual]

Implemented in MultistepDNS, RungeKuttaDNS, CNABstyleDNS, and PPEDNS.
Referenced by DNS::operator=().
Here is the caller graph for this function:

7.12.5.6 Real DNSAlgorithm::dPdx () const

References dPdxAct_.
Referenced by DNS::dPdx().
1046 {return dPdxAct_;}

Here is the caller graph for this function:

7.12.5.7 Real DNSAlgorithm::dPdxRef () const

References dPdxRef_.
Referenced by DNS::dPdxRef().

```
1047 {return dPdxRef_;}
```

Here is the caller graph for this function:

7.12.5.8 Real DNSAlgorithm::dt () const

References dt_.
Referenced by DNS::dt().

```
1042 {return dt_;}
```

Here is the caller graph for this function:

7.12.5.9 const DNSFlags & DNSAlgorithm::flags () const

References flags_.
Referenced by DNS::flags(), and DNS::reset_dt().

```
1052 {return flags_;}
```

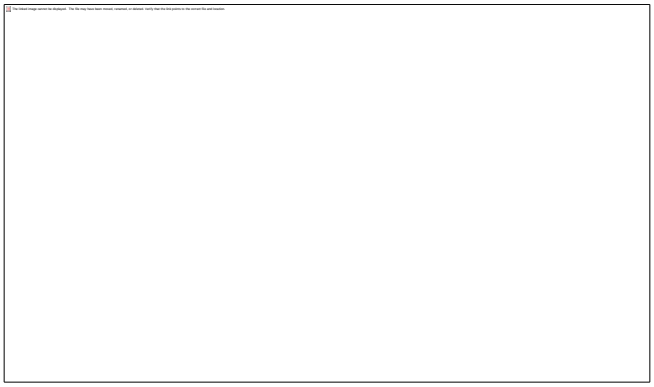
Here is the caller graph for this function:

7.12.5.10 bool DNSAlgorithm::full () const [virtual]

Reimplemented in MultistepDNS, CNABstyleDNS, and PPEDNS.
Referenced by DNS::advance(), DNS::DNS(), DNS::full(), and DNS::reset_dt().

```
1018 {return true;}
```

Here is the caller graph for this function:



7.12.5.11 bool DNSAlgorithm::isAliasedMode (int *kx*, int *kz*) const [protected]

References abs(), kxd_max_, and kzd_max_.

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), and MultistepDNS::reset_dt().

```
1057                                        {
1058   return (abs(kx) > kxd_max_ || (abs(kz) > kzd_max_)) ? true : false;
1059 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.12.5.12 int DNSAlgorithm::kxmaxDealiased () const [protected]

References kxd_max_.

```
1055 {return kxd_max_;}
```

7.12.5.13 int DNSAlgorithm::kzmaxDealiased () const [protected]

References kzd_max_.

```
1056 {return kzd_max_;}
```

7.12.5.14 Real DNSAlgorithm::Lx () const

References Lx_.

Referenced by DNS::reset_dt().

```
1038 {return Lx_;}
```

Here is the caller graph for this function:

7.12.5.15 Real DNSAlgorithm::Lz () const

References Lz_.
Referenced by DNS::reset_dt().

```
1039 {return Lz_;}
```

Here is the caller graph for this function:

7.12.5.16 int DNSAlgorithm::Mx () const

7.12.5.17 int DNSAlgorithm::My () const

7.12.5.18 int DNSAlgorithm::Mz () const

7.12.5.19 int DNSAlgorithm::Ninitsteps () const

References Ninitsteps_.
Referenced by DNS::DNS(), DNS::Ninitsteps(), and DNS::reset_dt().

```
1051 {return Ninitsteps_;}
```

Here is the caller graph for this function:

7.12.5.20 Real DNSAlgorithm::nu () const

References nu_.
Referenced by DNS::reset_dt().

```
1043 {return nu_;}
```

Here is the caller graph for this function:

7.12.5.21 int DNSAlgorithm::Nx () const

References Nx_.
Referenced by DNS::reset_dt().

```
1035 {return Nx_;}
```

Here is the caller graph for this function:

7.12.5.22 int DNSAlgorithm::Ny () const

References Ny_.
Referenced by DNS::reset_dt().

```
1036 {return Ny_;}
```

Here is the caller graph for this function:

7.12.5.23 int DNSAlgorithm::Nz () const

References Nz_.
Referenced by DNS::reset_dt().

```
1037 {return Nz_;}
```

Here is the caller graph for this function:

7.12.5.24 DNSAlgorithm& DNSAlgorithm::operator= (const DNSAlgorithm & *dns*)

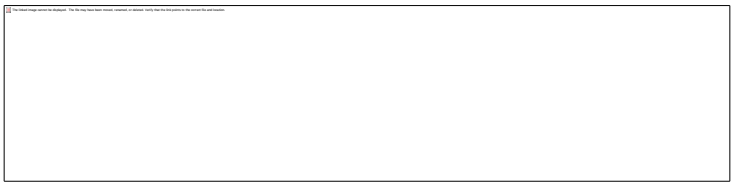
Reimplemented in MultistepDNS, RungeKuttaDNS, CNABstyleDNS, and PPEDNS.

7.12.5.25 int DNSAlgorithm::order () const

References order_.
Referenced by DNS::order(), and PPEDNS::PPEDNS().

```
1050 {return order_;}
```

Here is the caller graph for this function:



7.12.5.26 bool DNSAlgorithm::push (const FlowField & *u*) [virtual]

Reimplemented in MultistepDNS, CNABstyleDNS, and PPEDNS.
Referenced by DNS::advance(), and DNS::push().

```
1017 {return true;}
```

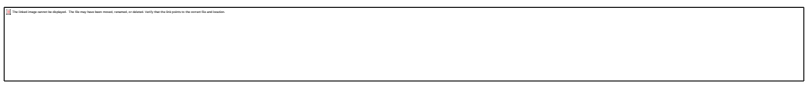
Here is the caller graph for this function:

7.12.5.27 void DNSAlgorithm::reset_dPdx (Real *dPdx*)

References DNSFlags::constraint, dPdxRef_, flags_, PressureGradient, and UbulkRef_.
Referenced by DNS::reset_dPdx().

```
1024                                {
1025   dPdxRef_ = dPdx;
1026   UbulkRef_ = 0.0;
1027   flags_.constraint = PressureGradient;
1028 }
```

Here is the caller graph for this function:

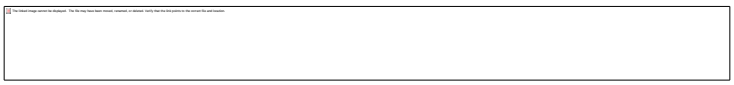


7.12.5.28 virtual void DNSAlgorithm::reset_dt (Real *dt*) [pure virtual]

Implemented in MultistepDNS, RungeKuttaDNS, CNABstyleDNS, and PPEDNS.

Referenced by DNS::reset_dt().

Here is the caller graph for this function:



7.12.5.29 void DNSAlgorithm::reset_time (Real *t*)

References t_.

Referenced by DNS::reset_time().

```
1022 {t_=t;}
```

Here is the caller graph for this function:

7.12.5.30 void DNSAlgorithm::reset_Ubulk (Real *Ubulk*)

References BulkVelocity, DNSFlags::constraint, dPdxRef_, flags_, and UbulkRef_.

Referenced by DNS::reset_Ubulk().

```
1029                                {
1030   dPdxRef_ = 0.0;
1031   UbulkRef_ = Ubulk;
1032   flags_.constraint = BulkVelocity;
1033 }
```

Here is the caller graph for this function:

7.12.5.31 Real DNSAlgorithm::time () const

References t_.

Referenced by DNS::reset_dt(), and DNS::time().

```
1045 {return t_;}
```

Here is the caller graph for this function:

7.12.5.32 TimeStepMethod DNSAlgorithm::timestepping () const

References flags_, and DNSFlags::timestepping.
Referenced by DNS::timestepping().

```
1054 {return flags_.timestepping;}
```

Here is the caller graph for this function:

7.12.5.33 const ChebyCoeff & DNSAlgorithm::Ubase () const

References Ubase_.
Referenced by DNS::reset_dt().

```
1053 {return Ubase_;}
```

Here is the caller graph for this function:

7.12.5.34 Real DNSAlgorithm::Ubulk () const

References UbulkAct_.
Referenced by DNS::Ubulk().

```
1048 {return UbulkAct_;}
```

Here is the caller graph for this function:

7.12.5.35 Real DNSAlgorithm::UbulkRef () const

References UbulkRef_.

Referenced by DNS::UbulkRef().

```
1049 {return UbulkRef_;}
```

Here is the caller graph for this function:

7.12.6 Member Data Documentation

7.12.6.1 Real DNSAlgorithm::a_ [protected]

Referenced by a(), DNSAlgorithm(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), and MultistepDNS::reset_dt().

7.12.6.2 Real DNSAlgorithm::b_ [protected]

Referenced by b(), DNSAlgorithm(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), and MultistepDNS::reset_dt().

7.12.6.3 Real DNSAlgorithm::cfl_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), CFL(), DNSAlgorithm(), MultistepDNS::MultistepDNS(), PPEDNS::PPEDNS(), PPEDNS::push(), MultistepDNS::push(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), and MultistepDNS::reset_dt().

7.12.6.4 Real DNSAlgorithm::dPdxAct_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm(), and dPdx().

7.12.6.5 Real DNSAlgorithm::dPdxRef_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm(), dPdxRef(), reset_dPdx(), and reset_Ubulk().

7.12.6.6 Real DNSAlgorithm::dt_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), CNABstyleDNS::CNABstyleDNS(), DNSAlgorithm(), dt(), MultistepDNS::MultistepDNS(), PPEDNS::PPEDNS(), PPEDNS::push(), CNABstyleDNS::push(), MultistepDNS::push(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), MultistepDNS::reset_dt(), and RungeKuttaDNS::RungeKuttaDNS().

7.12.6.7 DNSFlags DNSAlgorithm::flags_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm(), flags(), MultistepDNS::MultistepDNS(), PPEDNS::PPEDNS(), PPEDNS::push(), CNABstyleDNS::push(), MultistepDNS::push(), reset_dPdx(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), MultistepDNS::reset_dt(), reset_Ubulk(), and timestepping().

7.12.6.8 int DNSAlgorithm::kxd_max_ [protected]

Referenced by isAliasedMode(), and kxmaxDealiased().

7.12.6.9 int DNSAlgorithm::kzd_max_ [protected]

Referenced by isAliasedMode(), and kzmaxDealiased().

7.12.6.10 Real DNSAlgorithm::Lx_ [protected]

Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), Lx(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), and MultistepDNS::reset_dt().

7.12.6.11 Real DNSAlgorithm::Lz_ [protected]

Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), Lz(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), and MultistepDNS::reset_dt().

7.12.6.12 int DNSAlgorithm::Mx_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), CNABstyleDNS::CNABstyleDNS(),

DNSAlgorithm(), MultistepDNS::MultistepDNS(), PPEDNS::PPEDNS(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), MultistepDNS::reset_dt(), RungeKuttaDNS::RungeKuttaDNS(), CNABstyleDNS::~CNABstyleDNS(), MultistepDNS::~MultistepDNS(), PPEDNS::~PPEDNS(), and RungeKuttaDNS::~RungeKuttaDNS().

7.12.6.13 int DNSAlgorithm::Mz_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), CNABstyleDNS::CNABstyleDNS(), DNSAlgorithm(), MultistepDNS::MultistepDNS(), PPEDNS::PPEDNS(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), MultistepDNS::reset_dt(), and RungeKuttaDNS::RungeKuttaDNS().

7.12.6.14 int DNSAlgorithm::Ninitsteps_ [protected]

Referenced by CNABstyleDNS::CNABstyleDNS(), MultistepDNS::MultistepDNS(), Ninitsteps(), PPEDNS::PPEDNS(), PPEDNS::reset_dt(), MultistepDNS::reset_dt(), and RungeKuttaDNS::RungeKuttaDNS().

7.12.6.15 Real DNSAlgorithm::nu_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), nu(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), and MultistepDNS::reset_dt().

7.12.6.16 int DNSAlgorithm::Nx_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), and Nx().

7.12.6.17 int DNSAlgorithm::Ny_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm(), Ny(), and PPEDNS::reset_dt().

7.12.6.18 int DNSAlgorithm::Nyd_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), and MultistepDNS::reset_dt().

7.12.6.19 int DNSAlgorithm::Nz_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), and Nz().

7.12.6.20 int DNSAlgorithm::order_ [protected]

Referenced by PPEDNS::advance(), MultistepDNS::advance(), CNABstyleDNS::CNABstyleDNS(), MultistepDNS::MultistepDNS(), order(), PPEDNS::PPEDNS(), PPEDNS::push(), MultistepDNS::push(), and RungeKuttaDNS::RungeKuttaDNS().

7.12.6.21 ComplexChebyCoeff DNSAlgorithm::Pk_ [protected]

Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), and MultistepDNS::advance().

7.12.6.22 ComplexChebyCoeff DNSAlgorithm::Pyk_ [protected]

Referenced by CNABstyleDNS::advance(), and RungeKuttaDNS::advance().

7.12.6.23 ComplexChebyCoeff DNSAlgorithm::Rxk_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), and MultistepDNS::advance().

7.12.6.24 ComplexChebyCoeff DNSAlgorithm::Ryk_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), and MultistepDNS::advance().

7.12.6.25 ComplexChebyCoeff DNSAlgorithm::Rzk_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), and MultistepDNS::advance().

7.12.6.26 Real DNSAlgorithm::t_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), PPEDNS::push(), CNABstyleDNS::push(), MultistepDNS::push(), reset_time(), and time().

7.12.6.27 FlowField DNSAlgorithm::tmp_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm(), MultistepDNS::MultistepDNS(), PPEDNS::PPEDNS(), PPEDNS::push(),

CNABstyleDNS::push(), MultistepDNS::push(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), and MultistepDNS::reset_dt().

7.12.6.28 ChebyCoeff DNSAlgorithm::Ubase_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm(), MultistepDNS::MultistepDNS(), PPEDNS::PPEDNS(), PPEDNS::push(), CNABstyleDNS::push(), MultistepDNS::push(), and Ubase().

7.12.6.29 ChebyCoeff DNSAlgorithm::Ubaseyy_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), and DNSAlgorithm().

7.12.6.30 Real DNSAlgorithm::UbulkAct_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm(), and Ubulk().

7.12.6.31 Real DNSAlgorithm::UbulkBase_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), and DNSAlgorithm().

7.12.6.32 Real DNSAlgorithm::UbulkRef_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm(), reset_dPdx(), reset_Ubulk(), and UbulkRef().

7.12.6.33 ComplexChebyCoeff DNSAlgorithm::uk_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), and MultistepDNS::advance().

7.12.6.34 ComplexChebyCoeff DNSAlgorithm::vk_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), and MultistepDNS::advance().

7.12.6.35 ComplexChebyCoeff DNSAlgorithm::wk_ [protected]

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), and MultistepDNS::advance().

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

- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.12.6.37

7.13 DNSFlags Class Reference

#include <dns.h>

7.13.1 Public Member Functions

- DNSFlags (MeanConstraint constraint=BulkVelocity, TimeStepMethod timestepping=SBDF3, TimeStepMethod initstepping=SMRK2, NonlinearMethod nonlinearity=SkewSymmetric, Dealiasing dealiasing=NoDealiasing, bool taucorrection=true, Verbosity verbosity=PrintTicks)
- bool dealias_xz () const
- bool dealias_y () const

7.13.2 Public Attributes

- MeanConstraint constraint
- TimeStepMethod timestepping
- TimeStepMethod initstepping
- NonlinearMethod nonlinearity
- Dealiasing dealiasing
- bool taucorrection
- Verbosity verbosity

7.13.3 Constructor & Destructor Documentation

7.13.3.1 DNSFlags::DNSFlags (MeanConstraint *constraint* = BulkVelocity, TimeStepMethod *timestepping* = SBDF3, TimeStepMethod *initstepping* = SMRK2, NonlinearMethod *nonlinearity* = SkewSymmetric, Dealiasing *dealiasing* = NoDealiasing, bool *taucorrection* = true, Verbosity *verbosity* = PrintTicks)

References dealias_y(), nonlinearity, and Rotational.

```
2495   :
2496   constraint(constraint_),
2497   timestepping(timestepping_),
2498   initstepping(initstepping_),
2499   nonlinearity(nonlinearity_),
2500   dealiasing(dealiasing_),
2501   taucorrection(taucorrection_),
2502   verbosity(verbosity_)
2503 {
2504   if (dealias_y() && (nonlinearity != Rotational)) {
2505     cerr << "DNSFlags::DNSFlags: DealiasY and DealiasXYZ work only with\n";
2506     cerr << "Rotational nonlinearity in the current version of channelflow.\n";
2507     cerr << "Setting nonlinearity to Rotational." << endl;
```

```
2508    nonlinearity = Rotational;
2509   }
2510   // maybe should print warnings about initstepping only mattering for SBDF and
CNAB
2511 }
```

Here is the call graph for this function:

7.13.4 Member Function Documentation

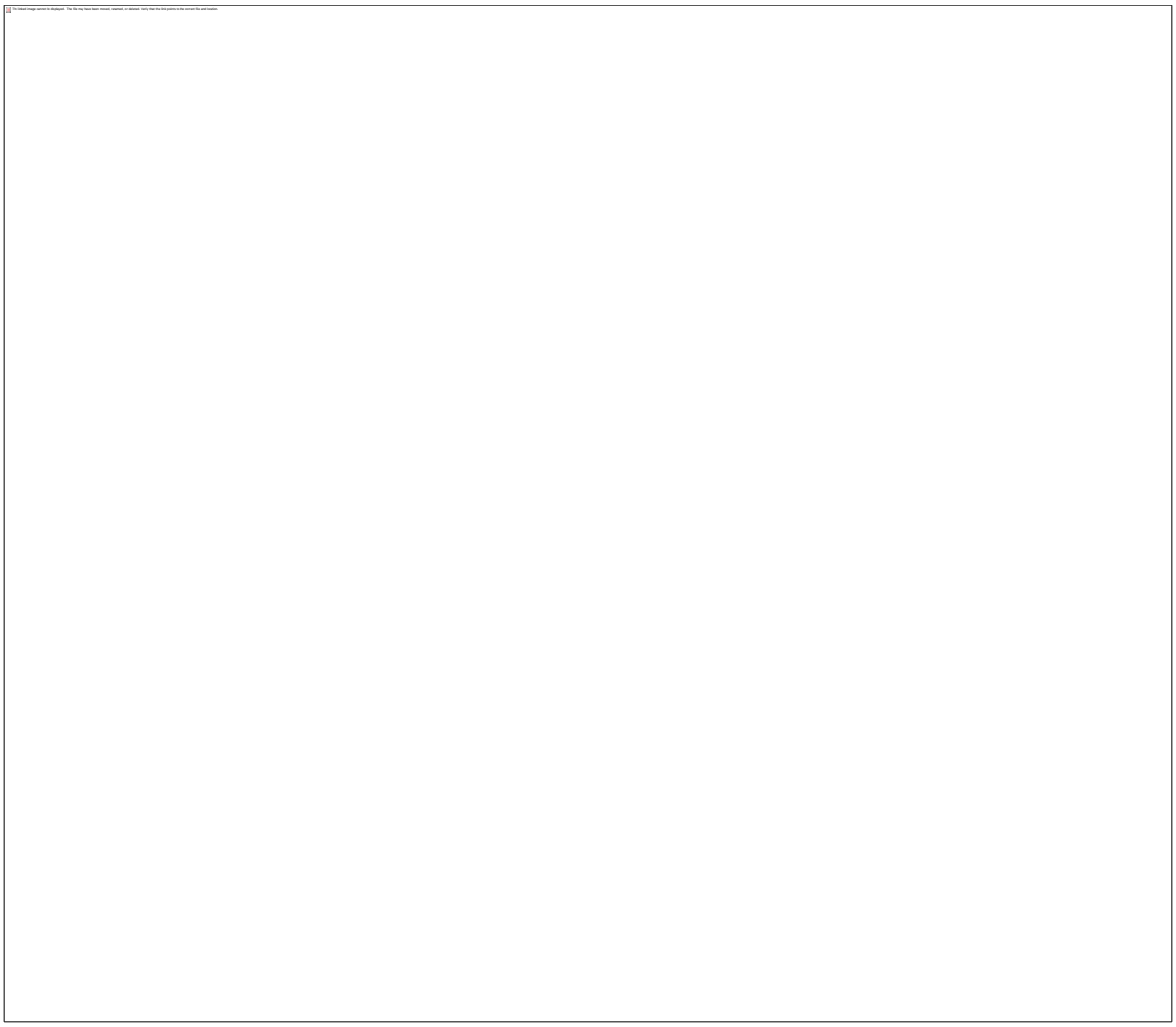
7.13.4.1 bool DNSFlags::dealias_xz () const

References dealiasing, DealiasXYZ, and DealiasXZ.

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm::DNSAlgorithm(), MultistepDNS::MultistepDNS(), PPEDNS::PPEDNS(), PPEDNS::push(), MultistepDNS::push(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), and MultistepDNS::reset_dt().

```
2513                          {
2514   return ((dealiasing == DealiasXZ || dealiasing == DealiasXYZ) ? true:false);
2515 }
```

Here is the caller graph for this function:



7.13.4.2 bool DNSFlags::dealias_y () const

References dealiasing, DealiasXYZ, and DealiasY.

Referenced by DNSFlags().

```
2517                          {
2518   return ((dealiasing == DealiasY || dealiasing == DealiasXYZ) ? true:false);
2519 }
```

Here is the caller graph for this function:

7.13.5 Member Data Documentation

7.13.5.1 MeanConstraint DNSFlags::constraint

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm::DNSAlgorithm(), operator<<(), DNSAlgorithm::reset_dPdx(), and DNSAlgorithm::reset_Ubulk().

7.13.5.2 Dealiasing DNSFlags::dealiasing

Referenced by dealias_xz(), dealias_y(), and operator<<().

7.13.5.3 TimeStepMethod DNSFlags::initstepping

Referenced by DNS::DNS(), operator<<(), and DNS::reset_dt().

7.13.5.4 NonlinearMethod DNSFlags::nonlinearity

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), DNSAlgorithm::DNSAlgorithm(), DNSFlags(), MultistepDNS::MultistepDNS(), operator<<(), PPEDNS::PPEDNS(), PPEDNS::push(), CNABstyleDNS::push(), and MultistepDNS::push().

7.13.5.5 bool DNSFlags::taucorrection

Referenced by operator<<(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), and MultistepDNS::reset_dt().

7.13.5.6 TimeStepMethod DNSFlags::timestepping

Referenced by CNABstyleDNS::CNABstyleDNS(), DNS::DNS(), MultistepDNS::MultistepDNS(), DNS::newAlgorithm(), operator<<(), PPEDNS::PPEDNS(), CNABstyleDNS::reset_dt(), DNS::reset_dt(), RungeKuttaDNS::RungeKuttaDNS(), and DNSAlgorithm::timestepping().

7.13.5.7 Verbosity DNSFlags::verbosity

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), and operator<<().

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

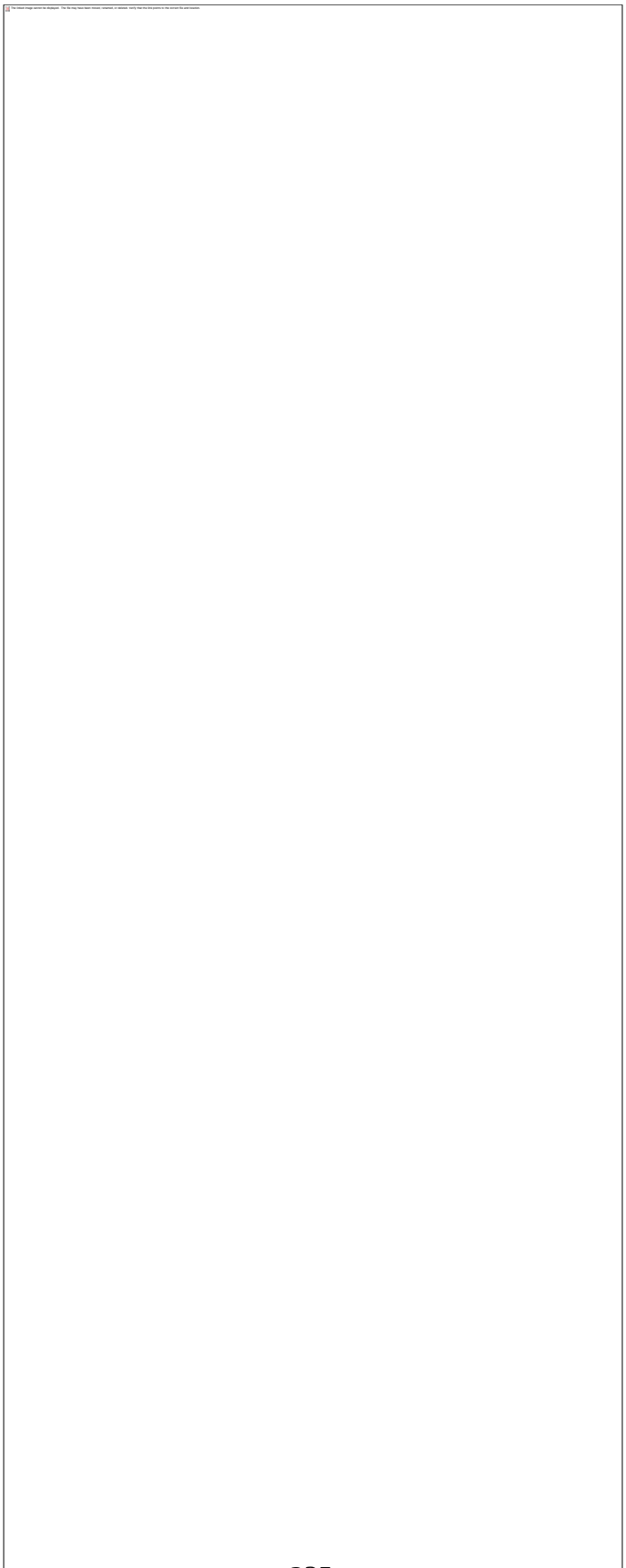
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.13.5.9

7.14 FieldSeries Class Reference

#include <utilfuncs.h>

Collaboration diagram for FieldSeries:



7.14.1 Public Member Functions

- FieldSeries ()
- FieldSeries (int N)
- void push (const FlowField &f, Real t)
- void interpolate (FlowField &f, Real t) const
- bool full () const

7.14.2 Private Attributes

- array< Real > t_
- array< FlowField > f_
- int emptiness_

7.14.3 Constructor & Destructor Documentation

7.14.3.1 FieldSeries::FieldSeries ()

```
132   :
133   t_(0),
134   f_(0),
135   emptiness_(0)
136 {}
```

7.14.3.2 FieldSeries::FieldSeries (int *N*)

```
139   :
140   t_(N),
141   f_(N),
142   emptiness_(N)
143 {}
```

7.14.4 Member Function Documentation

7.14.4.1 bool FieldSeries::full () const

References emptiness_.

Referenced by interpolate().

```
163                         {
164   return (emptiness_ == 0) ? true : false;
165 }
```

Here is the caller graph for this function:

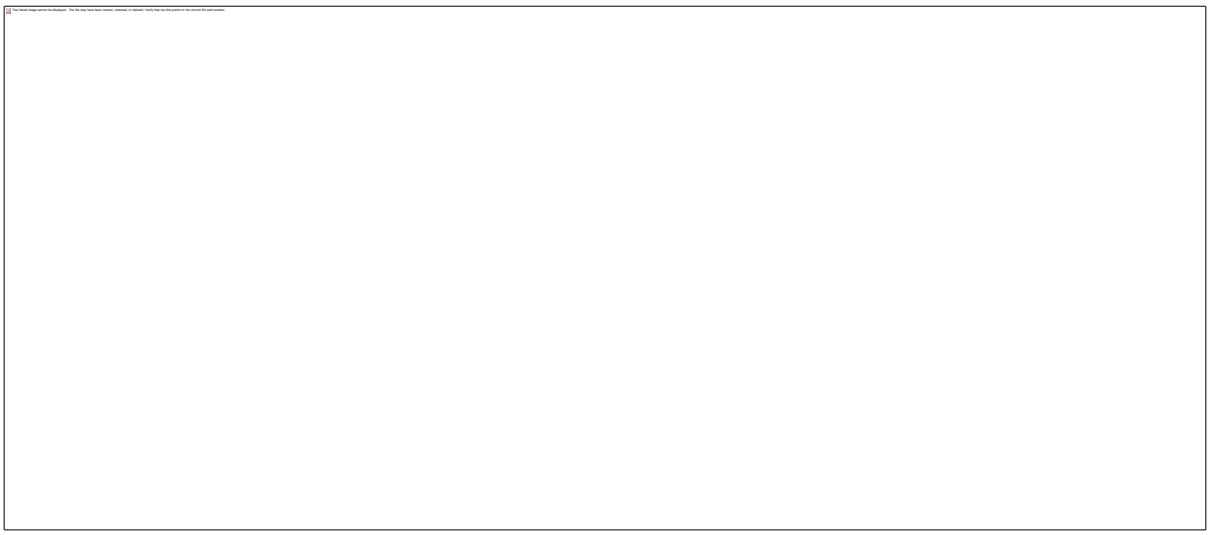
7.14.4.2 void FieldSeries::interpolate (FlowField & *f*, Real *t*) const

References cferror(), FlowField::cmplx(), f_, full(), Im(), array< T >::N(), polyInterp(), Re(), Spectral, and t_.

```
167                                         {
168   if (!(this->full()))
169     cferror("FieldSeries::interpolate(Real t, FlowField& f) : FieldSeries is not completely initialized. Take more time steps before interpolating");
170
171   for (int n=0; n<f_.N(); ++n) {
172     assert(f_[0].congruent(f_[n]));
173     ;
174   }
175
176   if (f_[0].xzstate() == Spectral) {
177     int Mx = f_[0].Mx();
178     int Mz = f_[0].Mz();
179     int Ny = f_[0].Ny();
180     int Nd = f_[0].Nd();
181     int N = f_.N();
182     array<Real> fn(N);
183     for (int i=0; i<Nd; ++i)
184       for (int ny=0; ny<Ny; ++ny)
185         for (int mx=0; mx<Mx; ++mx)
186           for (int mz=0; mz<Mz; ++mz) {
187             for (int n=0; n<N; ++n)
188               fn[n] = Re(f_[n].cmplx(mx,ny,mz,i));
189             Real a = polyInterp(fn,t_,t);
190
191             for (int n=0; n<N; ++n)
192               fn[n] = Im(f_[n].cmplx(mx,ny,mz,i));
193             Real b = polyInterp(fn,t_,t);
194
195             f.cmplx(mx,ny,mz,i) = Complex(a,b);
196           }
197   }
```

```
198   else {
199     int Nx = f_[0].Nx();
200     int Ny = f_[0].Ny();
201     int Nz = f_[0].Nz();
202     int Nd = f_[0].Nd();
203     int N = f_.N();
204     array<Real> fn(N);
205     for (int i=0; i<Nd; ++i)
206       for (int ny=0; ny<Ny; ++ny)
207         for (int nx=0; nx<Nx; ++nx)
208           for (int nz=0; nz<Nz; ++nz) {
209             for (int n=0; n<N; ++n)
210               fn[n] = f_[n](nx,ny,nz,i);
211             f(nx,ny,nz,i) = polyInterp(fn,t_,t);
212           }
213   }
214 }
```

Here is the call graph for this function:



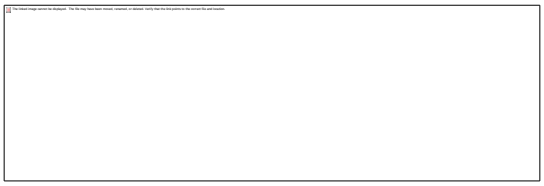
7.14.4.3 void FieldSeries::push (const FlowField & *f*, Real *t*)

References emptiness_, f_, array< T >::N(), swap(), and t_.

```
148                                       {
149   for (int n=f_.N()-1; n>0; --n) {
150     if (f_[n].congruent(f_[n-1]))
151       swap(f_[n], f_[n-1]);
152     else
153       f_[n] = f_[n-1];
154     t_[n] = t_[n-1];
```

```
155   }
156   if (f_.N() > 0) {
157     f_[0] = f;
158     t_[0] = t;
159   }
160   --emptiness_;
161 }
```

Here is the call graph for this function:



7.14.5 Member Data Documentation

7.14.5.1 int FieldSeries::emptiness_ [private]

Referenced by full(), and push().

7.14.5.2 array<FlowField> FieldSeries::f_ [private]

Referenced by interpolate(), and push().

7.14.5.3 array<Real> FieldSeries::t_ [private]

Referenced by interpolate(), and push().

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

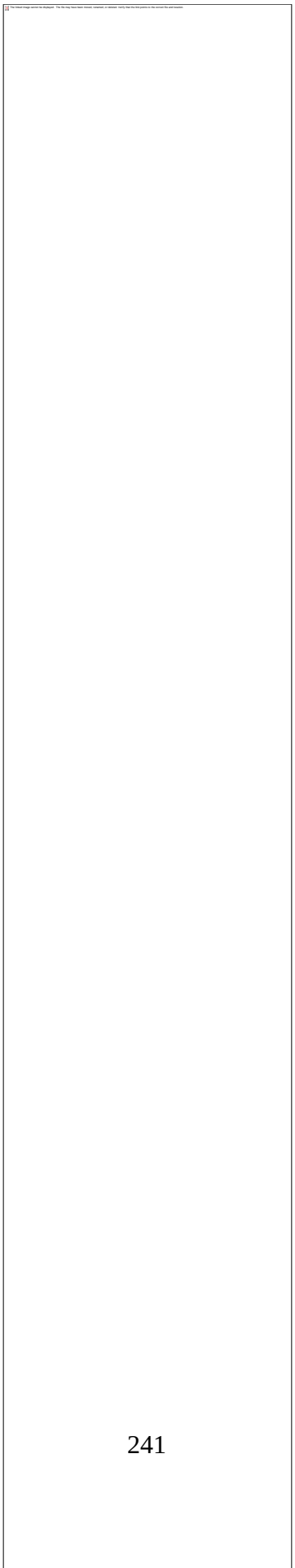
- /_cuda/channelflow-0.9.22/channelflow/utilfuncs.h
- /_cuda/channelflow-0.9.22/channelflow/utilfuncs.cpp

7.14.5.5

7.15 FlowField Class Reference

#include <flowfield.h>

Collaboration diagram for FlowField:


The inkwell image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.15.1 Public Member Functions

- FlowField ()
- FlowField (int Nx, int Ny, int Nz, int Nd, Real Lx, Real Lz, Real a, Real b, fieldstate xzstate=Spectral, fieldstate ystate=Spectral, int fftw_flags=FFTW_ESTIMATE)
- FlowField (int Nx, int Ny, int Nz, int Nd, int tensorOrder, Real Lx, Real Lz, Real a, Real b, fieldstate xzstate=Spectral, fieldstate ystate=Spectral, int fftw_flags=FFTW_ESTIMATE)
- FlowField (const FlowField &u)
- FlowField (const std::string &filebase)
- FlowField (const std::string &filebase, int major, int minor, int update)
- ~FlowField ()
- FlowField & operator= (const FlowField &u)
- bool isNull ()
- void resize (int Nx, int Ny, int Nz, int Nd, Real Lx, Real Lz, Real a, Real b, int fftw_flags=FFTW_ESTIMATE)
- void rescale (Real Lx, Real Lz)
- void interpolate (const FlowField &u)
- void optimizeFFTW (int fftw_flags=FFTW_MEASURE)
- void call_fftw ()
- Real & operator() (int nx, int ny, int nz, int i)
- const Real & operator() (int nx, int ny, int nz, int i) const
- Complex & cmplx (int mx, int my, int mz, int i)
- const Complex & cmplx (int mx, int my, int mz, int i) const
- Complex & coeff (int mx, int my, int mz, int i)
- const Complex & coeff (int mx, int my, int mz, int i) const
- Real & operator() (int nx, int ny, int nz, int i, int j)
- const Real & operator() (int nx, int ny, int nz, int i, int j) const
- Complex & cmplx (int mx, int ny, int mz, int i, int j)
- const Complex & cmplx (int mx, int ny, int mz, int i, int j) const
- Complex & coeff (int mx, int ny, int mz, int i, int j)
- const Complex & coeff (int mx, int ny, int mz, int i, int j) const
- void addProfile (const ChebyCoeff &profile, int i=0)
- void addProfile (const ComplexChebyCoeff &profile, int mx, int mz, int i=0, bool addconj=false)
- void addProfile (const BasisFunc &profile, bool addconj=false)
- ComplexChebyCoeff profile (int mx, int mz, int i) const
- BasisFunc profile (int mx, int mz) const
- void translate (Real delta_x, Real delta_z)
- void realfft_xz ()
- void irealfft_xz ()
- void chebyfft_y ()
- void ichebyfft_y ()
- void makeSpectral_xz ()

- void makePhysical_xz ()
- void makeSpectral_y ()
- void makePhysical_y ()
- void makeSpectral ()
- void makePhysical ()
- void makeState (fieldstate xzstate, fieldstate ystate)
- void setToZero ()
- void perturb (Real magnitude, Real spectralDecay)
- void addPerturbation (int kx, int kz, Real mag, Real spectralDecay)
- void addPerturbations (int kxmax, int kzmax, Real mag, Real spectralDecay)
- void addPerturbations (Real magnitude, Real spectralDecay)
- int numXmodes () const
- int numYmodes () const
- int numZmodes () const
- int numXgridpts () const
- int numYgridpts () const
- int numZgridpts () const
- int vectorDim () const
- int Nx () const
- int Ny () const
- int Nz () const
- int Nd () const
- int Mx () const
- int My () const
- int Mz () const
- int mx (int kx) const
- int mz (int kz) const
- int kx (int mx) const
- int kz (int mz) const
- int kxmax () const
- int kzmax () const
- int kxmin () const
- int kzmin () const
- int kxmaxDealiased () const
- int kzmaxDealiased () const
- bool isAliased (int kx, int kz) const
- Real Lx () const
- Real Ly () const
- Real Lz () const
- Real a () const
- Real b () const
- Real x (int nx) const
- Real y (int ny) const
- Real z (int nz) const

- Vector xgridpts () const
- Vector ygridpts () const
- Vector zgridpts () const
- Complex Dx (int mx) const
- Complex Dz (int mz) const
- Complex Dx (int mx, int n) const
- Complex Dz (int mz, int n) const
- FlowField & operator*= (Real x)
- FlowField & operator*= (Complex x)
- FlowField & operator+= (const ChebyCoeff &U)
- FlowField & operator-= (const ChebyCoeff &U)
- FlowField & operator+= (const ComplexChebyCoeff &U)
- FlowField & operator-= (const ComplexChebyCoeff &U)
- FlowField & operator+= (const BasisFunc &U)
- FlowField & operator-= (const BasisFunc &U)
- FlowField & operator+= (const FlowField &u)
- FlowField & operator-= (const FlowField &u)
- FlowField & operator*= (const FlowField &u)
- bool geomCongruent (const FlowField &f) const
- bool congruent (const FlowField &f) const
- bool congruent (const BasisFunc &phi) const
- void asciiSave (const std::string &filebase) const
- void binarySave (const std::string &filebase) const
- void saveSlice (int k, int i, int nk, const std::string &filebase) const
- void saveProfile (int mx, int mz, const std::string &filebase) const
- void saveProfile (int mx, int mz, const std::string &filebase, const ChebyTransform &t) const
- void saveSpectrum (const std::string &filebase, int i, int ny=-1, bool kxorder=true) const
- void saveSpectrum (const std::string &filebase, bool kxorder=true) const
- void saveDivSpectrum (const std::string &filebase, bool kxorder=true) const
- void print () const
- void dump () const
- void print (int nx, int ny, int nz, int i) const
- Real energy (bool normalize=true) const
- Real energy (int mx, int mz, bool normalize=true) const
- Real dudy_a () const
- Real dudy_b () const
- Real CFLfactor () const
- Real CFLfactor (const ChebyCoeff &Ubase) const
- void setState (fieldstate xz, fieldstate y)
- void assertState (fieldstate xz, fieldstate y) const
- fieldstate xzstate () const
- fieldstate ystate () const
- void setDealiased (bool b)

7.15.2 Private Member Functions

- int flatten (int nx, int ny, int nz, int i) const
- int complex_flatten (int mx, int my, int mz, int i) const
- int flatten (int nx, int ny, int nz, int i, int j) const
- int complex_flatten (int mx, int my, int mz, int i, int j) const
- void fftw_initialize (int fftw_flags=FFTW_ESTIMATE)

7.15.3 Private Attributes

- int Nx_
- int Ny_
- int Nz_
- int Nzpad_
- int Nzpad2_
- int Nd_
- Real Lx_
- Real Lz_
- Real a_
- Real b_
- bool dealiased_
- Real * rdata_
- Complex * cdata_
- Real * scratch_
- fieldstate xzstate_
- fieldstate ystate_
- fftw_plan xz_plan_
- fftw_plan xz_iplan_
- fftw_plan y_plan_

7.15.4 Friends

- void swap (FlowField &f, FlowField &g)

7.15.5 Constructor & Destructor Documentation

7.15.5.1 FlowField::FlowField ()

Referenced by FlowField().

```
42   :
43   Nx_(0),
44   Ny_(0),
45   Nz_(0),
46   Nzpad_(0),
47   Nzpad2_(0),
48   Nd_(0),
```

```
49  Lx_(0),
50  Lz_(0),
51  a_(0),
52  b_(0),
53  dealiased_(false),
54  rdata_(0),
55  cdata_(0),
56  scratch_(0),
57  xzstate_(Spectral),
58  ystate_(Spectral),
59  xz_plan_(0),
60  xz_iplan_(0),
61  y_plan_(0)
62 {}
```

Here is the caller graph for this function:

7.15.5.2 FlowField::FlowField (int *Nx*, int *Ny*, int *Nz*, int *Nd*, Real *Lx*, Real *Lz*, Real *a*, Real *b*, fieldstate *xzstate* = Spectral, fieldstate *ystate* = Spectral, int *fftw_flags* = FFTW_ESTIMATE)

References resize().

```
67  :
68  Nx_(0),
69  Ny_(0),
70  Nz_(0),
71  Nzpad_(0),
72  Nzpad2_(0),
73  Nd_(0),
74  Lx_(0),
75  Lz_(0),
76  a_(0),
77  b_(0),
78  dealiased_(false),
79  rdata_(0),
80  cdata_(0),
81  scratch_(0),
82  xzstate_(xzstate),
83  ystate_(ystate),
84  xz_plan_(0),
85  xz_iplan_(0),
```

```
86   y_plan_(0)
87 {
88   // This isn't in classic C++ initialization style, but it consolidates
89   // all resize and initialization code, and the relative overhead is neglgble.
90   resize(Nx, Ny, Nz, Nd, Lx, Lz, a, b, fftw_flags);
91 }
```

Here is the call graph for this function:

7.15.5.3 FlowField::FlowField (int *Nx*, int *Ny*, int *Nz*, int *Nd*, int *tensorOrder*, Real *Lx*, Real *Lz*, Real *a*, Real *b*, fieldstate *xzstate* = Spectral, fieldstate *ystate* = Spectral, int *fftw_flags* = FFTW_ESTIMATE)

References Nd_, and resize().

```
96   :
97   Nx_(0),
98   Ny_(0),
99   Nz_(0),
100   Nzpad_(0),
101   Nzpad2_(0),
102   Nd_(pow(Nd,tensorOrder)),
103   Lx_(0),
104   Lz_(0),
105   a_(0),
106   b_(0),
107   dealiased_(false),
108   rdata_(0),
109   cdata_(0),
110   scratch_(0),
111   xzstate_(xzstate),
112   ystate_(ystate),
113   xz_plan_(0),
114   xz_iplan_(0),
115   y_plan_(0)
116 {
117   // This isn't in classic C++ initialization style, but it consolidates
118   // all resize and initialization code, and the relative overhead is neglgble.
119   resize(Nx, Ny, Nz, Nd_, Lx, Lz, a, b, fftw_flags);
120 }
```

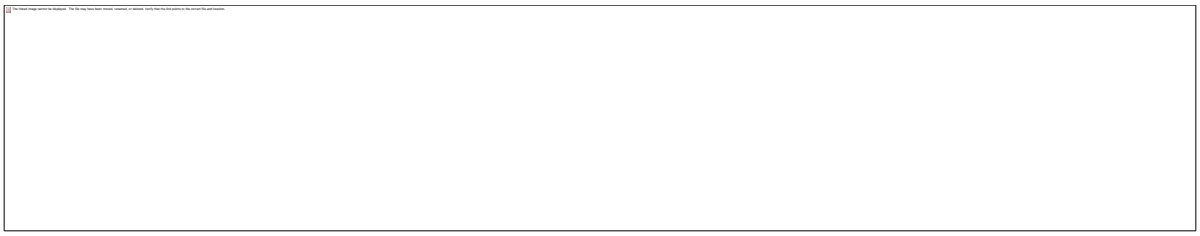
Here is the call graph for this function:

7.15.5.4 FlowField::FlowField (const FlowField & *u*)

References a_, b_, dealiased_, fftw_initialize(), Lx_, Lz_, Nd_, Nx_, Ny_, Nz_, Nzpad_, rdata_, resize(), setState(), xzstate_, and ystate_.

```
123   :
124   Nx_(0),
125   Ny_(0),
126   Nz_(0),
127   Nzpad_(0),
128   Nzpad2_(0),
129   Nd_(0),
130   Lx_(0),
131   Lz_(0),
132   a_(0),
133   b_(0),
134   dealiased_(f.dealiased_),
135   rdata_(0),
136   cdata_(0),
137   scratch_(0),
138   xzstate_(Spectral),
139   ystate_(Spectral),
140   xz_plan_(0),
141   xz_iplan_(0),
142   y_plan_(0)
143 {
144   resize(f.Nx_, f.Ny_, f.Nz_, f.Nd_, f.Lx_, f.Lz_, f.a_, f.b_);
145   setState(f.xzstate_, f.ystate_);
146   dealiased_ = f.dealiased_;
147   int N = Nd_*Ny_*Nx_*Nzpad_;
148   for (int i=0; i<N; ++i)
149     rdata_[i] = f.rdata_[i];
150
151   fftw_initialize(FFTW_ESTIMATE);
152 }
```

Here is the call graph for this function:



7.15.5.5 FlowField::FlowField (const std::string & *filebase*)

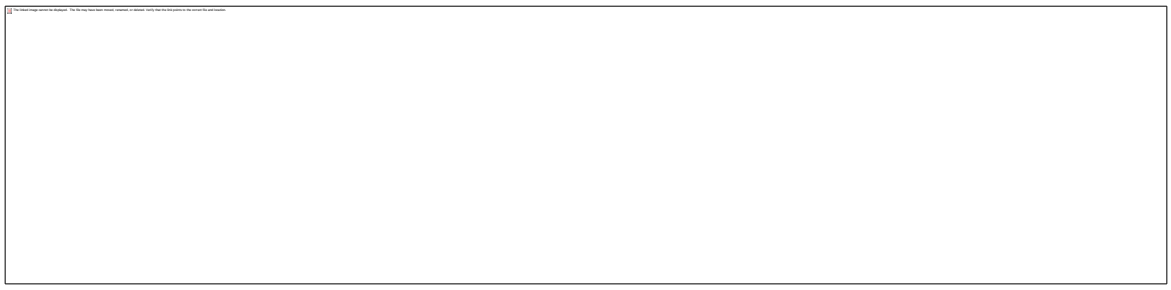
References a_, b_, dealiased_, flatten(), Lx_, Lz_, Nd_, Nx_, Ny_, Nz_, Nzpad_, rdata_, read(), resize(), Spectral, xzstate_, and ystate_.

```
155   :
156   Nx_(0),
157   Ny_(0),
158   Nz_(0),
159   Nzpad_(0),
160   Nzpad2_(0),
161   Nd_(0),
162   Lx_(0.0),
163   Lz_(0.0),
164   a_(0.0),
165   b_(0.0),
166   dealiased_(false),
167   rdata_(0),
168   cdata_(0),
169   scratch_(0),
170   xzstate_(Spectral),
171   ystate_(Spectral),
172   xz_plan_(0),
173   xz_iplan_(0),
174   y_plan_(0)
175 {
176
177   string filename(filebase);
178   filename += string(".ff");
179   ifstream is(filename.c_str(), ios::in | ios::binary);
180
181   // If filebase.ff doesn't work, try filebase alone.
182   if (!is.good()) {
183     is.close();
184     is.open(filebase.c_str(), ios::in | ios::binary);
185     if (is.good())
186       filename = filebase;
187   }
```

```
188  if (!is.good()) {
189    cerr << "FlowField::FlowField(filebase) : can't open file "
190         << filename << " or " << filebase << endl;
191    exit(1);
192  }
193
194  int major_version, minor_version, update_version;
195  read(is, major_version);
196  read(is, minor_version);
197  read(is, update_version);
198  if (major_version != 0 || minor_version != 9) {
199    cerr << "FlowField::FlowField(filebase) : file " << filename
200         << " is not in version > 0.9.16 format.\n"
201         << "Version appears to be " << major_version << '.'
202         << minor_version << '.' << update_version << endl;
203    cerr << "Try using FlowField::FlowField(filebase, chflow_version) constructor.\n";
204    exit(1);
205  }
206
207  read(is, Nx_);
208  read(is, Ny_);
209  read(is, Nz_);
210  read(is, Nd_);
211  read(is, xzstate_);
212  read(is, ystate_);
213  read(is, Lx_);
214  read(is, Lz_);
215  read(is, a_);
216  read(is, b_);
217  read(is, dealiased_);
218  resize(Nx_, Ny_, Nz_, Nd_, Lx_, Lz_, a_, b_);
219
220  // Read data only for non-aliased modes, assume 0 for aliased.
221  if (dealiased_ == true && xzstate_ == Spectral) {
222    int Nxd=2*(Nx_/6);
223    int Nzd=2*(Nz_/3)+1;
224
225    // In innermost loop, array index is (nz + Nzpad2_*(nx + Nx_*(ny + Ny_*i))),
226    // which is the same as the FlowField::flatten function.
227    for (int i=0; i<Nd_; ++i) {
228      for (int ny=0; ny<Ny_; ++ny) {
229
230        for (int nx=0; nx<=Nxd; ++nx) {
231          for (int nz=0; nz<=Nzd; ++nz)
232            read(is, rdata_[flatten(nx,ny,nz,i)]);
233          for (int nz=Nzd+1; nz<Nzpad_; ++nz)
```

```
234           rdata_[flatten(nx,ny,nz,i)] = 0.0;
235         }
236         for (int nx=Nxd+1; nx<=Nxd; ++nx)
237           for (int nz=0; nz<=Nzpad_; ++nz)
238             rdata_[flatten(nx,ny,nz,i)] = 0.0;
239
240         for (int nx=Nx_-Nxd; nx<Nx_; ++nx) {
241           for (int nz=0; nz<=Nzd; ++nz)
242             read(is, rdata_[flatten(nx,ny,nz,i)]);
243           for (int nz=Nzd+1; nz<Nzpad_; ++nz)
244             rdata_[flatten(nx,ny,nz,i)] = 0.0;
245         }
246       }
247     }
248   }
249   else {
250     int N = Nd_*Ny_*Nx_*Nzpad_;
251     for (int i=0; i<N; ++i)
252       read(is, rdata_[i]);
253   }
254 }
```

Here is the call graph for this function:



7.15.5.6 FlowField::FlowField (const std::string & *filebase*, int *major*, int *minor*, int *update*)

References a_, b_, dealiased_, flatten(), FlowField(), Lx_, Lz_, Nd_, Nx_, Ny_, Nz_, Nzpad_, rdata_, resize(), Spectral, xzstate_, and ystate_.

```
257   :
258   Nx_(0),
259   Ny_(0),
260   Nz_(0),
261   Nzpad_(0),
262   Nzpad2_(0),
```

```
263   Nd_(0),
264   Lx_(0.0),
265   Lz_(0.0),
266   a_(0.0),
267   b_(0.0),
268   dealiased_(false),
269   rdata_(0),
270   cdata_(0),
271   scratch_(0),
272   xzstate_(Spectral),
273   ystate_(Spectral),
274   xz_plan_(0),
275   xz_iplan_(0),
276   y_plan_(0)
277 {
278
279   string filename(filebase);
280   filename += string(".ff");
281   ifstream is(filename.c_str(), ios::in | ios::binary);
282
283   // If filebase.ff doesn't work, try filebase alone.
284   if (!is.good())
285     is.open(filebase.c_str(), ios::in | ios::binary);
286
287   if (!is.good()) {
288     cerr << "FlowField::FlowField(filebase, major,minor,update) : can't open file "
289          << filename << " or " << filebase << endl;
290     exit(1);
291   }
292
293   if (major > 0 || minor > 9) {
294     cerr << "FlowField::FlowField(filebase, major,minor,update) :\n"
295          << major << '.' << minor << '.' << update << " is higher than 0.9.x" << endl;
296     exit(1);
297   }
298
299   if (update >= 16) {
300     (*this) = FlowField(filebase); // not efficient but saves code redundancy
301     return;
302   }
303   else {
304     is.read((char*) &Nx_, sizeof(int));
305     is.read((char*) &Ny_, sizeof(int));
306     is.read((char*) &Nz_, sizeof(int));
307     is.read((char*) &Nd_, sizeof(int));
308     is >> xzstate_;
```

```
309   is >> ystate_;
310   is.read((char*) &Lx_, sizeof(Real));
311   is.read((char*) &Lz_, sizeof(Real));
312   is.read((char*) &a_, sizeof(Real));
313   is.read((char*) &b_, sizeof(Real));
314   char s;
315   is.get(s);
316   dealiased_ = (s=='1') ? true : false;
317
318   resize(Nx_, Ny_, Nz_, Nd_, Lx_, Lz_, a_, b_);
319
320   // Read data only for non-aliased modes, assume 0 for aliased.
321   if (dealiased_ == true && xzstate_ == Spectral) {
322     int Nxd=2*(Nx_/6);
323     int Nzd=2*(Nz_/3)+1;
324
325     // In innermost loop, array index is (nz + Nzpad2_*(nx + Nx_*(ny+Ny_*i)))
326     // which is the same as the FlowField::flatten function.
327     for (int i=0; i<Nd_; ++i) {
328       for (int ny=0; ny<Ny_; ++ny) {
329
330         for (int nx=0; nx<=Nxd; ++nx) {
331           for (int nz=0; nz<=Nzd; ++nz)
332             is.read((char*) (rdata_ + flatten(nx,ny,nz,i)), sizeof(Real));
333           for (int nz=Nzd+1; nz<Nzpad_; ++nz)
334             rdata_[flatten(nx,ny,nz,i)] = 0.0;
335         }
336         for (int nx=Nxd+1; nx<=Nxd; ++nx)
337           for (int nz=0; nz<=Nzpad_; ++nz)
338             rdata_[flatten(nx,ny,nz,i)] = 0.0;
339
340         for (int nx=Nx_-Nxd; nx<Nx_; ++nx) {
341           for (int nz=0; nz<=Nzd; ++nz)
342             is.read((char*) (rdata_ + flatten(nx,ny,nz,i)), sizeof(Real));
343           for (int nz=Nzd+1; nz<Nzpad_; ++nz)
344             rdata_[flatten(nx,ny,nz,i)] = 0.0;
345         }
346       }
347     }
348   }
349   else {
350     int N = Nd_*Ny_*Nx_*Nzpad_;
351     for (int i=0; i<N; ++i)
352       is.read((char*) (rdata_ + i), sizeof(Real));
353   }
354 }
```

```
355 }
```

Here is the call graph for this function:

7.15.5.7 FlowField::~FlowField ()

References rdata_, scratch_, xz_iplan_, xz_plan_, and y_plan_.

```
389                     {
390   if (scratch_) fftw_free(scratch_);
391   if (rdata_) fftw_free(rdata_);
392   if (y_plan_) fftw_destroy_plan(y_plan_);
393   if (xz_plan_) fftw_destroy_plan(xz_plan_);
394   if (xz_iplan_) fftw_destroy_plan(xz_iplan_);
395 }
```

7.15.6 Member Function Documentation

7.15.6.1 Real FlowField::a () const [inline]

References a_.

Referenced by bcDist2(), bcNorm2(), PoissonSolver::congruent(), convectionNL(), cross(), curl(), diff(), div(), divergenceNL(), DNSAlgorithm::DNSAlgorithm(), dot(), energy(), PoissonSolver::geomCongruent(), grad(), interpolate(), L2Dist2(), L2InnerProduct(), L2Norm2(), lapl(), linearizedNL(), norm(), norm2(), outer(), PPEDNS::PPEDNS(), PPEDNS::push(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), PressureSolver::verify(), xdiff(), ydiff(), and zdiff().

```
430 {return a_;}
```

Here is the caller graph for this function:

7.15.6.2 void FlowField::addPerturbation (int *kx*, int *kz*, Real *mag*, Real *spectralDecay*)

References a_, assertState(), b_, cmplx(), Lx_, Lz_, mx(), mz(), Nd_, Ny_, randomProfile(), and Spectral.

Referenced by addPerturbations().

```
1062                                                {
1063   assertState(Spectral, Spectral);
1064   if (mag == 0.0)
1065     return;
1066
1067   // Add a dive-free perturbation to the base flow.
1068   ComplexChebyCoeff u(Ny_, a_, b_, Spectral);
1069   ComplexChebyCoeff v(Ny_, a_, b_, Spectral);
1070   ComplexChebyCoeff w(Ny_, a_, b_, Spectral);
1071   randomProfile(u,v,w, kx, kz, Lx_, Lz_, mag, decay);
1072   int m_x = mx(kx);
1073   int m_z = mz(kz);
1074   for (int ny=0; ny<Ny_; ++ny) {
1075     cmplx(m_x, ny, m_z, 0) += u[ny];
1076     cmplx(m_x, ny, m_z, 1) += v[ny];
1077     cmplx(m_x, ny, m_z, 2) += w[ny];
1078   }
1079   if (kz==0 && kx!=0) {
1080     int m_x = mx(-kx);
1081     int m_z = mz(0); // -kz=0
1082     for (int i=0; i<Nd_; ++i)
1083       for (int ny=0; ny<Ny_; ++ny) {
1084         cmplx(m_x, ny, m_z, 0) += conj(u[ny]);
1085         cmplx(m_x, ny, m_z, 1) += conj(v[ny]);
1086         cmplx(m_x, ny, m_z, 2) += conj(w[ny]);
1087       }
1088   }
1089   return;
1090 }
```

Here is the call graph for this function:

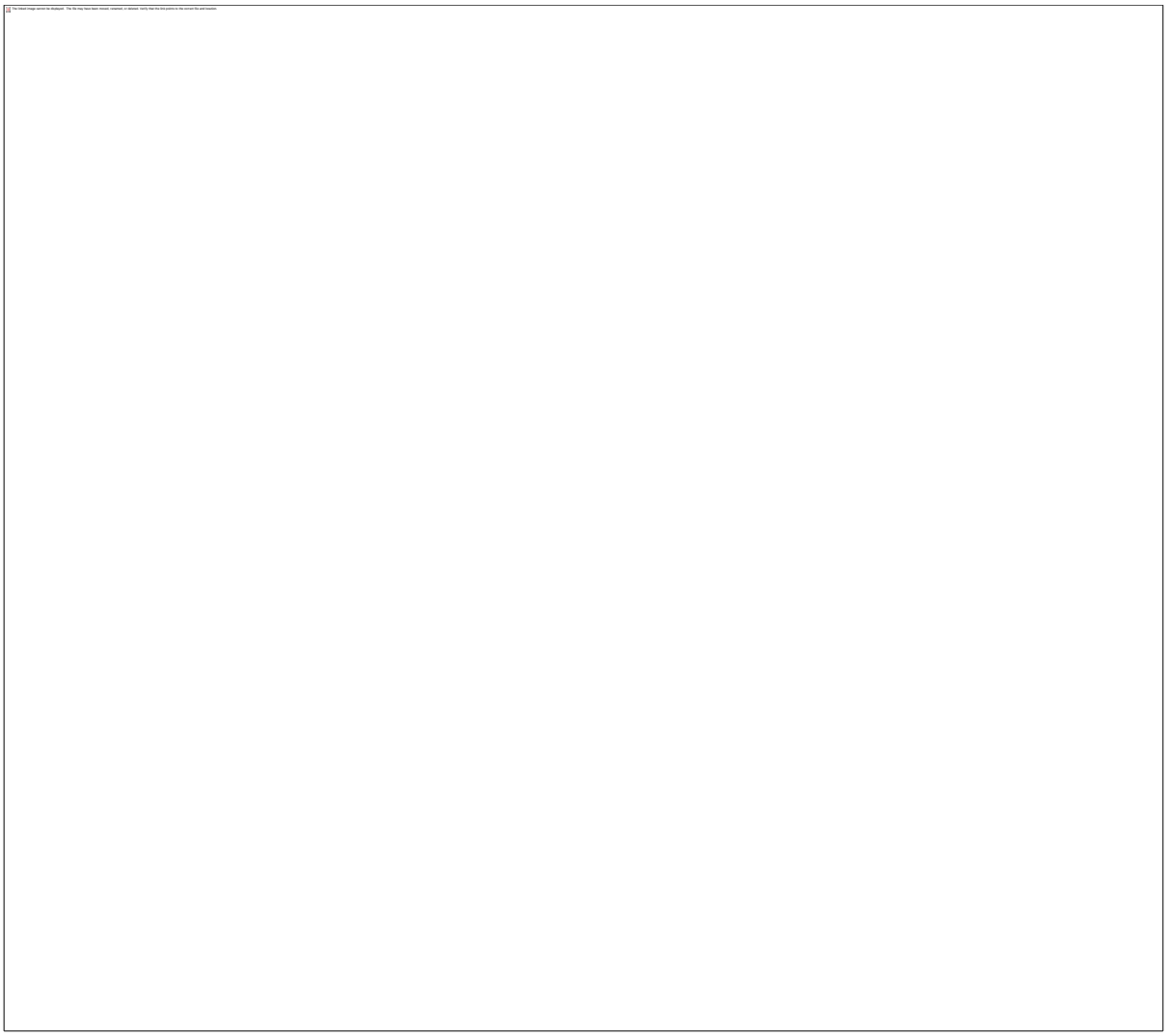
Here is the caller graph for this function:

7.15.6.3 void FlowField::addPerturbations (Real *magnitude*, Real *spectralDecay*)

References abs(), addPerturbation(), assertState(), kx(), kz(), makePhysical(), makeSpectral(), Mx(), mx(), Mz(), mz(), norm(), pow(), and Spectral.

```
1119                                        {
1120   assertState(Spectral, Spectral);
1121   if (mag == 0.0)
1122     return;
1123
1124   // Add a div-free perturbation to the base flow.
1125   for (int mx=0; mx<Mx(); ++mx) {
1126     int kx_ = kx(mx);
1127     for (int mz=0; mz<Mz(); ++mz) {
1128       int kz_ = kz(mz);
1129       Real norm = pow(decay, 2*(abs(kx_) + abs(kz_)));
1130       addPerturbation(kx_, kz_, mag*norm, decay);
1131     }
1132   }
1133   makePhysical();
1134   makeSpectral();
1135   return;
1136 }
```

Here is the call graph for this function:



7.15.6.4 void FlowField::addPerturbations (int *kxmax*, int *kzmax*, Real *mag*, Real *spectralDecay*)

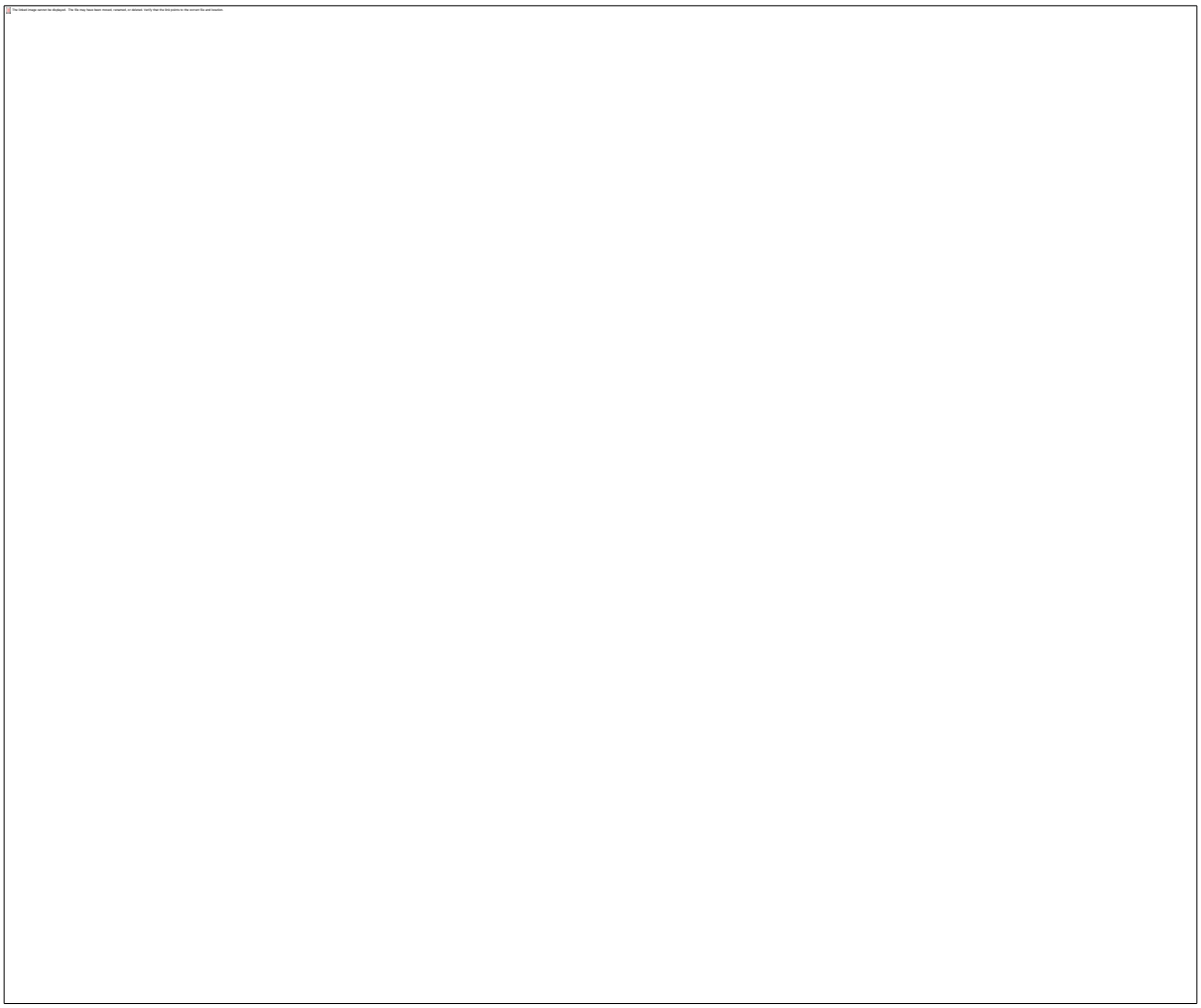
References abs(), addPerturbation(), assertState(), Greater(), kx(), kxmax(), kxmin(), kz(), kzmax(), lesser(), Lx_, Lz_, makePhysical(), makeSpectral(), norm(), pi, pow(), and Spectral.

Referenced by perturb().

```
1096                                                {
1097   assertState(Spectral, Spectral);
1098   if (mag == 0.0)
1099     return;
1100
1101   int Kxmin = Greater(-Kx, kxmin());
1102   int Kxmax = lesser(Kx, kxmax()-1);
1103   int Kzmax = lesser(Kz, kzmax()-1);
```

```
1104
1105   // Add a div-free perturbation to the base flow.
1106   for (int kx=Kxmin; kx<=Kxmax; ++kx)
1107     for (int kz=0; kz<=Kzmax; ++kz) {
1108       //Real norm = pow(10.0, -(abs(2*pi*kx/Lx_) + abs(2*pi*kz/Lz_)));
1109       //Real norm = pow(decay, 2*(abs(kx) + abs(kz)));
1110       Real norm = pow(decay, abs(2*pi*kx/Lx_) + abs(2*pi*kz/Lz_));
1111       if (!(kx==0 && kz==0))
1112         addPerturbation(kx, kz, mag*norm, decay);
1113     }
1114   makePhysical();
1115   makeSpectral();
1116   return;
1117 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.15.6.5 void FlowField::addProfile (const BasisFunc & *profile*, bool *addconj* = false)

References cmplx(), BasisFunc::kx(), BasisFunc::kz(), mx(), mz(), BasisFunc::Nd(), Nd_, Ny_, Spectral, BasisFunc::state(), xzstate_, and ystate_.

```
779                                                 {
780   assert(xzstate_ == Spectral);
781   assert(ystate_ == profile.state());
782   assert(Nd_ == profile.Nd());
783   int m_x = mx(profile.kx());
784   int m_z = mz(profile.kz());
785   for (int i=0; i<Nd_; ++i)
786     for (int ny=0; ny<Ny_; ++ny)
787       cmplx(m_x, ny, m_z, i) += profile[i][ny];
788 
789   if (addconj && profile.kx() !=0 && profile.kz() ==0) {
790     m_x = mx(-profile.kx());
791     for (int i=0; i<Nd_; ++i)
792       for (int ny=0; ny<Ny_; ++ny)
793         cmplx(m_x, ny, m_z, i) += conj(profile[i][ny]);
794   }
795 }
```

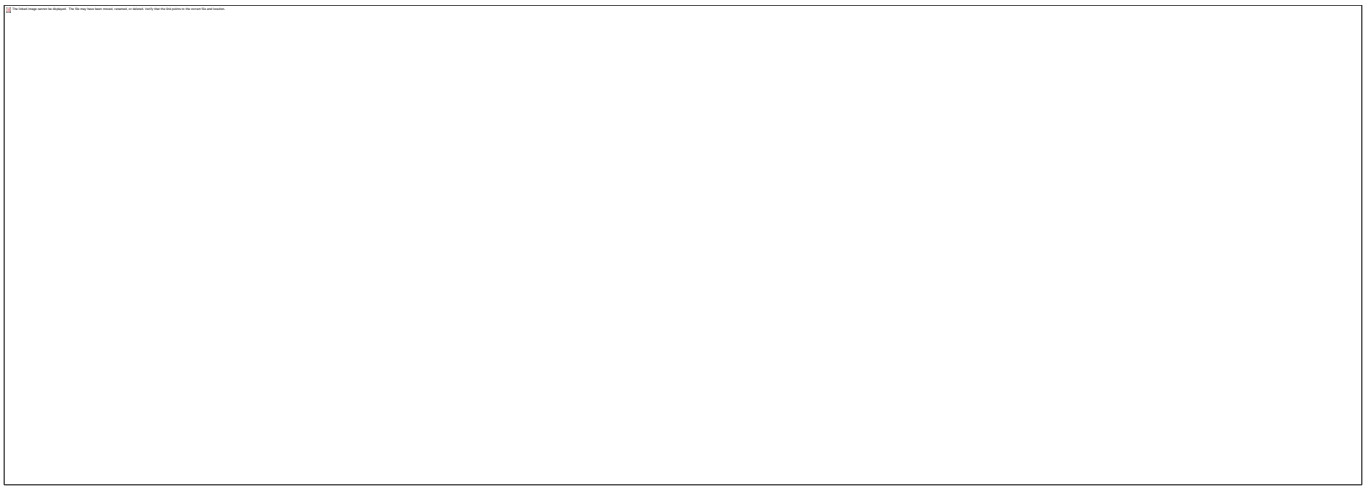
Here is the call graph for this function:

7.15.6.6 void FlowField::addProfile (const ComplexChebyCoeff & *profile*, int *mx*, int *mz*, int *i* = 0, bool *addconj* = false)

References cmplx(), kx(), kz(), mx(), Ny_, Spectral, ComplexChebyCoeff::state(), xzstate_, and ystate_.

```
742                                                 {
743   assert(xzstate_== Spectral);
744   assert(ystate_ == profile.state());
745   int k_x = kx(m_x);
746   int k_z = kz(m_z);
747   for (int ny=0; ny<Ny_; ++ny)
748     cmplx(m_x, ny, m_z, i) += profile[ny];
749   if (addconj && k_x !=0 && k_z ==0) {
750     m_x = mx(-k_x);
751     for (int ny=0; ny<Ny_; ++ny)
752       cmplx(m_x,ny,m_z, i) += conj(profile[ny]);
753   }
754 }
```

Here is the call graph for this function:



7.15.6.7 void FlowField::addProfile (const ChebyCoeff & *profile*, int *i* = 0)

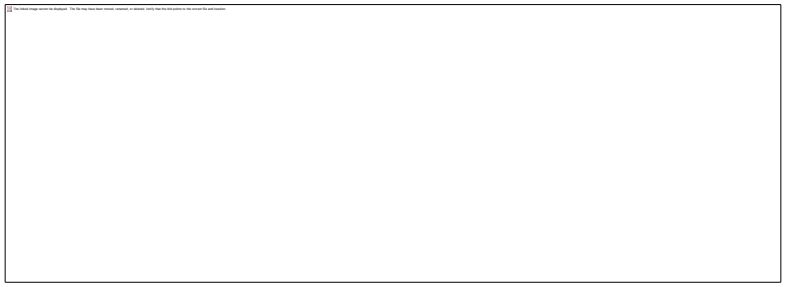
References cmplx(), kx(), kz(), mx(), mz(), Ny_, Spectral, ChebyCoeff::state(), xzstate_, and ystate_.

Referenced by changeBaseFlow(), operator+=(), and operator-=().

```
730                                                {
731   assert(xzstate_ == Spectral);
732   assert(ystate_  == profile.state());
733   int kx=0;
734   int kz=0;
735   int m_x = mx(kx);
736   int m_z = mz(kz);
737   for (int ny=0; ny<Ny_; ++ny)
738     cmplx(m_x, ny, m_z, i) += Complex(profile[ny], 0.0);
739 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



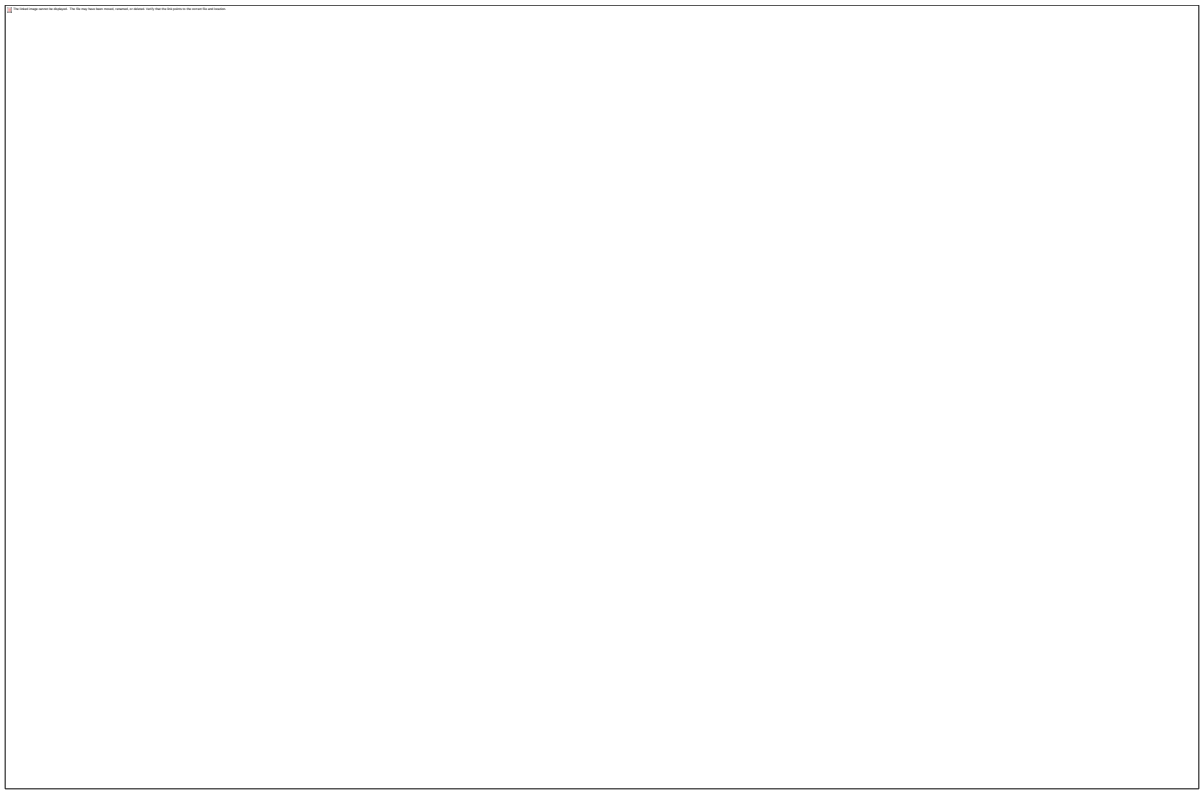
7.15.6.8 void FlowField::asciiSave (const std::string & *filebase*) const

References a_, b_, cmplx(), Im(), Lx_, Lz_, mx(), Mx(), My(), mz(), Mz(), Nd_, Nx_, Ny_, Nz_, Physical, Re(), REAL_DIGITS, REAL_IOWIDTH, xzstate_, and ystate_.

```
1461                                        {
1462   string filename(filebase);
1463   filename += ".asc";
1464
1465   ofstream os(filename.c_str());
1466   os << scientific << setprecision(REAL_DIGITS);
1467
1468   const char s = ' ';
1469   const char nl = '\n';
1470   const int w = REAL_IOWIDTH;
1471   os << "% Channelflow FlowField data" << nl;
1472   os << "% xzstate == " << xzstate_ << nl;
1473   os << "%  ystate == " << ystate_ << nl;
```

```
1474   os << "% Nx Ny Nz Nd == " << Nx_ <<s<< Ny_ <<s<< Nz_ <<s<< Nd_ << "
gridpoints\n";
1475   os << "% Mx My Mz Nd == " << Mx() <<s<< My() <<s<< Mz() <<s<< Nd_ << "
spectral modes\n";
1476   os << "% Lx Lz == " << setw(w) << Lx_ <<s<< setw(w) << Lz_ << nl;
1477   os << "% Lx Lz == " << setw(w) << Lx_ <<s<< setw(w) << Lz_ << nl;
1478   os << "% a  b  == " << setw(w) << a_ <<s<< setw(w) << b_ << nl;
1479   os << "% loop order:\n";
1480   os << "%   for (int i=0; i<Nd; ++i)\n";
1481   os << "%     for(long ny=0; ny<Ny; ++ny) // note: Ny == My\n";
1482   if (xzstate_ == Physical) {
1483     os << "%       for (int nx=0; nx<Nx; ++nx)\n";
1484     os << "%         for (int nz=0; nz<Nz; ++nz)\n";
1485     os << "%           os << f(nx, ny, nz, i) << newline;\n";
1486   }
1487   else {
1488     os << "%       for (int mx=0; mx<Mx; ++mx)\n";
1489     os << "%         for (int mz=0; mz<Mz; ++mz)\n";
1490     os << "%           os << Re(f.cmplx(mx, ny, mz, i) << ' ' << Im(f.cmplx(mx, ny, mz, i)
<< newline;\n";
1491   }
1492   if (xzstate_ == Physical) {
1493     for (int i=0; i<Nd_; ++i)
1494       for(long ny=0; ny<Ny_; ++ny)
1495         for (int nx=0; nx<Nx_; ++nx)
1496           for (int nz=0; nz<Nz_; ++nz)
1497             os << setw(w) << (*this)(nx, ny, nz, i) << nl;
1498   }
1499   else {
1500     for (int i=0; i<Nd_; ++i)
1501       for(long ny=0; ny<Ny_; ++ny)
1502         for (int mx=0; mx<Mx(); ++mx)
1503           for (int mz=0; mz<Mz(); ++mz)
1504             os << setw(w) << Re(cmplx(mx, ny, mz, i)) << s
1505                << setw(w) << Im(cmplx(mx, ny, mz, i)) << nl;
1506   }
1507   os.close();
1508 }
```

Here is the call graph for this function:



7.15.6.9 void FlowField::assertState (fieldstate *xz*, fieldstate *y*) const

References xzstate_, and ystate_.

Referenced by addPerturbation(), addPerturbations(), DNSAlgorithm::DNSAlgorithm(), rescale(), PoissonSolver::solve(), and PoissonSolver::verify().

```
1672                                                {
1673   assert(xzstate_ == xz && ystate_ == y);
1674 }
```

Here is the caller graph for this function:

7.15.6.10 Real FlowField::b () const [inline]

References b_.

Referenced by bcDist2(), bcNorm2(), PoissonSolver::congruent(), convectionNL(), cross(), curl(), diff(), div(), divergenceNL(), DNSAlgorithm::DNSAlgorithm(), dot(), energy(), PoissonSolver::geomCongruent(), grad(), interpolate(), L2Dist2(), L2InnerProduct(), L2Norm2(), lapl(), linearizedNL(), norm(), norm2(), outer(), PPEDNS::PPEDNS(), PPEDNS::push(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), PressureSolver::verify(), xdiff(), ydiff(), and zdiff().

```
431 {return b_;}
```

Here is the caller graph for this function:

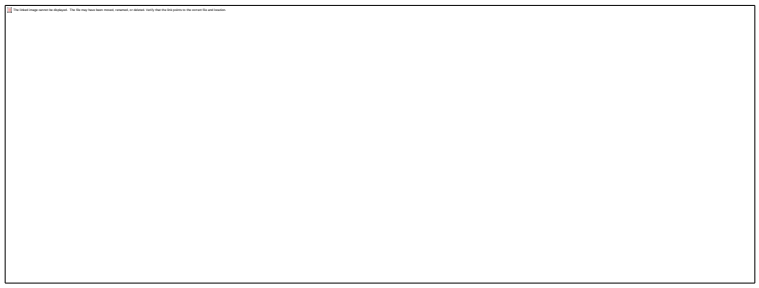
7.15.6.11 void FlowField::binarySave (const std::string & *filebase*) const

References a_, b_, channelflowVersion(), dealiased_, flatten(), Lx_, Lz_, Nd_, Nx_, Ny_, Nz_, Nzpad_, rdata_, Spectral, write(), xzstate_, and ystate_.

```
1510                                        {
1511   string filename(filebase);
1512   filename += string(".ff"); // "flow field"
1513   ofstream os(filename.c_str(), ios::out | ios::binary);
1514   if (!os.good()) {
1515     cerr << "FlowField::binarySave(filebase) : can't open file " << filename << endl;
1516     abort();
1517   }
1518 
1519   int major;
1520   int minor;
1521   int update;
1522   channelflowVersion(major, minor, update);
1523 
1524   write(os, major);
1525   write(os, minor);
1526   write(os, update);
1527   write(os, Nx_);
1528   write(os, Ny_);
1529   write(os, Nz_);
1530   write(os, Nd_);
1531   write(os, xzstate_);
1532   write(os, ystate_);
1533   write(os, Lx_);
1534   write(os, Lz_);
1535   write(os, a_);
1536   write(os, b_);
1537   write(os, dealiased_);
1538 
1539   // Write data only for non-aliased modes.
1540   if (dealiased_ && xzstate_ == Spectral) {
1541     int Nxd=2*(Nx_/6);
1542     int Nzd=2*(Nz_/3)+1;
1543 
1544     // In innermost loop, array index is (nz + Nzpad2_*(nx + Nx_*(ny + Ny_*i))),
1545     // which is the same as the FlowField::flatten function.
1546     for (int i=0; i<Nd_; ++i) {
1547       for (int ny=0; ny<Ny_; ++ny) {
1548 
1549         for (int nx=0; nx<=Nxd; ++nx) {
```

```
1550         for (int nz=0; nz<=Nzd; ++nz)
1551           write(os, rdata_[flatten(nx,ny,nz,i)]);
1552       }
1553       for (int nx=Nx_-Nxd; nx<Nx_; ++nx) {
1554         for (int nz=0; nz<=Nzd; ++nz)
1555           write(os, rdata_[flatten(nx,ny,nz,i)]);
1556       }
1557     }
1558   }
1559 }
1560 else {
1561   int Ntotal = Nd_* Nx_ * Ny_ * Nzpad_;
1562   for (int i=0; i<Ntotal; ++i)
1563     write(os, rdata_[i]);
1564 }
1565 }
```

Here is the call graph for this function:



7.15.6.12 void FlowField::call_fftw ()

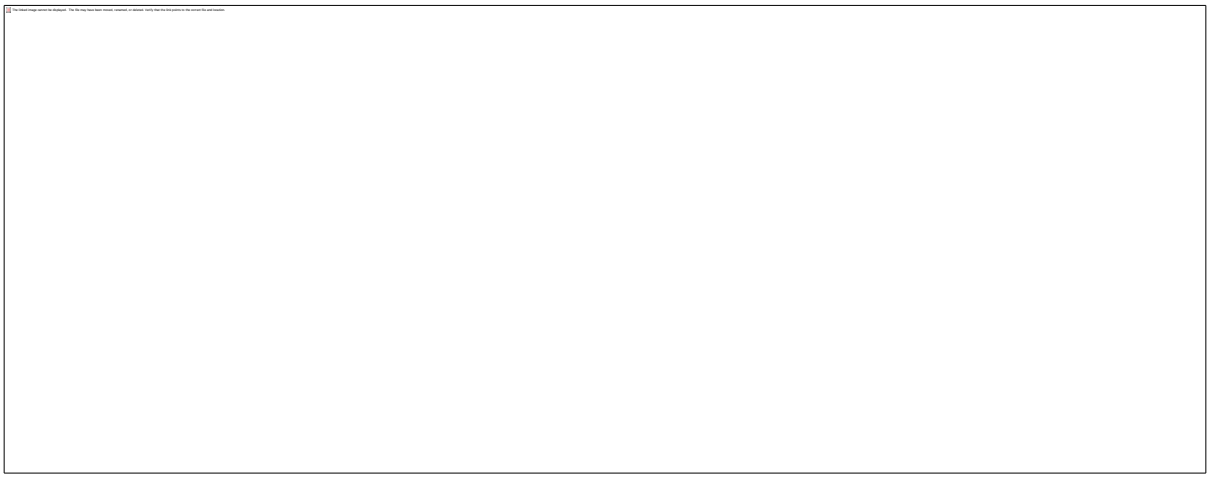
7.15.6.13 Real FlowField::CFLfactor (const ChebyCoeff & *Ubase*) const

References a_, b_, CFLfactor(), chebypoints(), Greater(), Lx_, Lz_, ChebyCoeff::N(), Nd_, Nx_, Ny_, Nz_, Physical, ChebyCoeff::state(), xzstate(), xzstate_, y(), ystate(), and ystate_.

```
1627                                          {
1628 if (Ubase.N() == 0)
1629   return CFLfactor();
1630
1631 assert(Nd_ == 3);
1632 assert(Ubase.state() == Physical);
1633 FlowField& u = (FlowField&) *this;
1634 fieldstate xzstate = xzstate_;
1635 fieldstate ystate = ystate_;
1636 u.makePhysical();
1637 Vector y = chebypoints(Ny_, a_, b_);
```

```
1638   Real cfl = 0.0;
1639   Vector dx(3);
1640   dx[0] = Lx_/Nx_;
1641   dx[2] = Lz_/Nz_;
1642   for (int i=0; i<3; ++i)
1643     for (int ny=0; ny<Ny_; ++ny) {
1644       dx[1] = (ny==0 || ny==Ny_-1) ? y[0]-y[1] : (y[ny-1]-y[ny+1])/2.0;
1645       Real U = (i==0) ? Ubase(ny) : 0;
1646       for (int nx=0; nx<Nx_; ++nx)
1647         for (int nz=0; nz<Nz_; ++nz)
1648           cfl = Greater(cfl, (u(nx,ny,nz,i)+U)/dx[i]);
1649     }
1650   u.makeState(xzstate,ystate);
1651
1652   return cfl;
1653 }
```

Here is the call graph for this function:



7.15.6.14 Real FlowField::CFLfactor () const

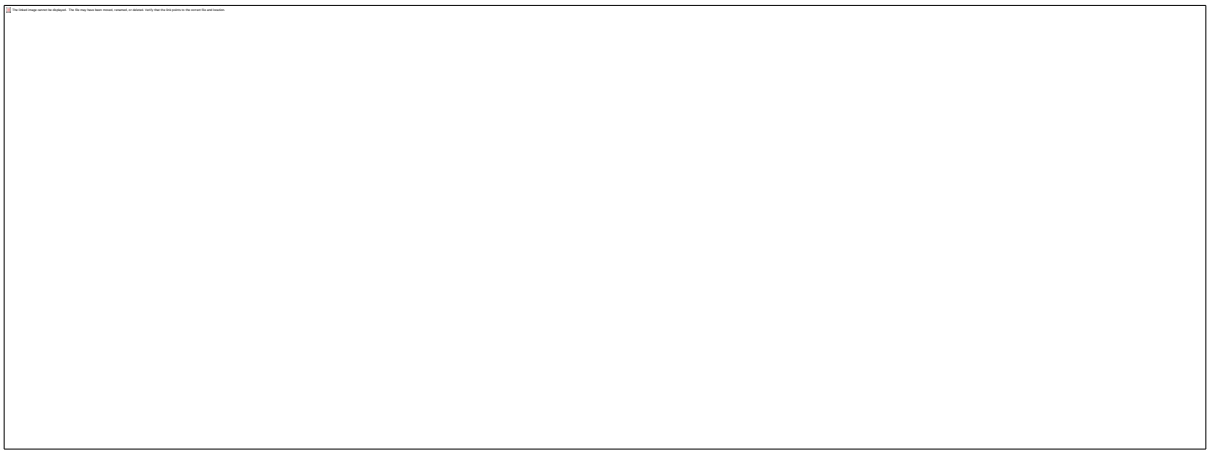
References a_, abs(), b_, chebypoints(), Greater(), Lx_, Lz_, makePhysical(), makeState(), Nd_, Nx_, Ny_, Nz_, xzstate(), xzstate_, y(), ystate(), and ystate_.

Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), CFLfactor(), and DNSAlgorithm::DNSAlgorithm().

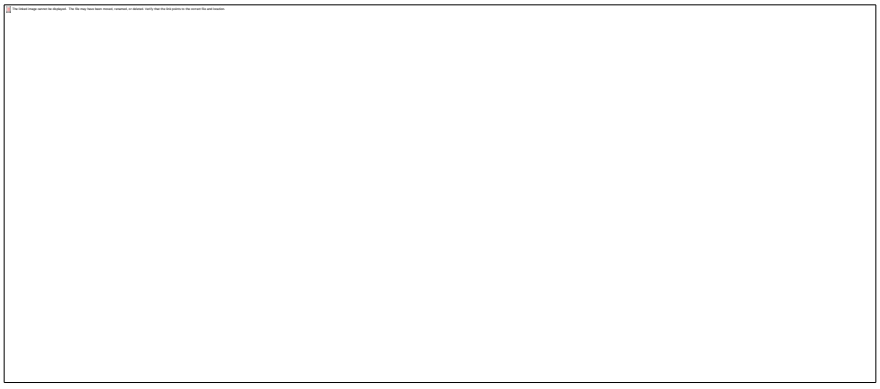
```
1604                                {
1605   assert(Nd_ == 3);
1606   FlowField& u = (FlowField&) *this;
1607   fieldstate xzstate = xzstate_;
1608   fieldstate ystate = ystate_;
1609   u.makePhysical();
```

```
1610   Vector y = chebypoints(Ny_, a_, b_);
1611   Real cfl = 0.0;
1612   Vector dx(3);
1613   dx[0] = Lx_/Nx_;
1614   dx[2] = Lz_/Nz_;
1615   for (int i=0; i<Nd_; ++i)
1616     for (int ny=0; ny<Ny_; ++ny) {
1617       dx[1] = (ny==0 || ny==Ny_-1) ? y[0]-y[1] : (y[ny-1]-y[ny+1])/2.0;
1618       for (int nx=0; nx<Nx_; ++nx)
1619         for (int nz=0; nz<Nz_; ++nz)
1620           cfl = Greater(cfl, abs(u(nx,ny,nz,i)/dx[i]));
1621   }
1622   u.makeState(xzstate,ystate);
1623
1624   return cfl;
1625 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.15.6.15 void FlowField::chebyfft_y ()

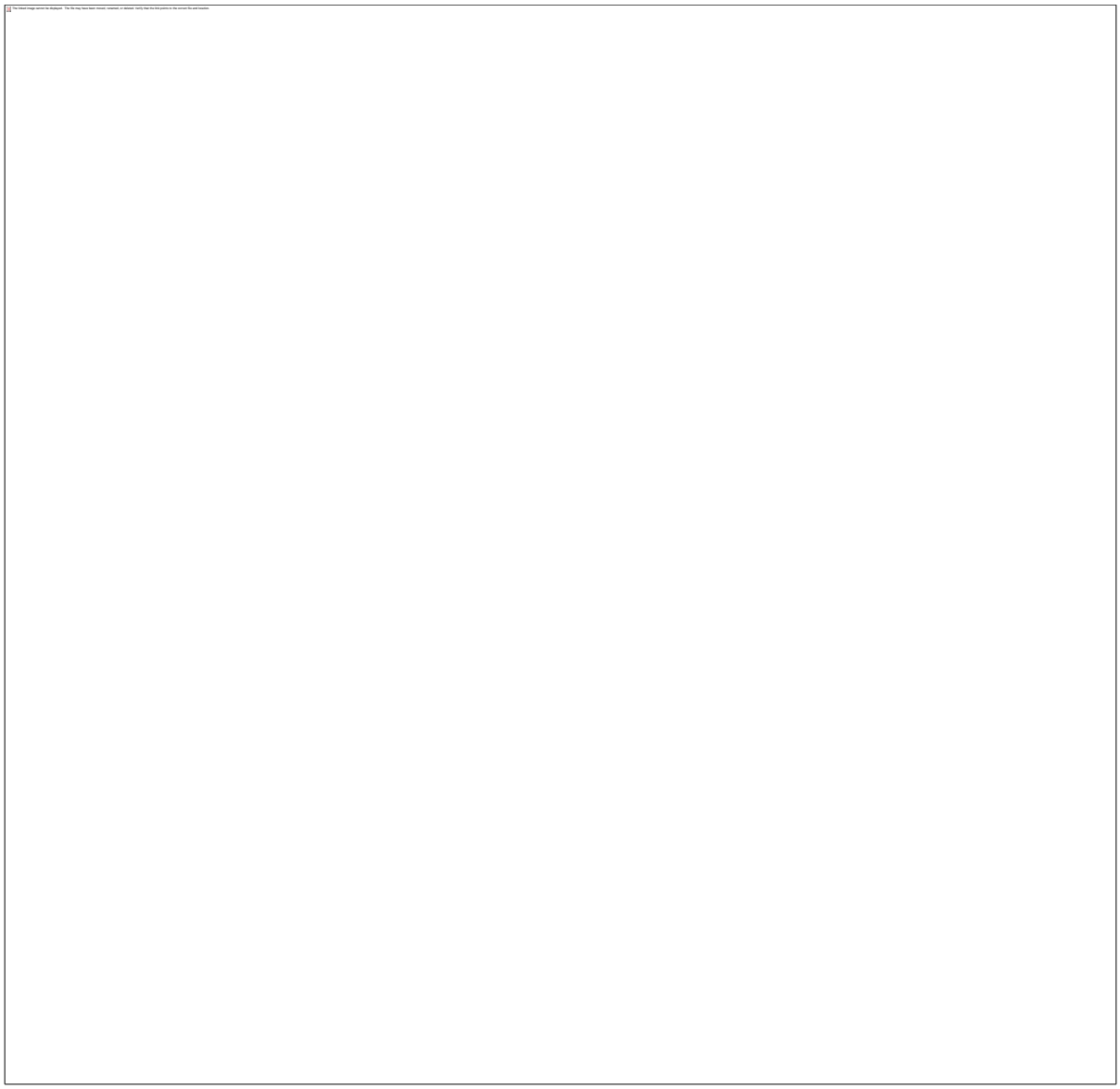
References flatten(), Nd_, Nx_, Ny_, Nzpad_, Physical, rdata_, scratch_, Spectral, y_plan_, and ystate_.

Referenced by assignOrrSommField(), and makeSpectral_y().

```
956                         {
957   assert(ystate_ == Physical);
958   if (Ny_ < 2) {
959     ystate_ = Spectral;
960     return;
961   }
962
963   Real nrm = 1.0/(Ny_-1); // needed because FFTW does unnormalized transforms
964   for (int i=0; i<Nd_; ++i)
965     for (int nx=0; nx<Nx_; ++nx)
966       for (int nz=0; nz<Nzpad_; ++nz) {
967
968         // Copy data spread through memory into a stride-1 scratch array.
969         for (int ny=0; ny<Ny_; ++ny)
970           scratch_[ny] = rdata_[flatten(nx, ny, nz, i)];
971
972         // Transform data
973         fftw_execute_r2r(y_plan_, scratch_, scratch_);
974
975         // Copy back to multi-d arrays, normalizing on the way
976         // 0th elem is different because of reln btwn cos and Cheb transforms
977         rdata_[flatten(nx, 0, nz, i)] = 0.5*nrm*scratch_[0];
978         for (int ny=1; ny<Ny_-1; ++ny)
979           rdata_[flatten(nx, ny, nz, i)] = nrm*scratch_[ny];
980         rdata_[flatten(nx, Ny_-1, nz, i)] = 0.5*nrm*scratch_[Ny_-1];
981       }
982   ystate_ = Spectral;
983   return;
984 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



7.15.6.16 const Complex & FlowField::cmplx (int *mx*, int *ny*, int *mz*, int *i*, int *j*) const [inline]

References cdata_, complex_flatten(), Spectral, and xzstate_.

```
410                                                 {
411   assert(xzstate_==Spectral);
412   return cdata_[complex_flatten(mx,my,mz,i,j)];
413 }
```

Here is the call graph for this function:

7.15.6.17 Complex & FlowField::cmplx (int *mx*, int *ny*, int *mz*, int *i*, int *j*) [inline]

References cdata_, complex_flatten(), Spectral, and xzstate_.

```
406                                                 {
407   assert(xzstate_==Spectral);
408   return cdata_[complex_flatten(mx,my,mz,i,j)];
409 }
```

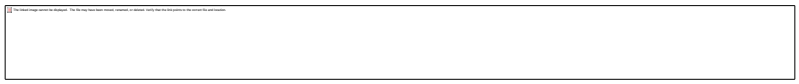
Here is the call graph for this function:

7.15.6.18 const Complex & FlowField::cmplx (int *mx*, int *my*, int *mz*, int *i*) const [inline]

References cdata_, complex_flatten(), Spectral, and xzstate_.

```
373                                                 {
374   assert(xzstate_==Spectral);
375   return cdata_[complex_flatten(mx,my,mz,i)];
376 }
```

Here is the call graph for this function:



7.15.6.19 Complex & FlowField::cmplx (int *mx*, int *my*, int *mz*, int *i*) [inline]

References cdata_, complex_flatten(), Spectral, and xzstate_.

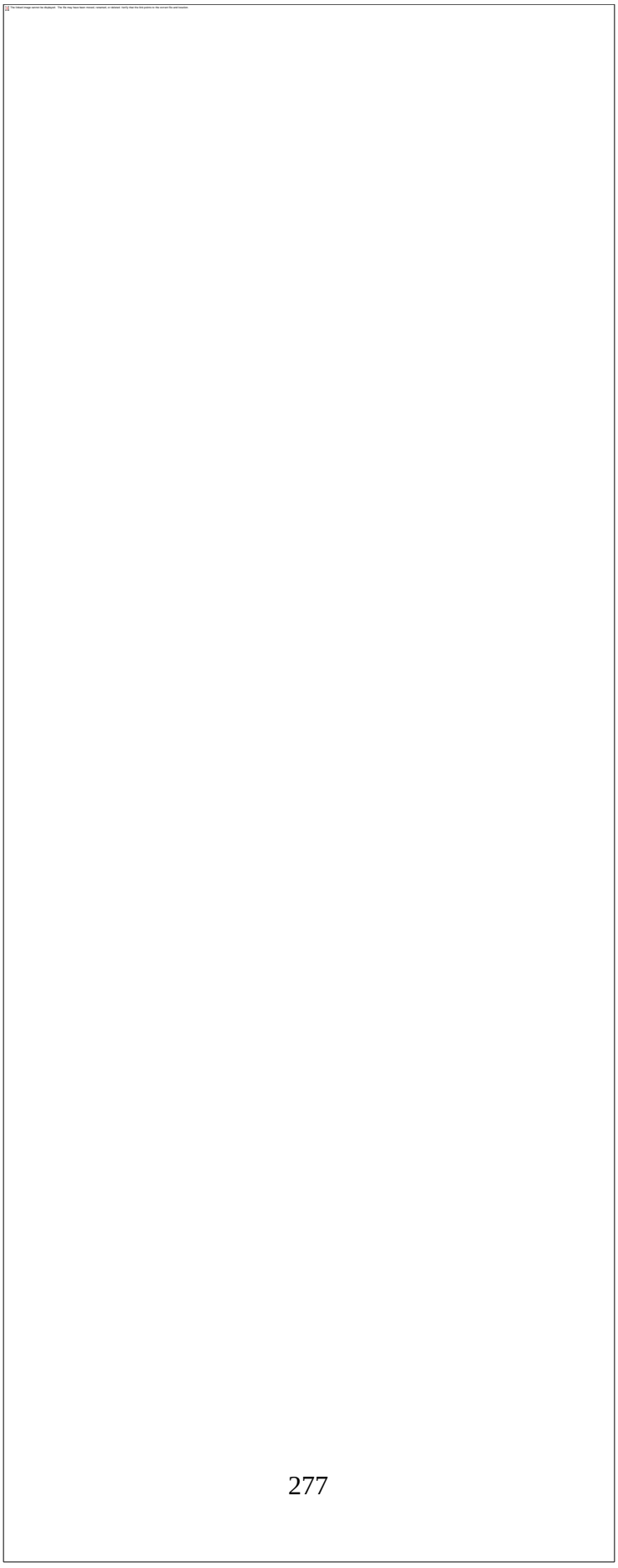
Referenced by TurbStats::addData(), addPerturbation(), addProfile(), PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), asciiSave(), assignOrrSommField(), bcDist2(), bcNorm2(), curl(), diff(), div(), DNSAlgorithm::DNSAlgorithm(), grad(), FieldSeries::interpolate(), interpolate(), L2Dist2(), L2InnerProduct(), L2Norm2(), lapl(), linearizedNL(), operator+=(), operator-=(), profile(), rescale(), rotationalNL(), skewsymmetricNL_task4::Run(), skewsymmetricNL_task3::Run(), saveDivSpectrum(), saveSpectrum(), skewsymmetricNL(), skewsymmetricNL_GPU(),

skewsymmetricNL_THREAD(), PressureSolver::solve(), PoissonSolver::solve(), translate(), PressureSolver::verify(), xdiff(), ydiff(), and zdiff().

```
369                                             {
370   assert(xzstate_==Spectral);
371   return cdata_[complex_flatten(mx,my,mz,i)];
372 }
```

Here is the call graph for this function:

Here is the caller graph for this function:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.15.6.20 const Complex & FlowField::coeff (int *mx*, int *ny*, int *mz*, int *i*, int *j*) const [inline]

References cdata_, complex_flatten(), Spectral, and xzstate_.

```
418                                                    {
419   assert(xzstate_==Spectral);
420   return cdata_[complex_flatten(mx,my,mz,i,j)];
421 }
```

Here is the call graph for this function:

7.15.6.21 Complex & FlowField::coeff (int *mx*, int *ny*, int *mz*, int *i*, int *j*) [inline]

References cdata_, complex_flatten(), Spectral, and xzstate_.

```
414                                                    {
415   assert(xzstate_==Spectral);
416   return cdata_[complex_flatten(mx,my,mz,i,j)];
417 }
```

Here is the call graph for this function:

7.15.6.22 const Complex & FlowField::coeff (int *mx*, int *my*, int *mz*, int *i*) const [inline]

References cdata_, complex_flatten(), Spectral, and xzstate_.

```
381                                                    {
382   assert(xzstate_==Spectral);
383   return cdata_[complex_flatten(mx,my,mz,i)];
384 }
```

Here is the call graph for this function:

7.15.6.23 Complex & FlowField::coeff (int *mx*, int *my*, int *mz*, int *i*) [inline]

References cdata_, complex_flatten(), Spectral, and xzstate_.

```
377                                                 {
378   assert(xzstate_==Spectral);
379   return cdata_[complex_flatten(mx,my,mz,i)];
380 }
```

Here is the call graph for this function:

7.15.6.24 int FlowField::complex_flatten (int *mx*, int *my*, int *mz*, int *i*, int *j*) const [inline, private]

References Nd_, Nx_, Ny_, and Nzpad2_.

```
392                                                                 {
393   assert(mx>=0 && mx<Nx_);     assert(my>=0 && my<Ny_);
394   assert(mz>=0 && mz<Nzpad2_); assert(i>=0  && i<Nd_);
395   return (mz + Nzpad2_*(mx + Nx_*(my + Ny_*(i + Nd_*j))));
396 }
```

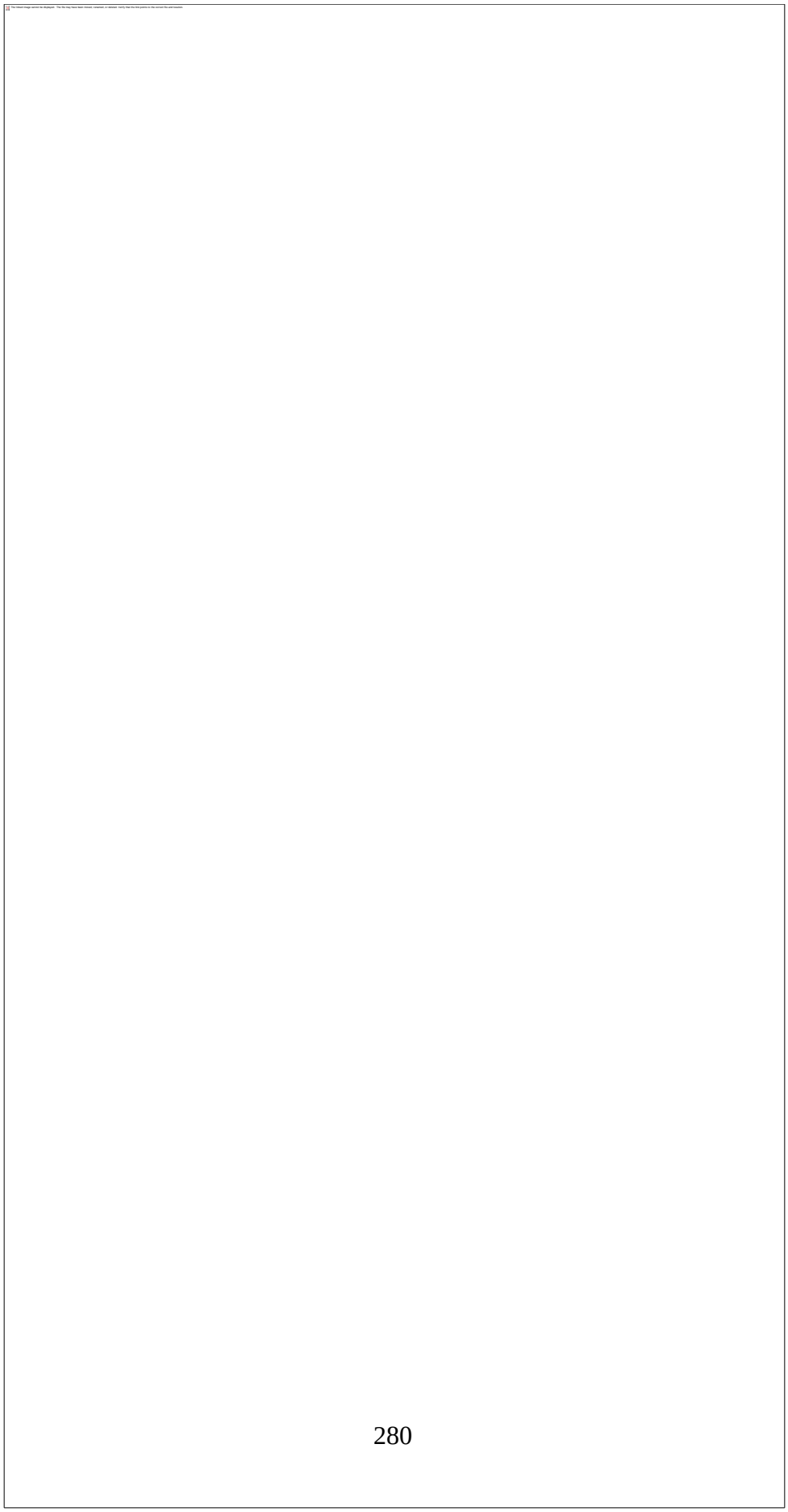
7.15.6.25 int FlowField::complex_flatten (int *mx*, int *my*, int *mz*, int *i*) const [inline, private]

References Nd_, Nx_, Ny_, and Nzpad2_.

Referenced by cmplx(), coeff(), and print().

```
355                                                         {
356   assert(mx>=0 && mx<Nx_);     assert(my>=0 && my<Ny_);
357   assert(mz>=0 && mz<Nzpad2_); assert(i>=0  && i<Nd_);
358   return (mz + Nzpad2_*(mx + Nx_*(my + Ny_*i)));
359 }
```

Here is the caller graph for this function:

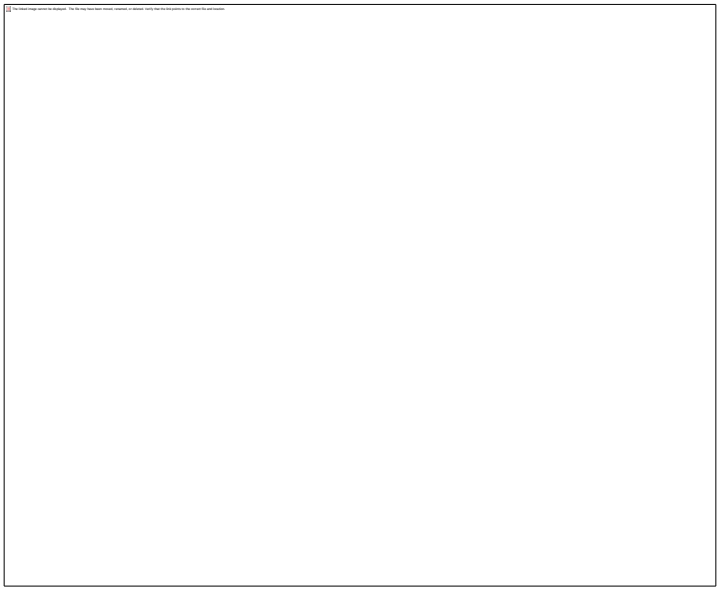


7.15.6.26 bool FlowField::congruent (const BasisFunc & *phi*) const

References BasisFunc::a(), a_, BasisFunc::b(), b_, BasisFunc::Lx(), Lx_, BasisFunc::Lz(), Lz_, BasisFunc::Ny(), Ny_, BasisFunc::state(), and ystate_.

```
837                                         {
838   return (Ny_==phi.Ny() && Lx_==phi.Lx() && Lz_==phi.Lz() &&
839           a_==phi.a() && b_==phi.b() && ystate_==phi.state());
840 }
```

Here is the call graph for this function:



7.15.6.27 bool FlowField::congruent (const FlowField & *f*) const

References a_, b_, Lx_, Lz_, Nd_, Nx_, Ny_, Nz_, xzstate_, and ystate_.

Referenced by bcDist2(), cross(), diff(), divDist2(), dot(), L1Dist(), L2Dist2(), L2InnerProduct(), lapl(), LinfDist(), operator*=(), operator+=(), operator-=(), swap(), xdiff(), ydiff(), and zdiff().

```
824                                         {
825   return (Nx_ == v.Nx_ &&
826           Ny_ == v.Ny_ &&
827           Nz_ == v.Nz_ &&
828           Nd_ == v.Nd_ &&
829           Lx_ == v.Lx_ &&
830           Lz_ == v.Lz_ &&
831           a_ == v.a_ &&
```

```
832         b_ == v.b_ &&
833         xzstate_ == v.xzstate_ &&
834         ystate_ == v.ystate_);
835 }
```

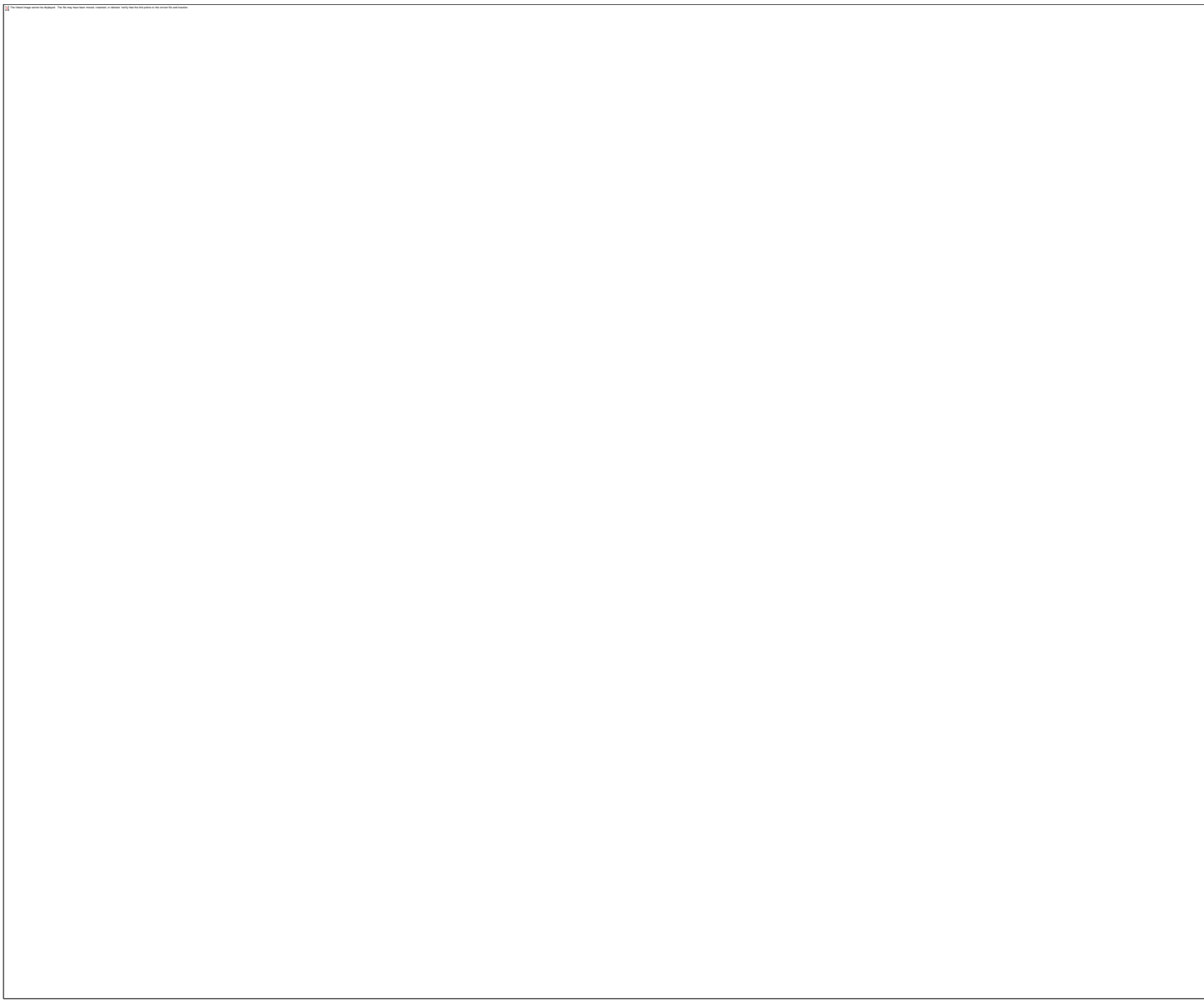
Here is the caller graph for this function:

7.15.6.28 Real FlowField::dudy_a () const

References diff(), ChebyCoeff::eval_a(), profile(), Re(), Spectral, BasisFunc::u(), and ystate_.

```
1591                          {
1592   assert(ystate_ == Spectral);
1593   BasisFunc prof = profile(0,0);
1594   ChebyCoeff dudy = diff(Re(prof.u()));
1595   return dudy.eval_a();
1596 }
```

Here is the call graph for this function:



7.15.6.29 Real FlowField::dudy_b () const

References diff(), ChebyCoeff::eval_b(), profile(), Re(), Spectral, BasisFunc::u(), and ystate_.

```
1597                                {
1598   assert(ystate_ == Spectral);
1599   BasisFunc prof = profile(0,0);
1600   ChebyCoeff dudy = diff(Re(prof.u()));
1601   return dudy.eval_b();
1602 }
```

Here is the call graph for this function:

7.15.6.30 void FlowField::dump () const

References Nd_, Nx_, Ny_, Nzpad_, and rdata_.

```
1567                         {
1568   for (int i=0; i<Nx_ * Ny_ * Nzpad_ * Nd_; ++i)
1569     cout << rdata_[i] << ' ';
1570   cout << endl;
1571 }
```

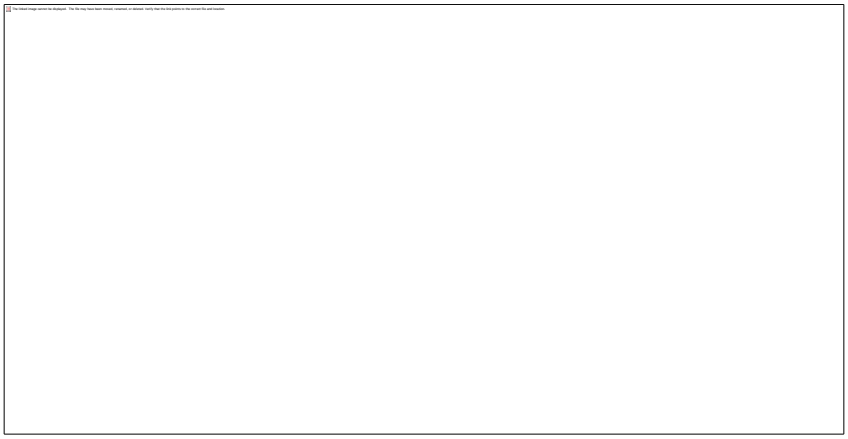
7.15.6.31 Complex FlowField::Dx (int *mx*, int *n*) const

References cferror(), kx(), kxmax(), Lx_, pi, pow(), and zero_last_mode().

```
1027                                    {
1028   Complex rot(0.0, 0.0);
1029   switch(n % 4) {
1030   case 0: rot = Complex( 1.0, 0.0); break;
1031   case 1: rot = Complex( 0.0, 1.0); break;
1032   case 2: rot = Complex(-1.0, 0.0); break;
```

```
1033   case 3: rot = Complex( 0.0,-1.0); break;
1034   default: cferror("FlowField::Dx(mx,n) : impossible: n % 4 > 4 !!");
1035   }
1036   int kx_ = kx(mx);
1037   return rot*(pow(2*pi*kx_/Lx_, n)*zero_last_mode(kx_,kxmax(),n));
1038 }
```

Here is the call graph for this function:



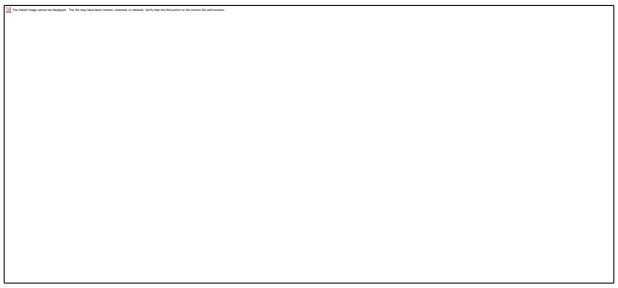
7.15.6.32 Complex FlowField::Dx (int *mx*) const

References kx(), kxmax(), Lx_, pi, and zero_last_mode().

Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), and PressureSolver::solve().

```
1052                                {
1053   int kx_ = kx(mx);
1054   return Complex(0.0, 2*pi*kx_/Lx_*zero_last_mode(kx_,kxmax(),1));
1055 }
```

Here is the call graph for this function:



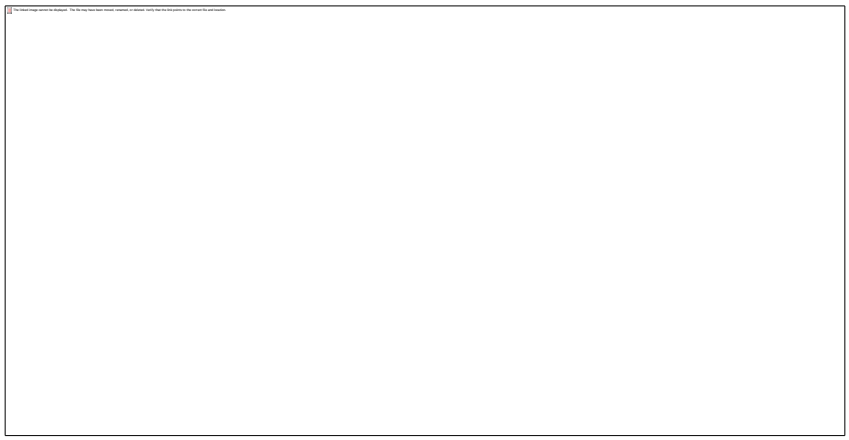
Here is the caller graph for this function:

7.15.6.33 Complex FlowField::Dz (int *mz*, int *n*) const

References cferror(), kz(), kzmax(), Lz_, pi, pow(), and zero_last_mode().

```
1039                                    {
1040   Complex rot(0.0, 0.0);
1041   switch(n % 4) {
1042   case 0: rot = Complex( 1.0, 0.0); break;
1043   case 1: rot = Complex( 0.0, 1.0); break;
1044   case 2: rot = Complex(-1.0, 0.0); break;
1045   case 3: rot = Complex( 0.0,-1.0); break;
1046   default: cferror("FlowField::Dx(mx,n) : impossible: n % 4 > 4 !!");
1047   }
1048   int kz_ = kz(mz);
1049   return rot*(pow(2*pi*kz_/Lz_, n)*zero_last_mode(kz_,kzmax(),n));
1050 }
```

Here is the call graph for this function:



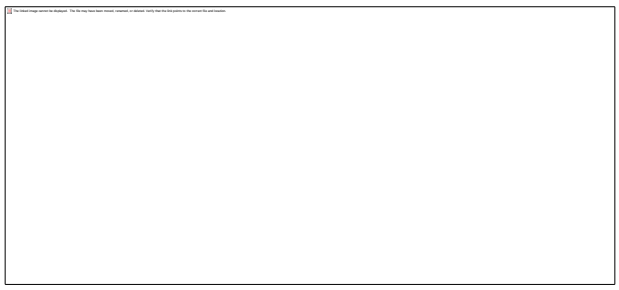
7.15.6.34 Complex FlowField::Dz (int *mz*) const

References kz(), kzmax(), Lz_, pi, and zero_last_mode().

Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), and PressureSolver::solve().

```
1056                                {
1057   int kz_ = kz(mz);
1058   return Complex(0.0, 2*pi*kz_/Lz_*zero_last_mode(kz_,kzmax(),1));
1059 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

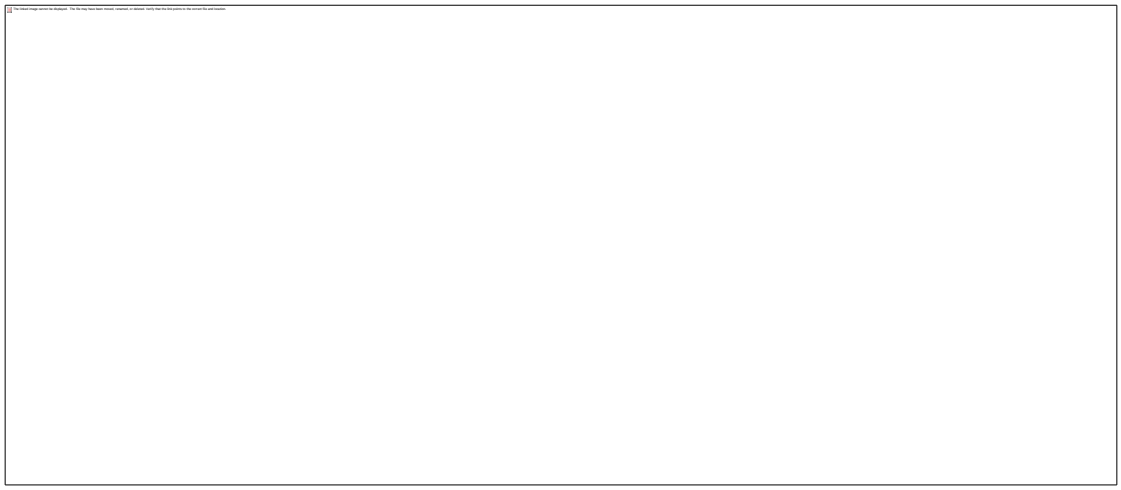
7.15.6.35 Real FlowField::energy (int *mx*, int *mz*, bool *normalize* = true) const

References a_, b_, L2Norm2(), Lx_, Lz_, Nd_, Ny_, ComplexChebyCoeff::set(), Spectral, xzstate_, and ystate_.

```
1577                                                {
1578   assert(xzstate_ == Spectral && ystate_ == Spectral);
1579   ComplexChebyCoeff u(Ny_,a_,b_,Spectral);
1580   Real e = 0.0;
1581   for (int i=0; i<Nd_; ++i) {
1582     for (int ny=0; ny<Ny_; ++ny)
```

```
1583       u.set(ny,this->cmplx(mx,ny,mz,i));
1584     e += L2Norm2(u, normalize);
1585   }
1586   if (!normalize)
1587     e *= Lx_*Lz_;
1588   return e;
1589 }
```

Here is the call graph for this function:



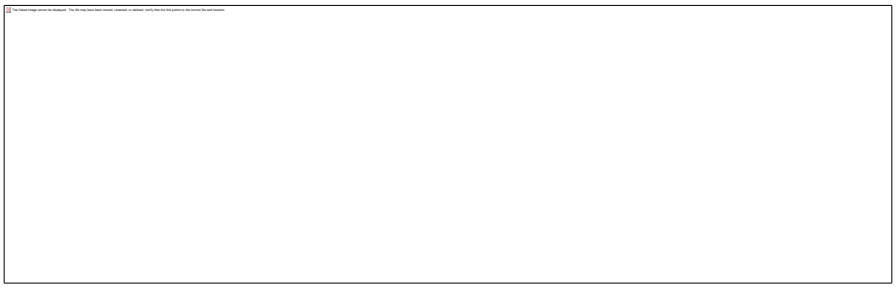
7.15.6.36 Real FlowField::energy (bool *normalize* = true) const

References L2Norm2(), Spectral, xzstate_, and ystate_.

Referenced by saveSpectrum().

```
1573                                    {
1574   assert(xzstate_ == Spectral && ystate_ == Spectral);
1575   return L2Norm2(*this, normalize);
1576 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.15.6.37 void FlowField::fftw_initialize (int *fftw_flags* = FFTW_ESTIMATE) [private]

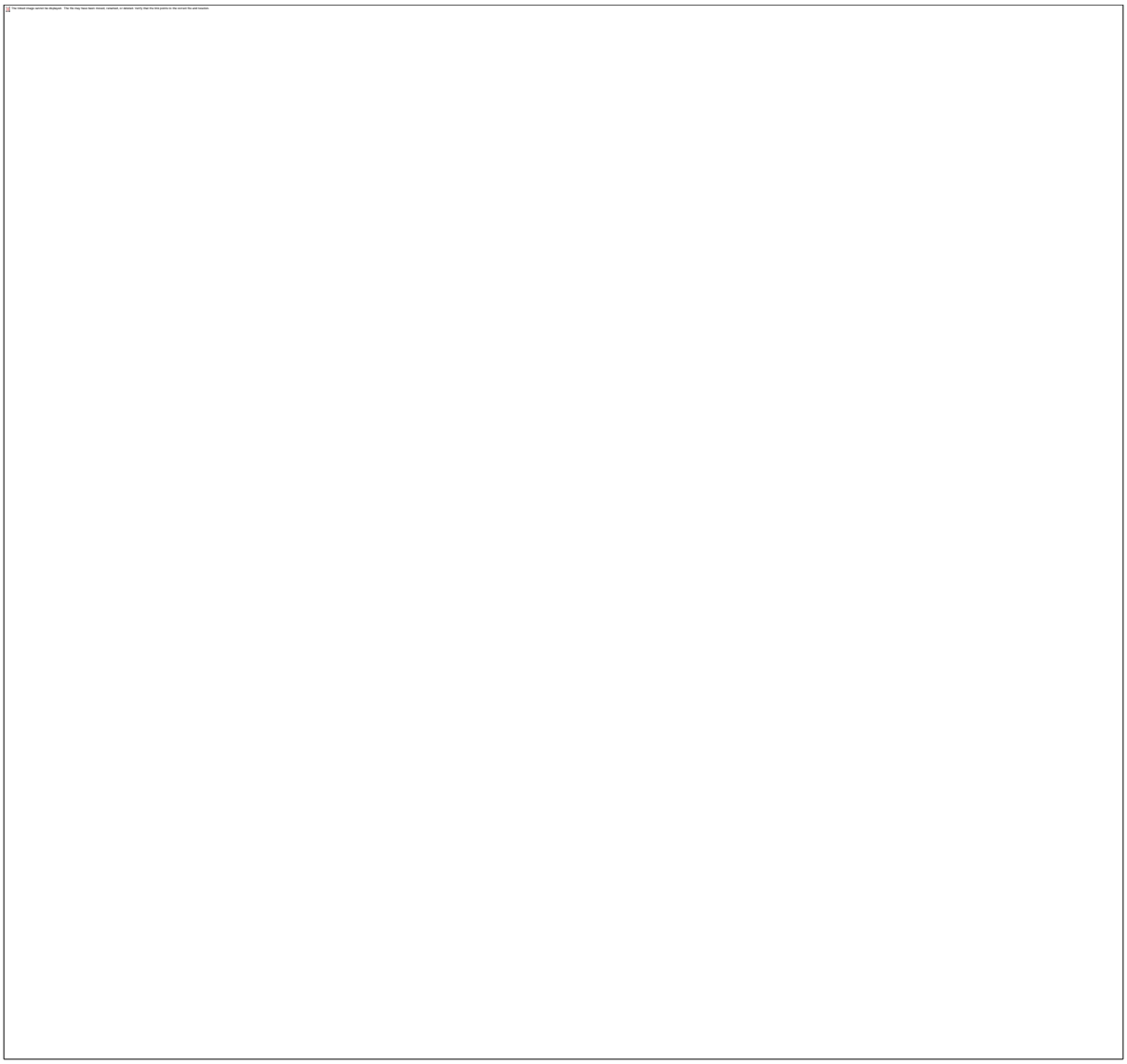
References cdata_, Nd_, Nx_, Ny_, Nz_, Nzpad2_, Nzpad_, rdata_, scratch_, xz_iplan_, xz_plan_, and y_plan_.

Referenced by FlowField(), optimizeFFTW(), and resize().

```
442                                     {
443   flags = flags | FFTW_DESTROY_INPUT;
444   flags = FFTW_MEASURE | FFTW_DESTROY_INPUT;
445
446   if (Nx_ !=0 && Nz_ !=0) {
447     fftw_complex* fcdata = (fftw_complex*) cdata_;
448
449     const int howmany = Ny_*Nd_;
450     const int rank = 2;
451
452     // These params describe the structure of the real-valued array
453     int real_n[rank];
454     real_n[0] = Nx_;
455     real_n[1] = Nz_;
456     int real_embed[rank];
457     real_embed[0] = Nx_;
458     real_embed[1] = Nzpad_;
459     const int real_stride = 1;
460     const int real_dist = Nx_*Nzpad_;
461
462     // These params describe the structure of the complex-valued array
463     int cplx_n[rank];
464     cplx_n[0] = Nx_;
465     cplx_n[1] = Nzpad2_;
466     int cplx_embed[rank];
467     cplx_embed[0] = Nx_;
468     cplx_embed[1] = Nzpad2_;
469     const int cplx_stride = 1;
470     const int cplx_dist = Nx_*Nzpad2_;
471
472     // Real -> Complex transform parameters
473     xz_plan_ =
474       fftw_plan_many_dft_r2c(rank, real_n, howmany,
475                              rdata_, real_embed, real_stride, real_dist,
```

```
476                     fcdata, cplx_embed, cplx_stride, cplx_dist,
477                     flags);
478
479   xz_iplan_ =
480     fftw_plan_many_dft_c2r(rank, real_n, howmany,
481                     fcdata, cplx_embed, cplx_stride, cplx_dist,
482                     rdata_, real_embed, real_stride, real_dist,
483                     flags);
484  }
485  else {
486   xz_plan_ = 0;
487   xz_iplan_ = 0;
488  }
489  if (Ny_ >= 2)
490   y_plan_  = fftw_plan_r2r_1d(Ny_,scratch_,scratch_,FFTW_REDFT00, flags);
491  else
492   y_plan_  = 0;
493 }
```

Here is the caller graph for this function:



7.15.6.38 int FlowField::flatten (int *nx*, int *ny*, int *nz*, int *i*, int *j*) const [inline, private]

References Nd_, Nx_, Ny_, and Nzpad_.

```
386                                                 {
387   assert(nx>=0 && nx<Nx_);    assert(ny>=0 && ny<Ny_);
388   assert(nz>=0 && nz<Nzpad_); assert(i>=0  && i<Nd_); assert(j>=0  && j<Nd_);
389   return nz + Nzpad_*(nx + Nx_*(ny + Ny_*(i + Nd_*j)));
390 }
```

7.15.6.39 int FlowField::flatten (int *nx*, int *ny*, int *nz*, int *i*) const [inline, private]

References Nd_, Nx_, Ny_, and Nzpad_.

Referenced by binarySave(), chebyfft_y(), FlowField(), ichebyfft_y(), operator()(), and print().

```
349                                                 {
350   assert(nx>=0 && nx<Nx_);    assert(ny>=0 && ny<Ny_);
351   assert(nz>=0 && nz<Nzpad_); assert(i>=0  && i<Nd_);
352   return nz + Nzpad_*(nx + Nx_*(ny + Ny_*i));
353 }
```

Here is the caller graph for this function:

7.15.6.40 bool FlowField::geomCongruent (const FlowField & *f*) const

References a_, b_, Lx_, Lz_, Nx_, Ny_, and Nz_.

Referenced by convectionNL(), cross(), curl(), div(), divergenceNL(), dot(), energy(), grad(), linearizedNL(), norm(), norm2(), outer(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), and skewsymmetricNL_THREAD().

```
814                                     {
815   return (Nx_ == v.Nx_ &&
816           Ny_ == v.Ny_ &&
817           Nz_ == v.Nz_ &&
818           Lx_ == v.Lx_ &&
819           Lz_ == v.Lz_ &&
820           a_ == v.a_ &&
821           b_ == v.b_);
822 }
```

Here is the caller graph for this function:

7.15.6.41 void FlowField::ichebyfft_y ()

References flatten(), Nd_, Nx_, Ny_, Nzpad_, Physical, rdata_, scratch_, Spectral, y_plan_, and ystate_.

Referenced by makePhysical_y().

```
986                         {
987   assert(ystate_ == Spectral);
988   if (Ny_ < 2) {
989     ystate_ = Physical;
990     return;
991   }
992   for (int i=0; i<Nd_; ++i)
993     for (int nx=0; nx<Nx_; ++nx)
994       for (int nz=0; nz<Nzpad_; ++nz) {
995
996         // Copy data spread through memory into a stride-1 scratch array.
997         // 0th and last elems are different because of reln btwn cos and
998         // Cheb transforms.
999         scratch_[0] = rdata_[flatten(nx, 0, nz, i)];
1000          for (int ny=1; ny<Ny_-1; ++ny)
1001            scratch_[ny] = 0.5*rdata_[flatten(nx, ny, nz, i)];
1002          scratch_[Ny_-1] = rdata_[flatten(nx, Ny_-1, nz, i)];
1003
1004          // Transform data
1005          fftw_execute_r2r(y_plan_, scratch_, scratch_);
1006
1007          // Copy transformed data back into main data array
1008          for (int ny=0; ny<Ny_; ++ny)
1009            rdata_[flatten(nx, ny, nz, i)] = scratch_[ny];
1010
1011        }
1012   ystate_ = Physical;
1013   return;
1014 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.15.6.42 void FlowField::interpolate (const FlowField & *u*)

References a(), b(), cmplx(), kx(), kxmax(), kxmin(), kz(), kzmax(), kzmin(), Lx(), Lz(), makeSpectral(), makeState(), mx(), mz(), Nd(), Nx(), Ny(), Nz(), setState(), setToZero(), Spectral, xzstate(), and ystate().

```
510                                     {
511   FlowField& g = *this;
512   FlowField& f = (FlowField&) ff;
513
```

```
514  fieldstate fxstate = f.xzstate();
515  fieldstate fystate = f.ystate();
516  fieldstate gxstate = g.xzstate();
517  fieldstate gystate = g.ystate();
518
519  // Always check these, debugging or not.
520  if (f.Lx() != g.Lx()  || f.Lz() != g.Lz() || f.a() != g.a()  ||
521      f.b() != g.b() || f.Nd() != g.Nd()) {
522    cerr << "FlowField::interpolate(const FlowField& f) error:\n"
523         << "FlowField doesn't match argument f geometrically.\n";
524    exit(1);
525  }
526  if (f.Nx()>g.Nx()  || f.Ny()>g.Ny()  || f.Nz()>g.Nz()) {
527    cerr << "FlowField::interpolate(const FlowField& f) error:\n"
528         << "interpolation requires Nx>=f.Nx, Ny>=f.Ny, Nz>=f.Nz\n";
529    exit(1);
530  }
531
532  f.makeSpectral();
533  g.setState(Spectral, Spectral);
534  g.setToZero();
535  for (int i=0; i<f.Nd(); ++i)
536    for (int ny=0; ny<f.Ny(); ++ny) {
537      for (int kx=f.kxmin(); kx<f.kxmax(); ++kx) {
538        int fmx = f.mx(kx);
539        int gmx = g.mx(kx);
540        for (int kz=f.kzmin(); kz<=f.kzmax(); ++kz) {
541          int fmz = f.mz(kz);
542          int gmz = g.mz(kz);
543          g.cmplx(gmx,ny,gmz,i) = f.cmplx(fmx,ny,fmz,i);
544        }
545      }
546      // Fourier discretizations go from -Mx/2+1 <= kx <= Mx/2
547      // So the last mode kx=Mx/2 has an implicit -kx counterpart that is
548      // not included in the Fourier representation. For f, this mode is
549      // implicitly/automatically included in the Fourier transforms.
550      // But if g has more x modes than f, we need to assign
551      // g(-kxmax) = conj(f(kxmax)) explicitly.
552      if (g.Nx() > f.Nx()) {
553        int kx = f.kxmax();
554        int fmx = f.mx(kx);
555        int gmx = g.mx(-kx);
556        for (int kz=f.kzmin(); kz<=f.kzmax(); ++kz) {
557          int fmz = f.mz(kz);
558          int gmz = g.mz(kz);
559          g.cmplx(gmx,ny,gmz,i) = conj(f.cmplx(fmx,ny,fmz,i));
```

```
560         }
561       }
562     }
563   f.makeState(fxstate, fystate);
564   g.makeState(gxstate, gystate);
565 }
```

Here is the call graph for this function:

7.15.6.43 void FlowField::irealfft_xz ()

References Physical, Spectral, xz_iplan_, and xzstate_.

Referenced by assignOrrSommField(), and makePhysical_xz().

```
945                         {
946   assert(xzstate_ == Spectral);
947   //cout << "rfftwnd_complex_to_real : " << endl;
948   //cout << "howmany = " << Nd_*Ny_ << endl;
949   //cout << "idist = " << Nx_*Nzpad_/2 << endl;
950
951   fftw_execute(xz_iplan_);
952   //rfftwnd_complex_to_real(xz_iplan_, Nd_*Ny_, (fftw_complex*)rdata_, 1,
Nx_*Nzpad_/2, 0,0,0);
953   xzstate_ = Physical;
954 }
```

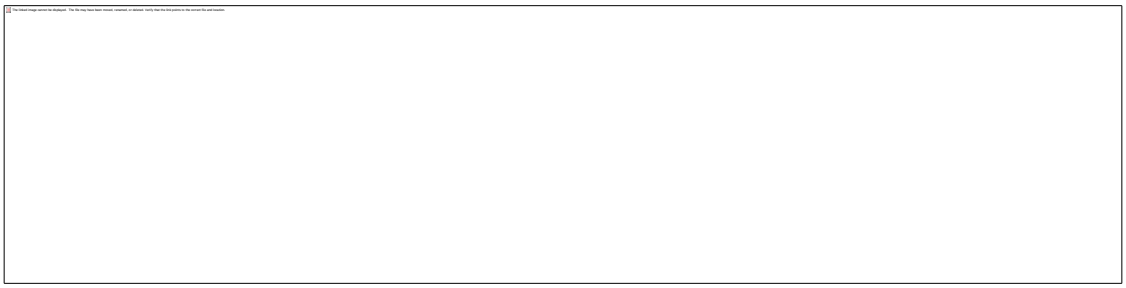
Here is the caller graph for this function:

7.15.6.44 bool FlowField::isAliased (int *kx*, int *kz*) const [inline]

References abs(), kxmaxDealiased(), and kzmaxDealiased().

```
492                                         {
493   return (abs(kx) > kxmaxDealiased() || kz > kzmaxDealiased()) ? true : false;
494 }
```

Here is the call graph for this function:



7.15.6.45 bool FlowField::isNull ()

References Nd_, Nx_, Ny_, and Nz_.

```
397                     {
398   return (Nx_ == 0 && Ny_ == 0 && Nz_ == 0 && Nd_ == 0);
399 }
```

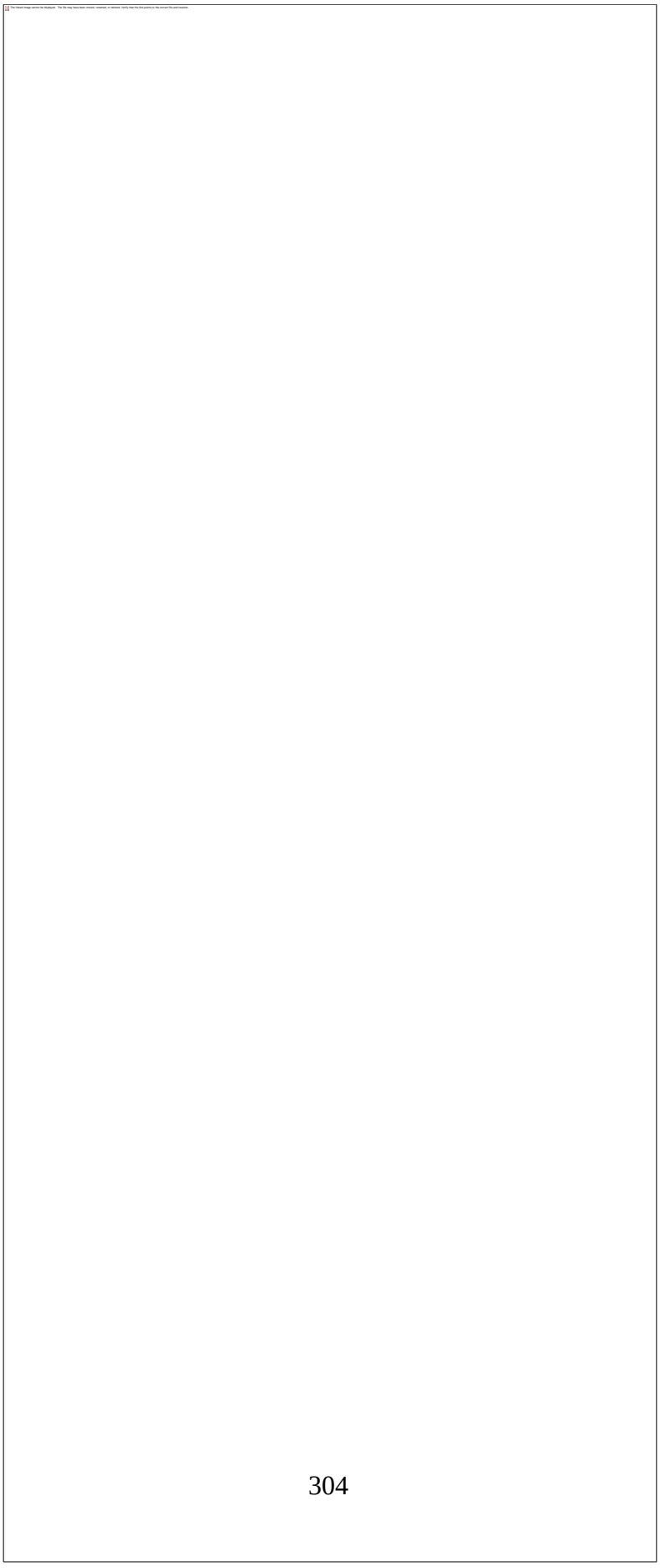
7.15.6.46 int FlowField::kx (int *mx*) const [inline]

References Nx_.

Referenced by addPerturbations(), addProfile(), PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), curl(), diff(), div(), DNSAlgorithm::DNSAlgorithm(), Dx(), grad(), interpolate(), L2Dist2(), L2Norm2(), lapl(), linearizedNL(), PoissonSolver::PoissonSolver(), profile(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), MultistepDNS::reset_dt(), rotationalNL(), skewsymmetricNL_task3::Run(), saveDivSpectrum(), saveSpectrum(), skewsymmetricNL(), skewsymmetricNL_GPU(), translate(), and xdiff().

```
433                             {
434   //assert(xzstate_ == Spectral);
435   assert(mx >= 0 && mx <Nx_);
436   return (mx <= Nx_/2) ? mx : mx - Nx_;
437 }
```

Here is the caller graph for this function:



7.15.6.47 int FlowField::kxmax () const [inline]

References Nx_.

Referenced by addPerturbations(), PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), curl(), diff(), div(), Dx(), grad(), interpolate(), linearizedNL(), perturb(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), MultistepDNS::reset_dt(), rotationalNL(), saveDivSpectrum(), saveSpectrum(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), and xdiff().

```
486 {return Nx_/2;}
```

Here is the caller graph for this function:

7.15.6.48 int FlowField::kxmaxDealiased () const [inline]

References Nx_.

Referenced by isAliased().

```
490 {return Nx_/3-1;}  // not Nx_/3
```

Here is the caller graph for this function:

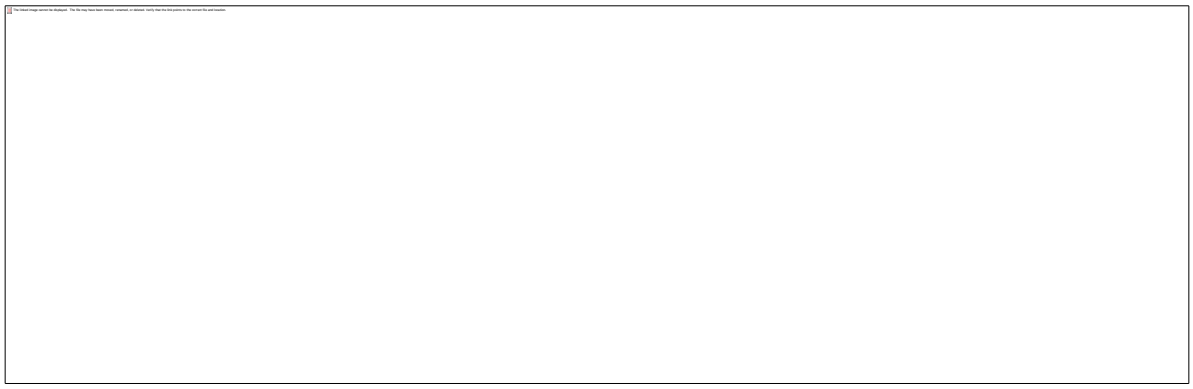
7.15.6.49 int FlowField::kxmin () const [inline]

References Nx_.

Referenced by addPerturbations(), interpolate(), saveDivSpectrum(), and saveSpectrum().

```
488 {return Nx_/2+1-Nx_;}
```

Here is the caller graph for this function:



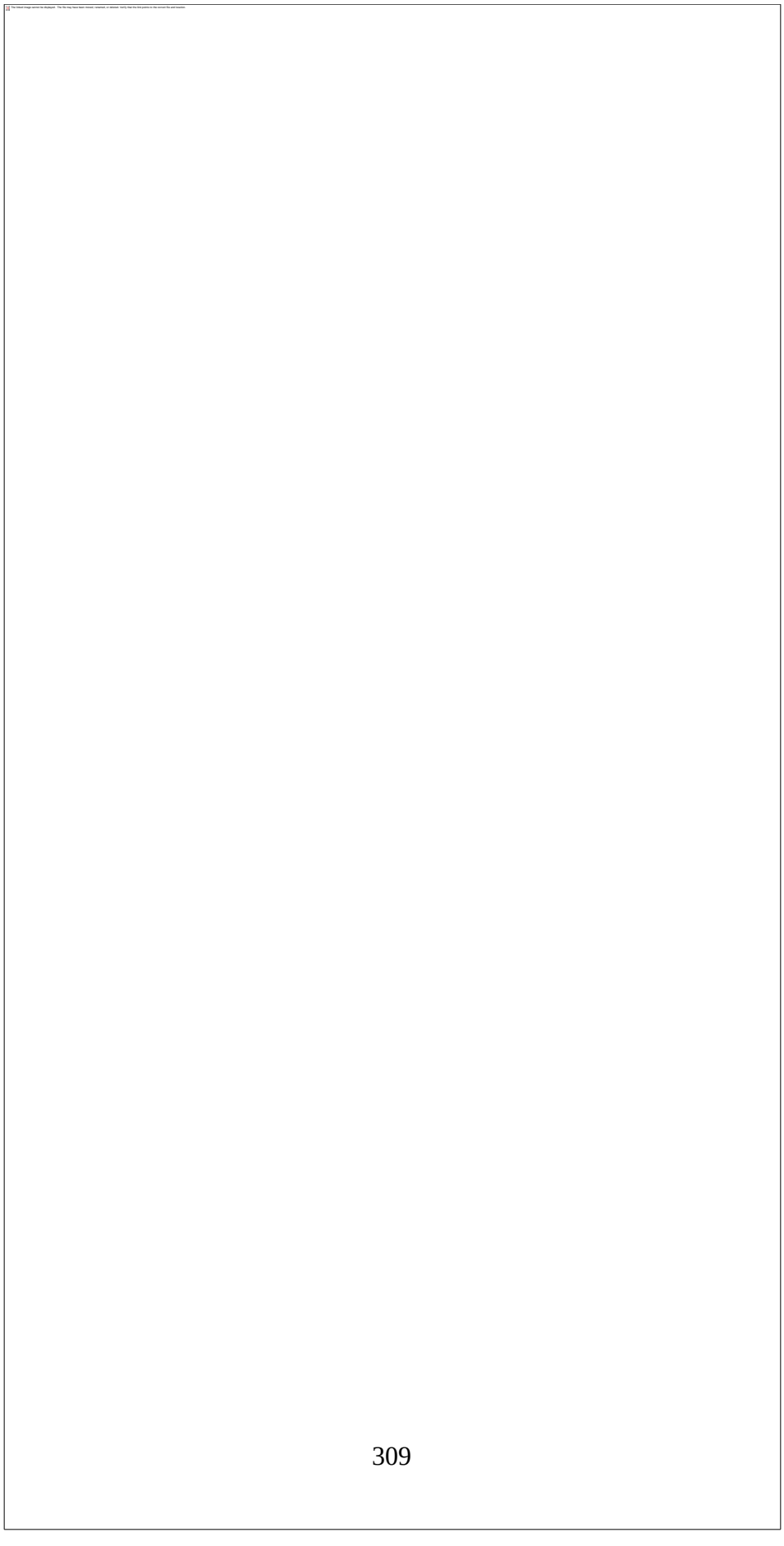
7.15.6.50 int FlowField::kz (int *mz*) const [inline]

References Nzpad2_.

Referenced by addPerturbations(), addProfile(), PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), curl(), diff(), div(), DNSAlgorithm::DNSAlgorithm(), Dz(), grad(), interpolate(), L2Dist2(), L2Norm2(), lapl(), PoissonSolver::PoissonSolver(), profile(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), MultistepDNS::reset_dt(), skewsymmetricNL_task3::Run(), saveDivSpectrum(), saveSpectrum(), skewsymmetricNL(), skewsymmetricNL_GPU(), translate(), and zdiff().

```
439                                 {
440   //assert(zstate_ == Spectral);
441   assert(mz>= 0 && mz <Nzpad2_);
442   return mz;
443 }
```

Here is the caller graph for this function:



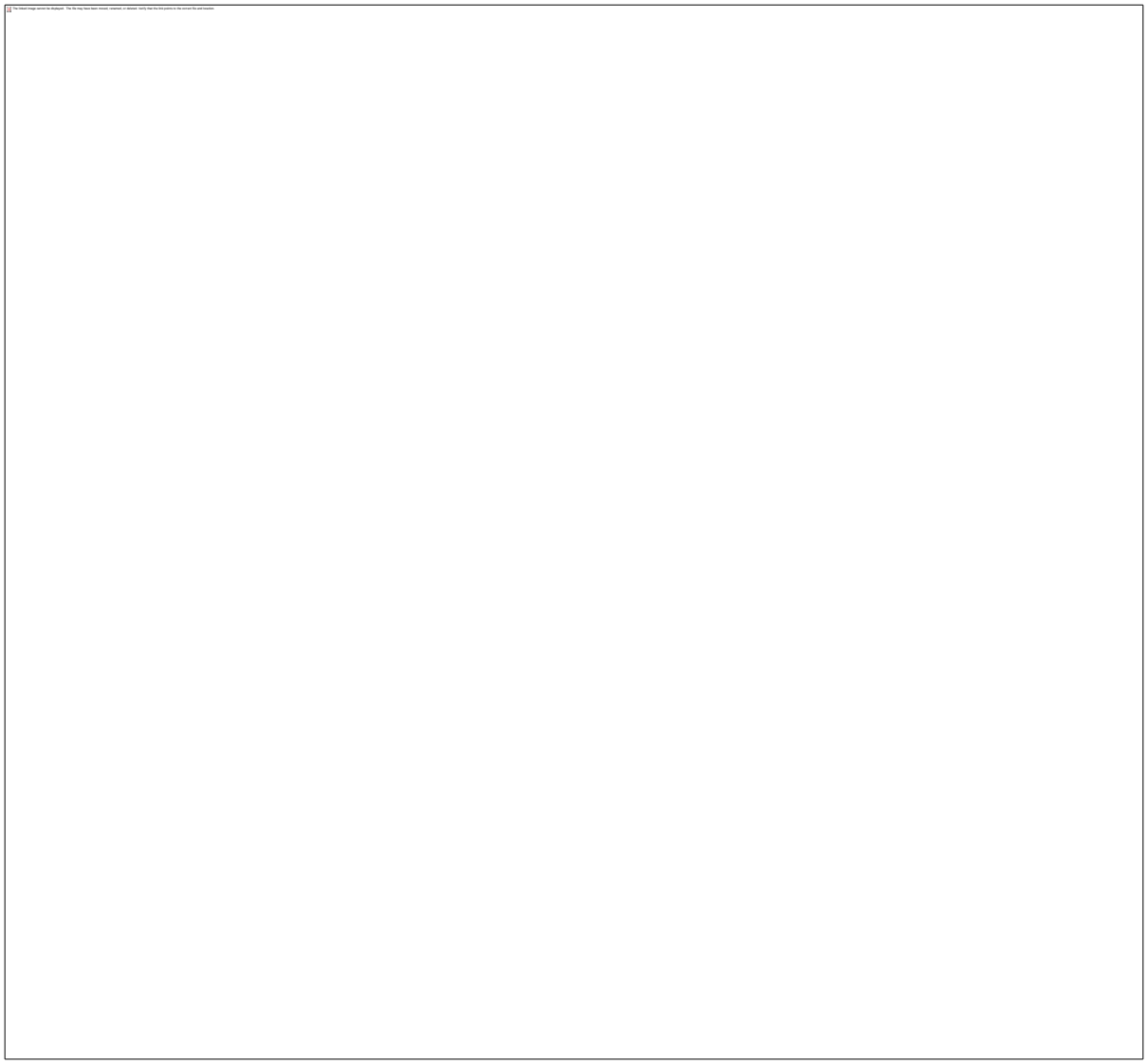
7.15.6.51 int FlowField::kzmax () const [inline]

References Nz_.

Referenced by addPerturbations(), PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), curl(), diff(), div(), Dz(), grad(), interpolate(), perturb(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), MultistepDNS::reset_dt(), saveDivSpectrum(), saveSpectrum(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), and zdiff().

```
487 {return Nz_/2;}
```

Here is the caller graph for this function:



7.15.6.52 int FlowField::kzmaxDealiased () const [inline]

References Nz_.

Referenced by isAliased().

```
491 {return Nz_/3-1;}  // not Nz_/3
```

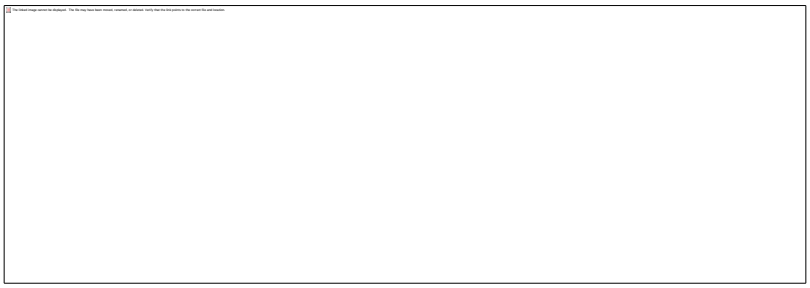
Here is the caller graph for this function:

7.15.6.53 int FlowField::kzmin () const [inline]

Referenced by interpolate(), saveDivSpectrum(), and saveSpectrum().

```
489 {return 0;}
```

Here is the caller graph for this function:



7.15.6.54 Real FlowField::Lx () const [inline]

References Lx_.

Referenced by bcDist2(), bcNorm2(), PoissonSolver::congruent(), convectionNL(), cross(), curl(), diff(), div(), divergenceNL(), DNSAlgorithm::DNSAlgorithm(), dot(), energy(), PoissonSolver::geomCongruent(), grad(), interpolate(), L1Dist(), L1Norm(), L2Dist2(), L2InnerProduct(), L2Norm2(), lapl(), linearizedNL(), norm(), norm2(), outer(), PPEDNS::PPEDNS(), PPEDNS::push(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), PressureSolver::verify(), xdiff(), ydiff(), and zdiff().

```
427 {return Lx_;}
```

Here is the caller graph for this function:

7.15.6.55 Real FlowField::Ly () const [inline]

References a_, and b_.

```
428 {return b_ - a_;}
```

7.15.6.56 Real FlowField::Lz () const [inline]

References Lz_.

Referenced by bcDist2(), bcNorm2(), PoissonSolver::congruent(), convectionNL(), cross(), curl(), diff(), div(), divergenceNL(), DNSAlgorithm::DNSAlgorithm(), dot(), energy(), PoissonSolver::geomCongruent(), grad(), interpolate(), L1Dist(), L1Norm(), L2Dist2(), L2InnerProduct(), L2Norm2(), lapl(), linearizedNL(), norm(), norm2(), outer(), PPEDNS::PPEDNS(), PPEDNS::push(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), PressureSolver::verify(), xdiff(), ydiff(), and zdiff().

```
429 {return Lz_;}
```

Here is the caller graph for this function:

7.15.6.57 void FlowField::makePhysical ()

References makePhysical_xz(), and makePhysical_y().

Referenced by addPerturbations(), CFLfactor(), changeBaseFlow(), convectionNL(), cross(), divergenceNL(), dot(), energy(), norm(), norm2(), outer(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), up2q(), and uq2p().

```
1021 {makePhysical_y(); makePhysical_xz();}
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.15.6.58 void FlowField::makePhysical_xz ()

References irealfft_xz(), Spectral, and xzstate_.

Referenced by TurbStats::addData(), makePhysical(), and makeState().

```
1017 {if (xzstate_==Spectral) irealfft_xz();}
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.15.6.59 void FlowField::makePhysical_y ()

References ichebyfft_y(), Spectral, and ystate_.
Referenced by TurbStats::addData(), makePhysical(), and makeState().

```
1019 {if (ystate_==Spectral) ichebyfft_y();}
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.15.6.60 void FlowField::makeSpectral ()

References makeSpectral_xz(), and makeSpectral_y().

Referenced by addPerturbations(), convectionNL(), cross(), curl(), diff(), div(), divergenceNL(), dot(), energy(), grad(), interpolate(), lapl(), linearizedNL(), norm(), norm2(), outer(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), xdiff(), ydiff(), and zdiff().

```
1020 {makeSpectral_xz(); makeSpectral_y();}
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.15.6.61 void FlowField::makeSpectral_xz ()

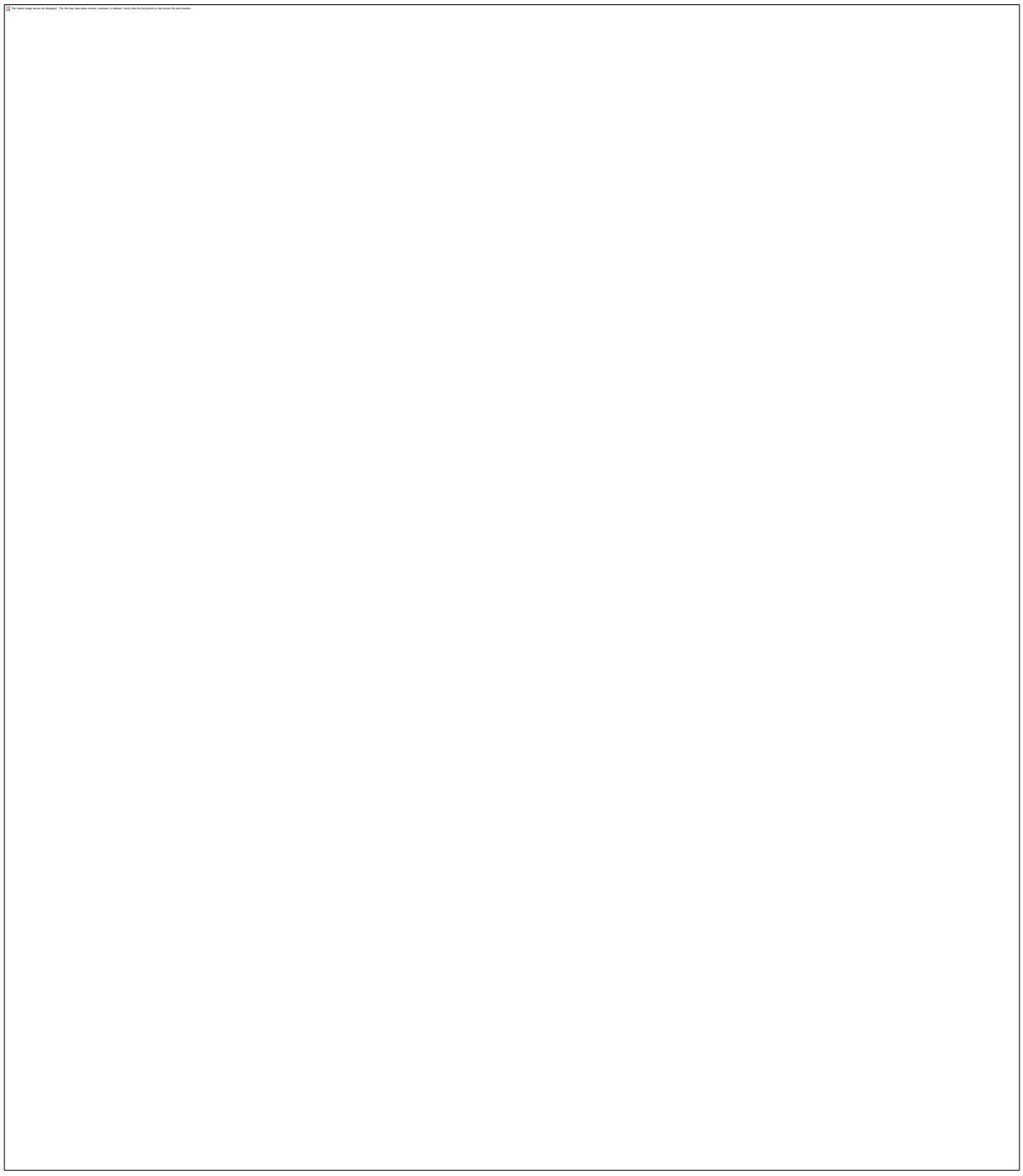
References Physical, realfft_xz(), and xzstate_.

Referenced by TurbStats::addData(), makeSpectral(), makeState(), xdiff(), and zdiff().

```
1016 {if (xzstate_==Physical) realfft_xz();}
```

Here is the call graph for this function:

Here is the caller graph for this function:



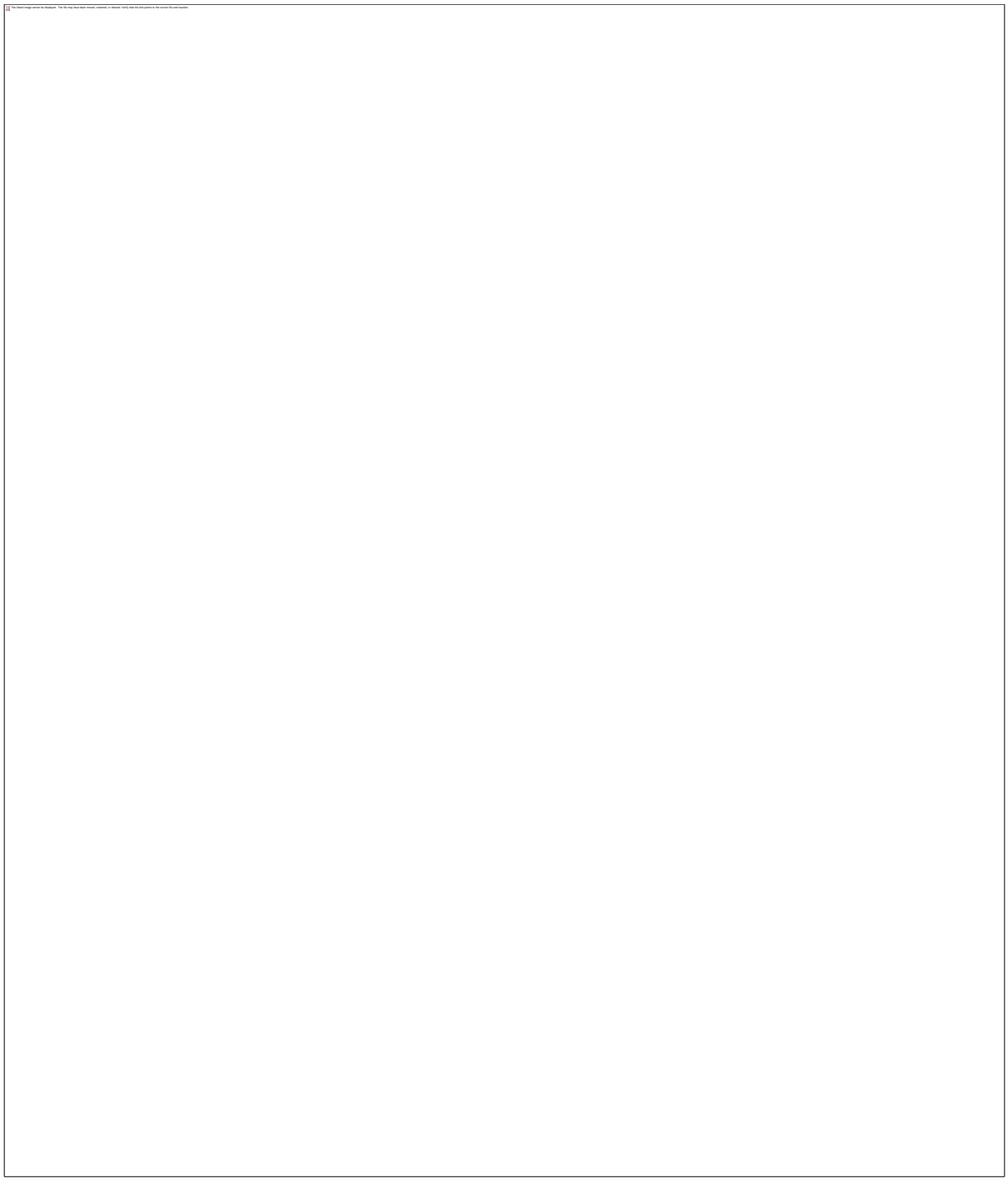
7.15.6.62 void FlowField::makeSpectral_y ()

References chebyfft_y(), Physical, and ystate_.
Referenced by makeSpectral(), makeState(), and ydiff().

```
1018 {if (ystate_==Physical) chebyfft_y();}
```

Here is the call graph for this function:

Here is the caller graph for this function:



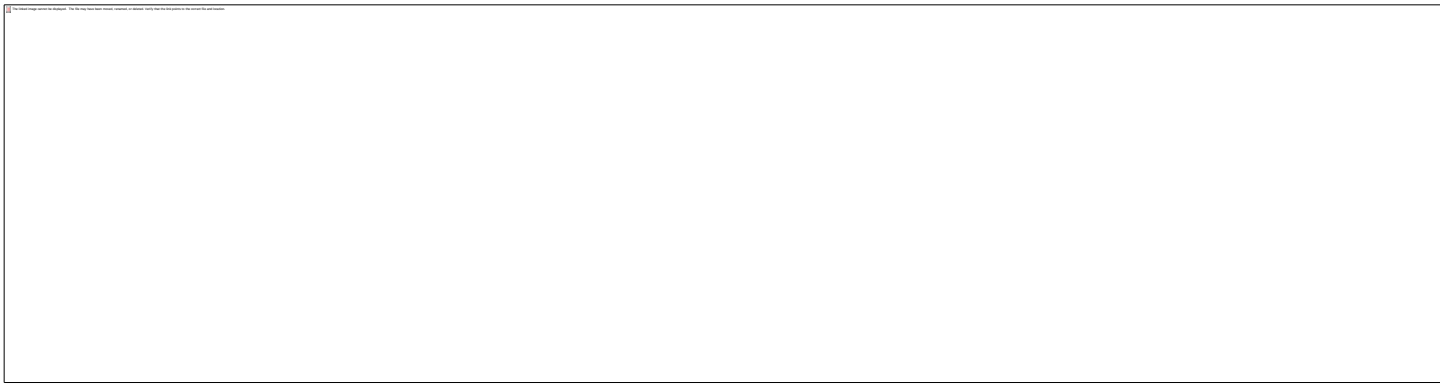
7.15.6.63 void FlowField::makeState (fieldstate *xzstate*, fieldstate *ystate*)

References makePhysical_xz(), makePhysical_y(), makeSpectral_xz(), makeSpectral_y(), and Physical.

Referenced by TurbStats::addData(), CFLfactor(), changeBaseFlow(), convectionNL(), cross(), curl(), diff(), div(), divergenceNL(), dot(), energy(), grad(), interpolate(), lapl(), linearizedNL(), norm(), norm2(), outer(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), translate(), up2q(), uq2p(), xdiff(), ydiff(), and zdiff().

```
1022                                                {
1023   (xzstate == Physical) ? makePhysical_xz() : makeSpectral_xz();
1024   (ystate  == Physical) ? makePhysical_y()  : makeSpectral_y();
1025 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.15.6.64 int FlowField::mx (int *kx*) const [inline]

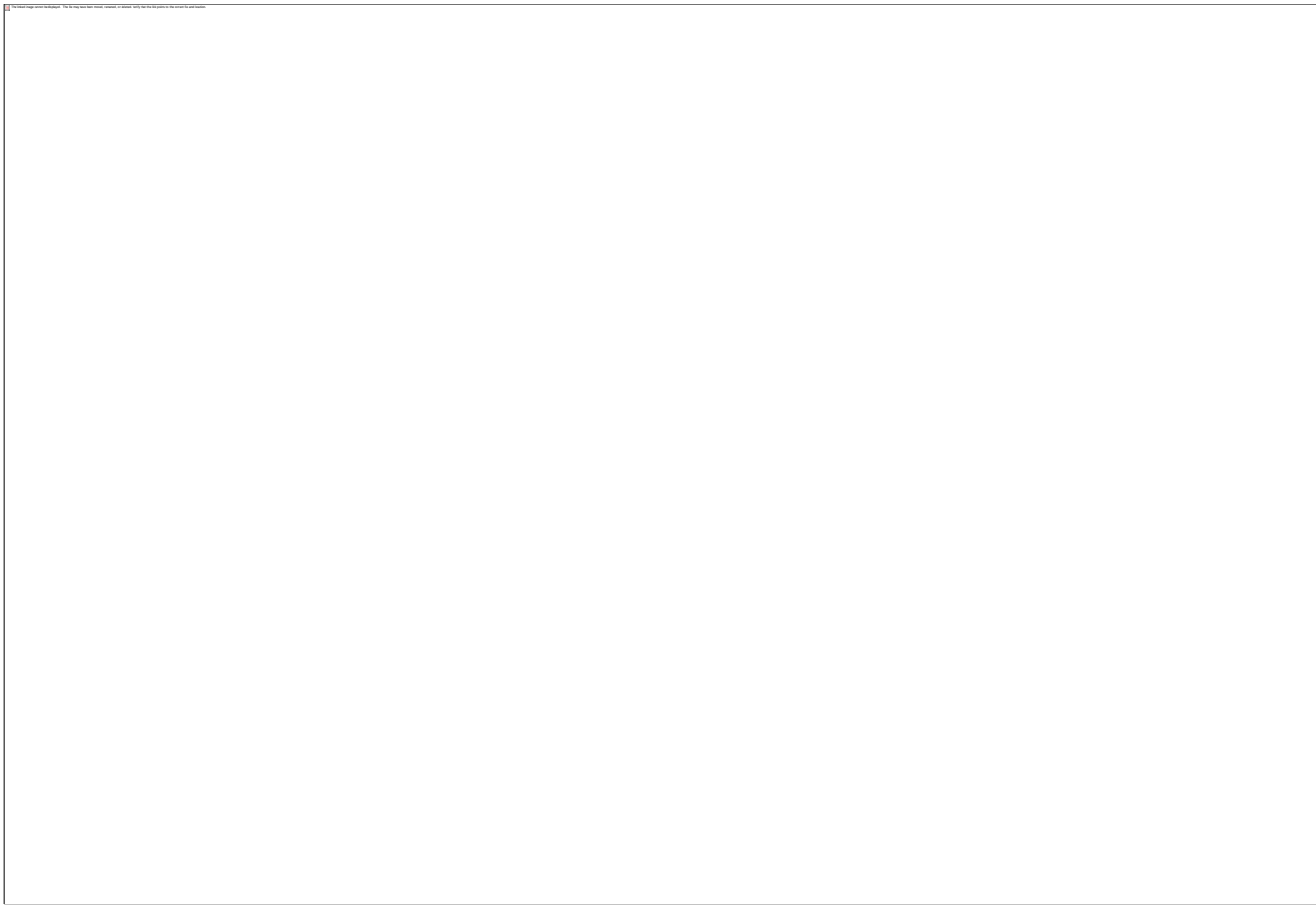
References Nx_.

Referenced by addPerturbation(), addPerturbations(), addProfile(), asciiSave(), interpolate(), L2InnerProduct(), rescale(), saveDivSpectrum(), saveSpectrum(), and translate().

```
445                                 {
446   //assert(xzstate_ == Spectral);
```

```
447   assert(kx >= 1-Nx_/2 && kx <= Nx_/2);
448   return (kx >= 0) ? kx : kx + Nx_;
449 }
```

Here is the caller graph for this function:



7.15.6.65 int FlowField::Mx () const [inline]

References Nx_.

Referenced by addPerturbations(), asciiSave(), assignOrrSommField(), bcDist2(), bcNorm2(), PoissonSolver::congruent(), curl(), diff(), div(), divDist2(), divNorm2(), PoissonSolver::geomCongruent(), grad(), L2Dist2(), L2InnerProduct(), L2Norm2(), lapl(), linearizedNL(), rescale(), rotationalNL(), saveDivSpectrum(), saveSpectrum(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), translate(), xdiff(), ydiff(), and zdiff().

```
482 {return Nx_;}
```

Here is the caller graph for this function:

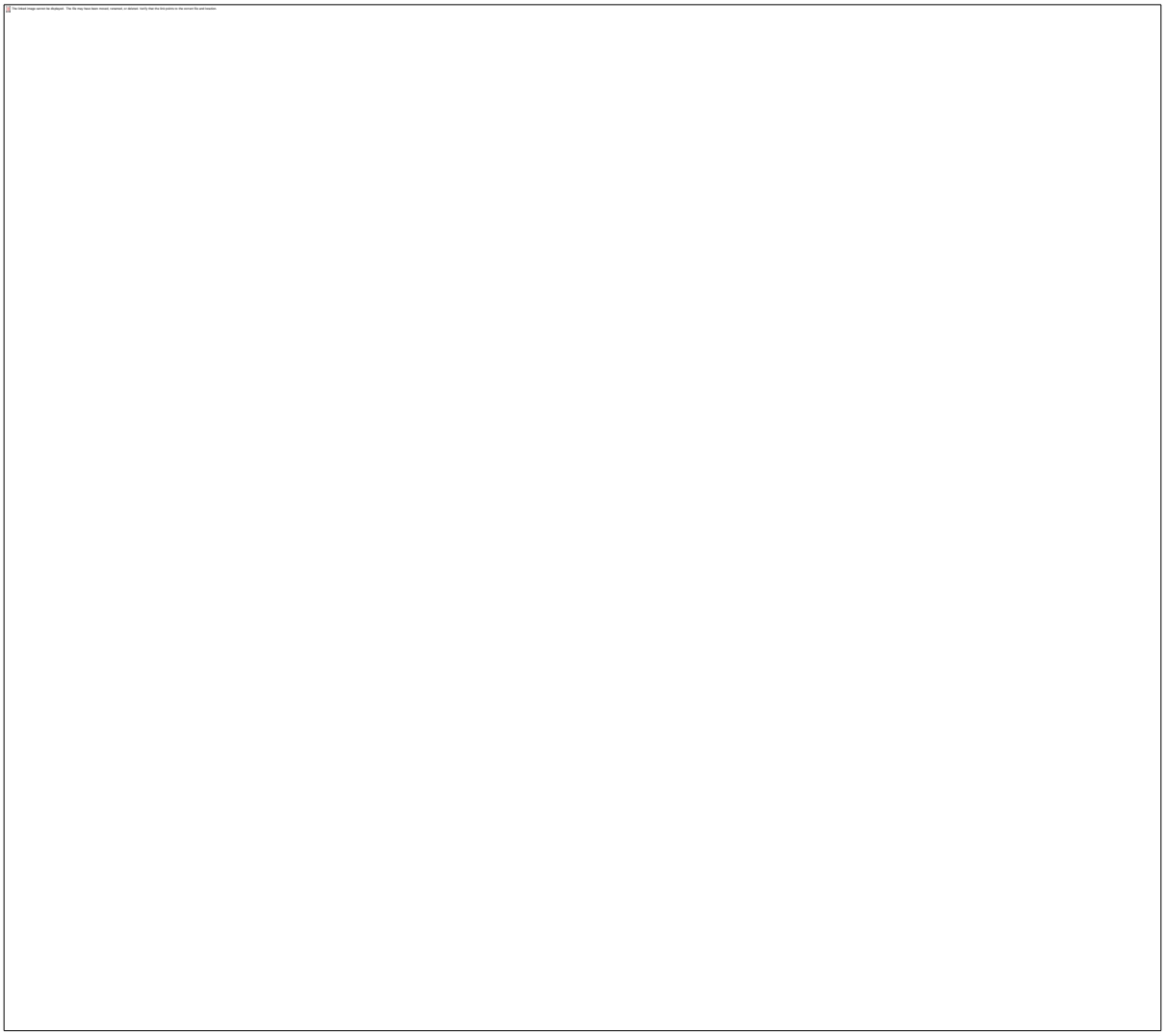
7.15.6.66 int FlowField::My () const [inline]

References Ny_.

Referenced by asciiSave(), bcDist2(), bcNorm2(), PoissonSolver::congruent(), curl(), diff(), div(), PoissonSolver::geomCongruent(), grad(), lapl(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), xdiff(), ydiff(), and zdiff().

```
483 {return Ny_;}
```

Here is the caller graph for this function:



7.15.6.67 int FlowField::mz (int *kz*) const [inline]

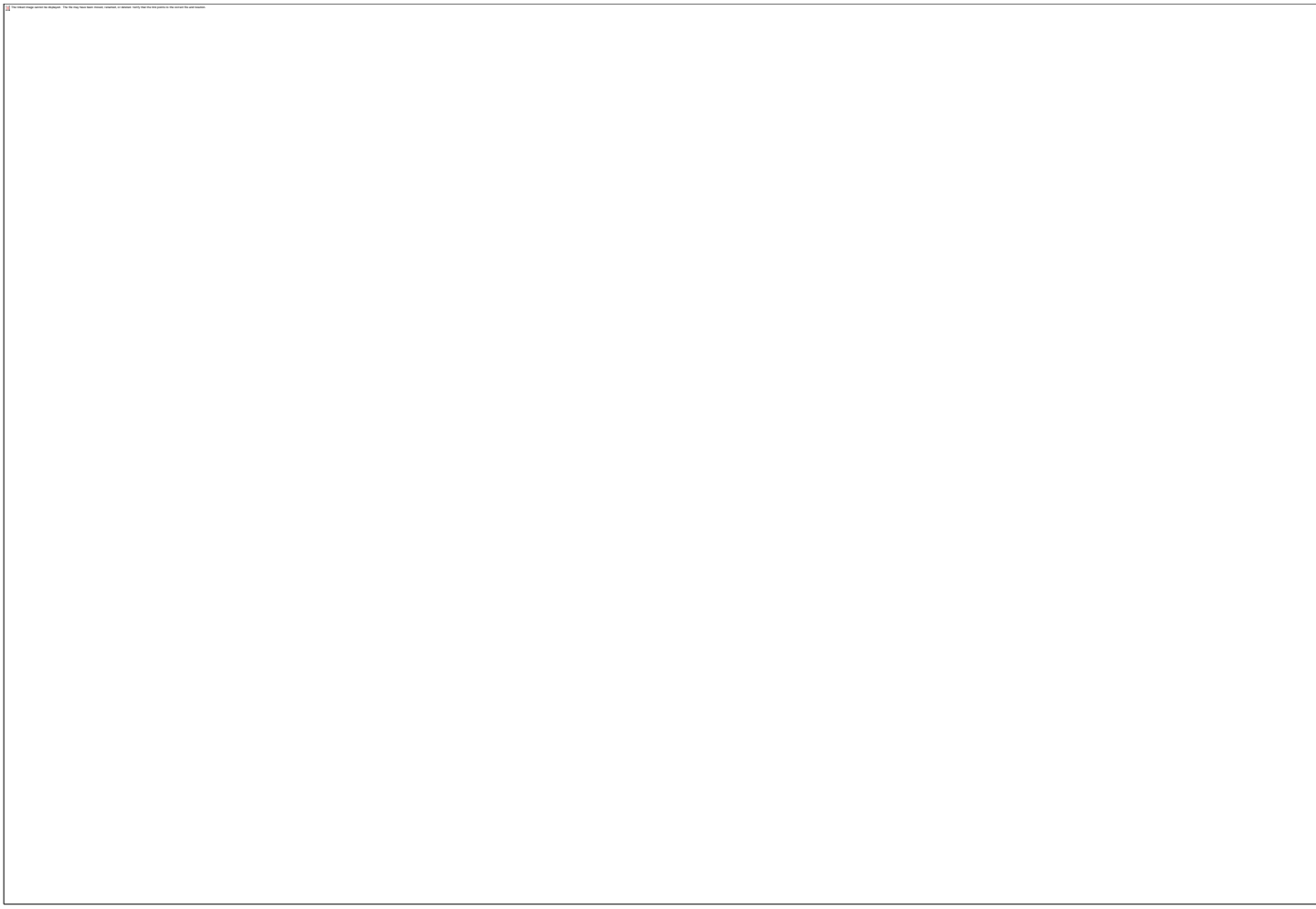
References Nzpad2_.

Referenced by addPerturbation(), addPerturbations(), addProfile(), asciiSave(), interpolate(), L2InnerProduct(), rescale(), saveDivSpectrum(), saveSpectrum(), and translate().

```
451                                {
452   //assert(zstate_ == Spectral);
```

```
453   assert(kz>= 0 && kz <Nzpad2_);
454   return kz;
455 }
```

Here is the caller graph for this function:



7.15.6.68 int FlowField::Mz () const [inline]

References Nzpad2_.

Referenced by addPerturbations(), asciiSave(), bcDist2(), bcNorm2(), PoissonSolver::congruent(), curl(), diff(), div(), divDist2(), divNorm2(), PoissonSolver::geomCongruent(), grad(), L2Dist2(), L2InnerProduct(), L2Norm2(), lapl(), linearizedNL(), rescale(), rotationalNL(), saveDivSpectrum(), saveSpectrum(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), translate(), xdiff(), ydiff(), and zdiff().

```
484 {return Nzpad2_;}
```

Here is the caller graph for this function:

7.15.6.69 int FlowField::Nd () const [inline]

References Nd_.

Referenced by bcDist2(), bcNorm2(), PoissonSolver::congruent(), convectionNL(), cross(), curl(), diff(), div(), divDist2(), divergenceNL(), divNorm2(), dot(), energy(), grad(), interpolate(), L2InnerProduct(), lapl(), linearizedNL(), norm(), norm2(), outer(), profile(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), up2q(), uq2p(), xdiff(), ydiff(), and zdiff().

```
496 {return Nd_;}
```

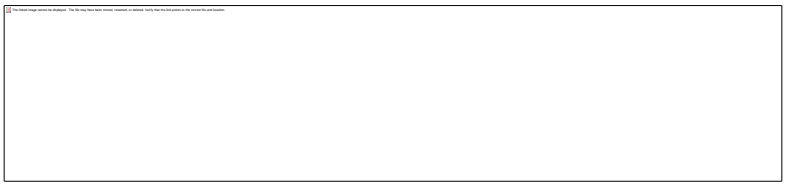
Here is the caller graph for this function:

7.15.6.70 int FlowField::numXgridpts () const [inline]

References Nx_.
Referenced by TurbStats::addData(), and changeBaseFlow().

```
474 {return Nx_;}
```

Here is the caller graph for this function:



7.15.6.71 int FlowField::numXmodes () const [inline]

References Nx_.

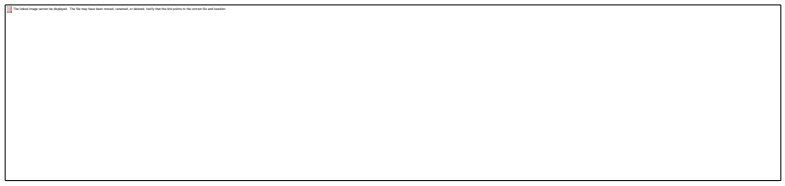
```
470 {return Nx_;}
```

7.15.6.72 int FlowField::numYgridpts () const [inline]

References Ny_.
Referenced by TurbStats::addData(), and changeBaseFlow().

```
475 {return Ny_; }
```

Here is the caller graph for this function:



7.15.6.73 int FlowField::numYmodes () const [inline]

References Ny_.
Referenced by rescale().

```
471 {return Ny_; }
```

Here is the caller graph for this function:

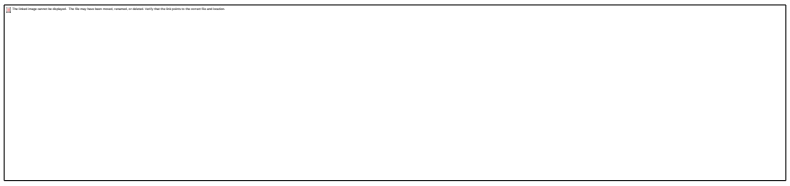
7.15.6.74 int FlowField::numZgridpts () const [inline]

References Nz_.

Referenced by TurbStats::addData(), and changeBaseFlow().

476 {return Nz_;}

Here is the caller graph for this function:



7.15.6.75 int FlowField::numZmodes () const [inline]

References Nzpad2_.

472 {return Nzpad2_;} // Nzpad2 = Nz/2+1

7.15.6.76 int FlowField::Nx () const [inline]

References Nx_.

Referenced by assignOrrSommField(), convectionNL(), cross(), curl(), diff(), div(), divergenceNL(), DNSAlgorithm::DNSAlgorithm(), dot(), energy(), grad(), interpolate(), L1Dist(), L1Norm(), lapl(), linearizedNL(), LinfDist(), LinfNorm(), norm(), norm2(), outer(), PPEDNS::PPEDNS(), PPEDNS::push(), rotationalNL(), skewsymmetricNL_task2::Run(), skewsymmetricNL_task::Run(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), up2q(), uq2p(), PressureSolver::verify(), xdiff(), ydiff(), and zdiff().

478 {return Nx_;}

Here is the caller graph for this function:

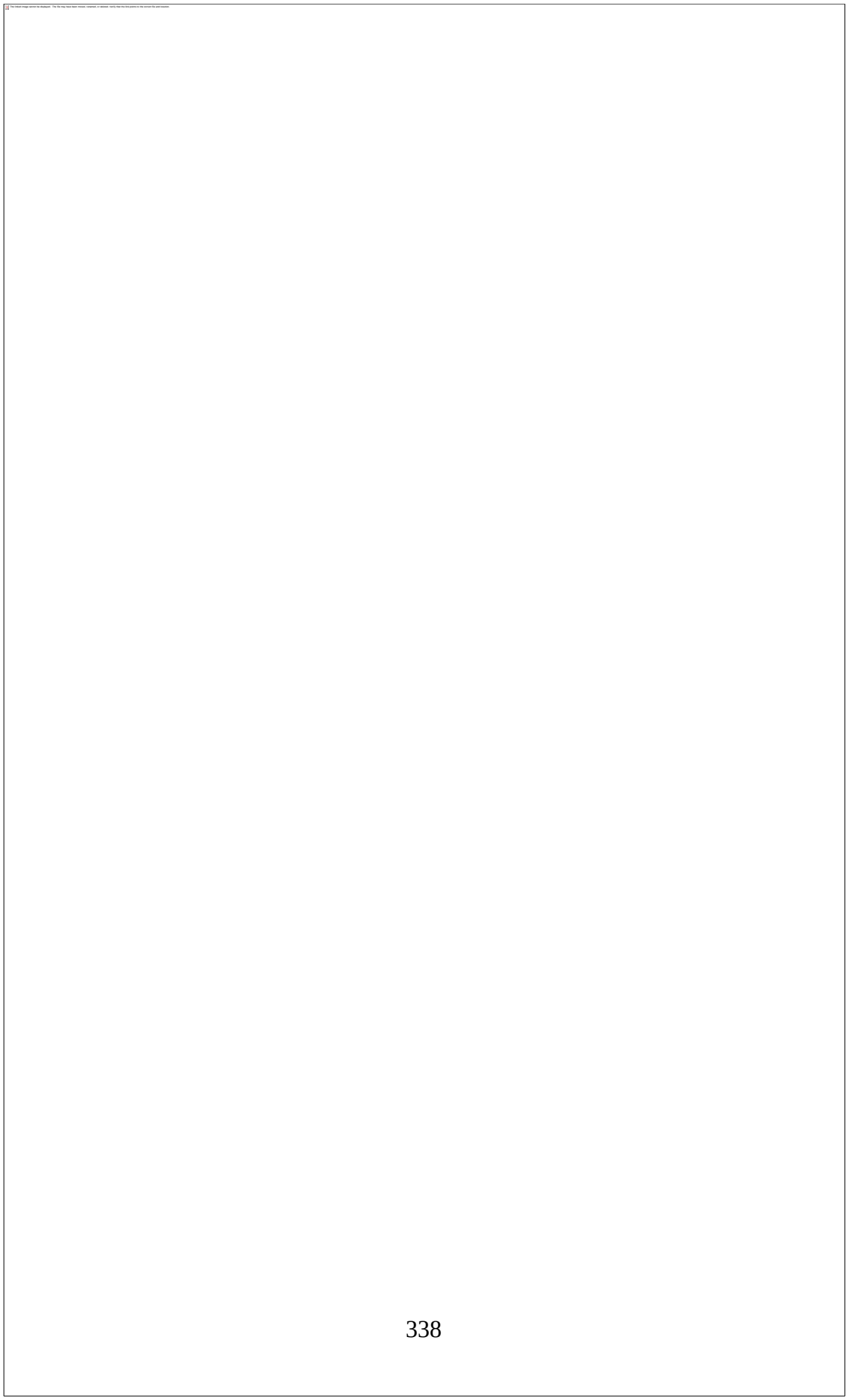
7.15.6.77 int FlowField::Ny () const [inline]

References Ny_.

Referenced by assignOrrSommField(), bcNorm2(), convectionNL(), cross(), curl(), diff(), div(), divergenceNL(), DNSAlgorithm::DNSAlgorithm(), dot(), energy(), grad(), interpolate(), L1Dist(), L1Norm(), L2Dist2(), L2InnerProduct(), L2Norm2(), lapl(), linearizedNL(), LinfDist(), LinfNorm(), norm(), norm2(), outer(), PPEDNS::PPEDNS(), PressureSolver::PressureSolver(), PPEDNS::push(), rotationalNL(), skewsymmetricNL_task2::Run(), skewsymmetricNL_task::Run(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), up2q(), uq2p(), PressureSolver::verify(), xdiff(), ydiff(), and zdiff().

```
479 {return Ny_;}
```

Here is the caller graph for this function:



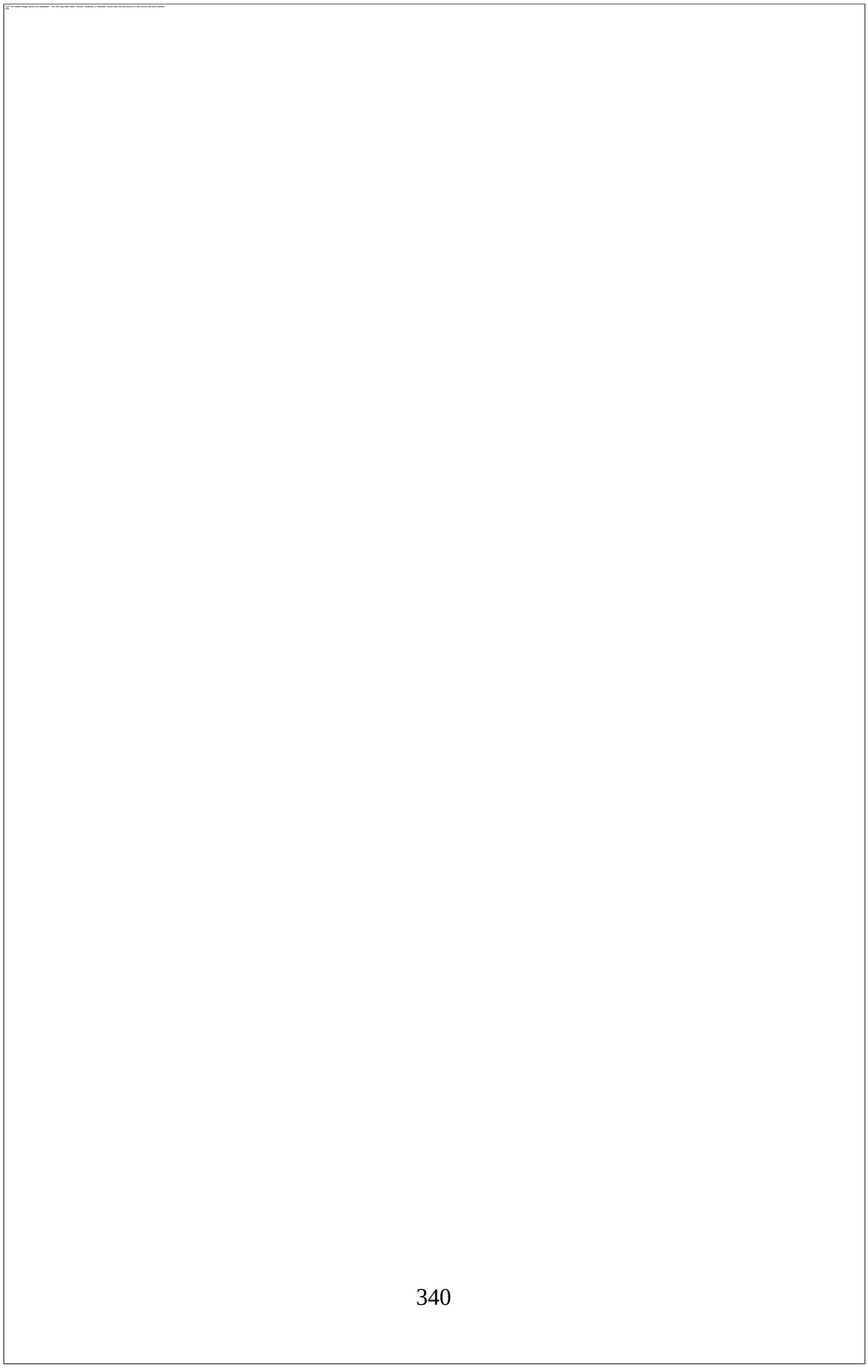
7.15.6.78 int FlowField::Nz () const [inline]

References Nz_.

Referenced by assignOrrSommField(), convectionNL(), cross(), curl(), diff(), div(), divergenceNL(), DNSAlgorithm::DNSAlgorithm(), dot(), energy(), grad(), interpolate(), L1Dist(), L1Norm(), lapl(), linearizedNL(), LinfDist(), LinfNorm(), norm(), norm2(), outer(), PPEDNS::PPEDNS(), PPEDNS::push(), rotationalNL(), skewsymmetricNL_task2::Run(), skewsymmetricNL_task::Run(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), up2q(), uq2p(), PressureSolver::verify(), xdiff(), ydiff(), and zdiff().

```
480 {return Nz_;}
```

Here is the caller graph for this function:



7.15.6.79 const Real & FlowField::operator() (int *nx*, int *ny*, int *nz*, int *i*, int *j*) const [inline]

References flatten(), Physical, rdata_, and xzstate_.

```
398                                                 {
399   assert(xzstate_==Physical);
400   return rdata_[flatten(nx,ny,nz,i,j)];
401 }
```

Here is the call graph for this function:

7.15.6.80 Real & FlowField::operator() (int *nx*, int *ny*, int *nz*, int *i*, int *j*) [inline]

References flatten(), Physical, rdata_, and xzstate_.

```
402                                                 {
403   assert(xzstate_==Physical);
404   return rdata_[flatten(nx,ny,nz,i,j)];
405 }
```

Here is the call graph for this function:

7.15.6.81 const Real & FlowField::operator() (int *nx*, int *ny*, int *nz*, int *i*) const [inline]

References flatten(), Physical, rdata_, and xzstate_.

```
361                                                 {
362   assert(xzstate_==Physical);
363   return rdata_[flatten(nx,ny,nz,i)];
364 }
```

Here is the call graph for this function:

7.15.6.82 Real & FlowField::operator() (int *nx*, int *ny*, int *nz*, int *i*) [inline]

References flatten(), Physical, rdata_, and xzstate_.

```
365                                                 {
366   assert(xzstate_==Physical);
367   return rdata_[flatten(nx,ny,nz,i)];
368 }
```

Here is the call graph for this function:

7.15.6.83 FlowField & FlowField::operator*= (const FlowField & *u*)

References cdata_, congruent(), Nd_, Nx_, Ny_, Nzpad2_, Nzpad_, rdata_, Spectral, and xzstate_.

```
918                                             {
919   assert(congruent(U));
920   if (xzstate_ == Spectral) {
921     int Ntotal = Nx_ * Ny_ * Nzpad2_ * Nd_;
922     for (int i=0; i<Ntotal; ++i)
923       cdata_[i] *= U.cdata_[i];
924   }
925   else {
926     int Ntotal = Nx_ * Ny_ * Nzpad_ * Nd_;
927     for (int i=0; i<Ntotal; ++i)
928       rdata_[i] *= U.rdata_[i];
929   }
930   return *this;
931 }
```

Here is the call graph for this function:

7.15.6.84 FlowField & FlowField::operator*= (Complex *x*)

References cdata_, Nd_, Nx_, Ny_, Nzpad2_, Spectral, and xzstate_.

```
850                                     {
```

```
851   assert(xzstate_ == Spectral);
852   int Ntotal = Nx_ * Ny_ * Nzpad2_ * Nd_;
853   for (int i=0; i<Ntotal; ++i)
854     cdata_[i] *= z;
855   return *this;
856 }
```

7.15.6.85 FlowField & FlowField::operator*= (Real *x*)

References Nd_, Nx_, Ny_, Nzpad_, and rdata_.

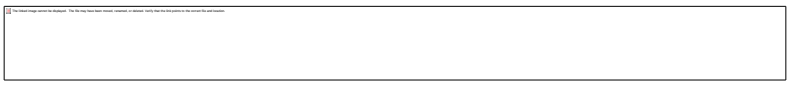
```
843                                 {
844   int Ntotal = Nx_ * Ny_ * Nzpad_ * Nd_;
845   for (int i=0; i<Ntotal; ++i)
846     rdata_[i] *= x;
847   return *this;
848 }
```

7.15.6.86 FlowField & FlowField::operator+= (const FlowField & *u*)

References congruent(), Nd_, Nx_, Ny_, Nzpad_, and rdata_.

```
904                                          {
905   assert(congruent(U));
906   int Ntotal = Nx_ * Ny_ * Nzpad_ * Nd_;
907   for (int i=0; i<Ntotal; ++i)
908     rdata_[i] += U.rdata_[i];
909   return *this;
910 }
```

Here is the call graph for this function:



7.15.6.87 FlowField & FlowField::operator+= (const BasisFunc & *U*)

References addProfile().

```
894                                          {
895   addProfile(profile);
896   return *this;
897 }
```

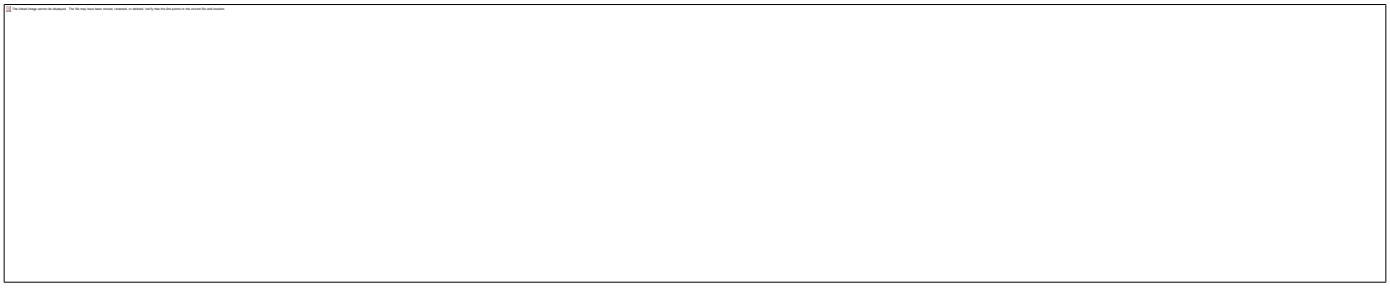
Here is the call graph for this function:

7.15.6.88 FlowField & FlowField::operator+= (const ComplexChebyCoeff & *U*)

References cmplx(), ComplexChebyCoeff::N(), Ny_, Spectral, ComplexChebyCoeff::state(), xzstate_, and ystate_.

```
876                                             {
877   assert(xzstate_ == Spectral);
878   assert(ystate_ == U.state());
879   assert(Ny_ == U.N());
880   for (int ny=0; ny<Ny_; ++ny)
881     cmplx(0, ny, 0, 0) += U[ny];
882   return *this;
883 }
```

Here is the call graph for this function:

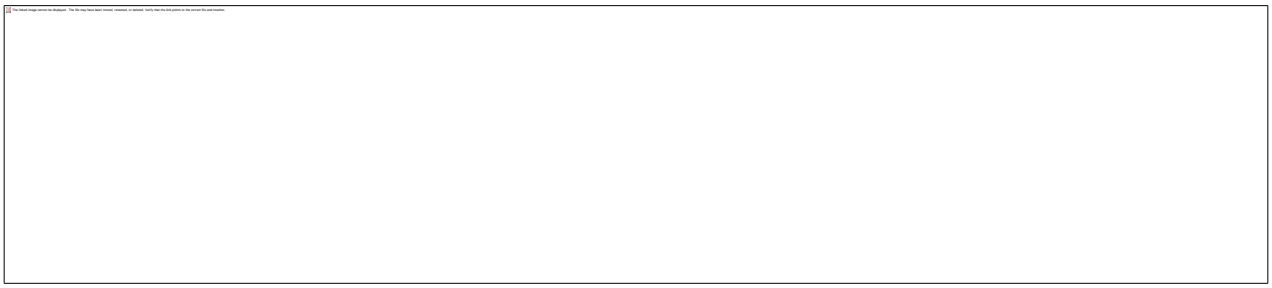


7.15.6.89 FlowField & FlowField::operator+= (const ChebyCoeff & *U*)

References cmplx(), ChebyCoeff::N(), Ny_, Spectral, ChebyCoeff::state(), xzstate_, and ystate_.

```
858                                          {
859   assert(xzstate_ == Spectral);
860   assert(ystate_ == U.state());
861   assert(Ny_ == U.N());
862   for (int ny=0; ny<Ny_; ++ny)
863     cmplx(0, ny, 0, 0) += Complex(U[ny], 0.0);
864   return *this;
865 }
```

Here is the call graph for this function:

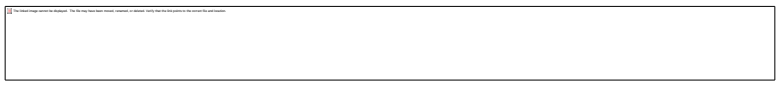


7.15.6.90 FlowField & FlowField::operator-= (const FlowField & *u*)

References congruent(), Nd_, Nx_, Ny_, Nzpad_, and rdata_.

```
911                                          {
912   assert(congruent(U));
913   int Ntotal = Nx_ * Ny_ * Nzpad_ * Nd_;
914   for (int i=0; i<Ntotal; ++i)
915     rdata_[i] -= U.rdata_[i];
916   return *this;
917 }
```

Here is the call graph for this function:



7.15.6.91 FlowField & FlowField::operator-= (const BasisFunc & *U*)

References addProfile().

```
898                                          {
899   BasisFunc copy(profile);
```

```
900   copy *= -1;
901   addProfile(copy);
902   return *this;
903 }
```

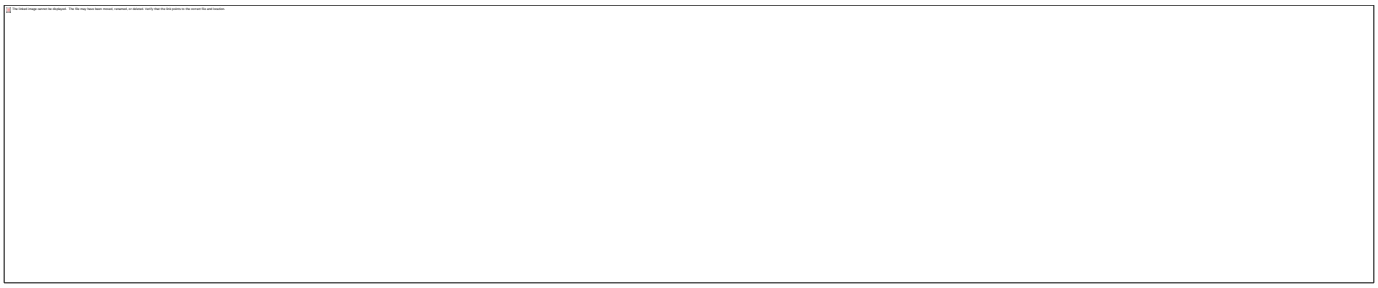
Here is the call graph for this function:

7.15.6.92 FlowField & FlowField::operator-= (const ComplexChebyCoeff & *U*)

References cmplx(), ComplexChebyCoeff::N(), Ny_, Spectral, ComplexChebyCoeff::state(), xzstate_, and ystate_.

```
885                                                 {
886   assert(xzstate_ == Spectral);
887   assert(ystate_ == U.state());
888   assert(Ny_ == U.N());
889   for (int ny=0; ny<Ny_; ++ny)
890     cmplx(0, ny, 0, 0) -= U[ny];
891   return *this;
892 }
```

Here is the call graph for this function:

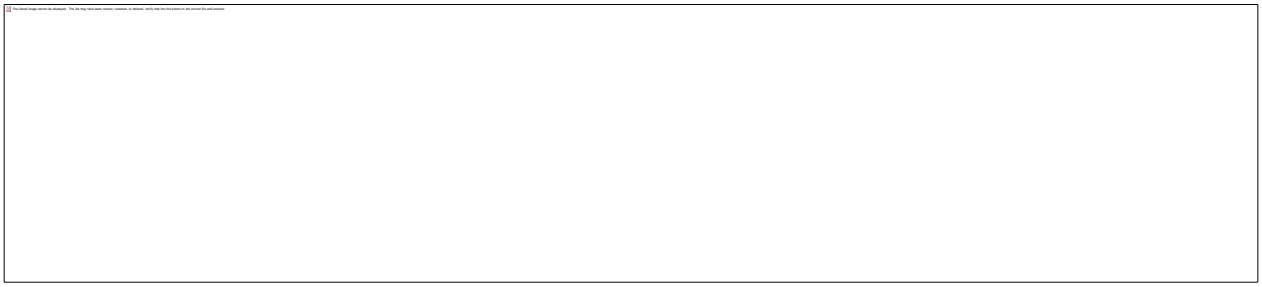


7.15.6.93 FlowField & FlowField::operator-= (const ChebyCoeff & *U*)

References cmplx(), ChebyCoeff::N(), Ny_, Spectral, ChebyCoeff::state(), xzstate_, and ystate_.

```
867                                         {
868   assert(xzstate_ == Spectral);
869   assert(ystate_ == U.state());
870   assert(Ny_ == U.N());
871   for (int ny=0; ny<Ny_; ++ny)
872     cmplx(0, ny, 0, 0) -= Complex(U[ny], 0.0);
873   return *this;
874 }
```

Here is the call graph for this function:



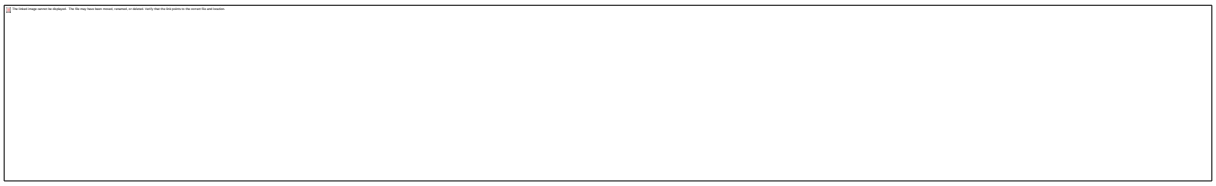
7.15.6.94 FlowField & FlowField::operator= (const FlowField & *u*)

References a_, b_, dealiased_, Lx_, Lz_, Nd_, Nx_, Ny_, Nz_, Nzpad_, rdata_, resize(), setState(), xzstate_, and ystate_.

```
379                                         {
380   resize(f.Nx_, f.Ny_, f.Nz_, f.Nd_, f.Lx_, f.Lz_, f.a_, f.b_);
381   setState(f.xzstate_, f.ystate_);
382   dealiased_ = f.dealiased_;
383   int Ntotal = Nx_ * Ny_ * Nzpad_ * Nd_;
384   for (int i=0; i<Ntotal; ++i)
385     rdata_[i] = f.rdata_[i];
```

```
386   return *this;
387 }
```

Here is the call graph for this function:

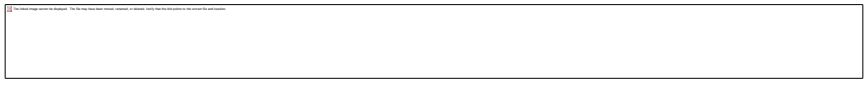


7.15.6.95 void FlowField::optimizeFFTW (int *fftw_flags* = FFTW_MEASURE)

References fftw_initialize(), xz_iplan_, xz_plan_, and y_plan_.

```
707                                     {
708   if (y_plan_) fftw_destroy_plan(y_plan_);
709   if (xz_plan_) fftw_destroy_plan(xz_plan_);
710   if (xz_iplan_) fftw_destroy_plan(xz_iplan_);
711
712   flags = flags | FFTW_DESTROY_INPUT;
713
714   fftw_initialize(flags);
715 }
```

Here is the call graph for this function:

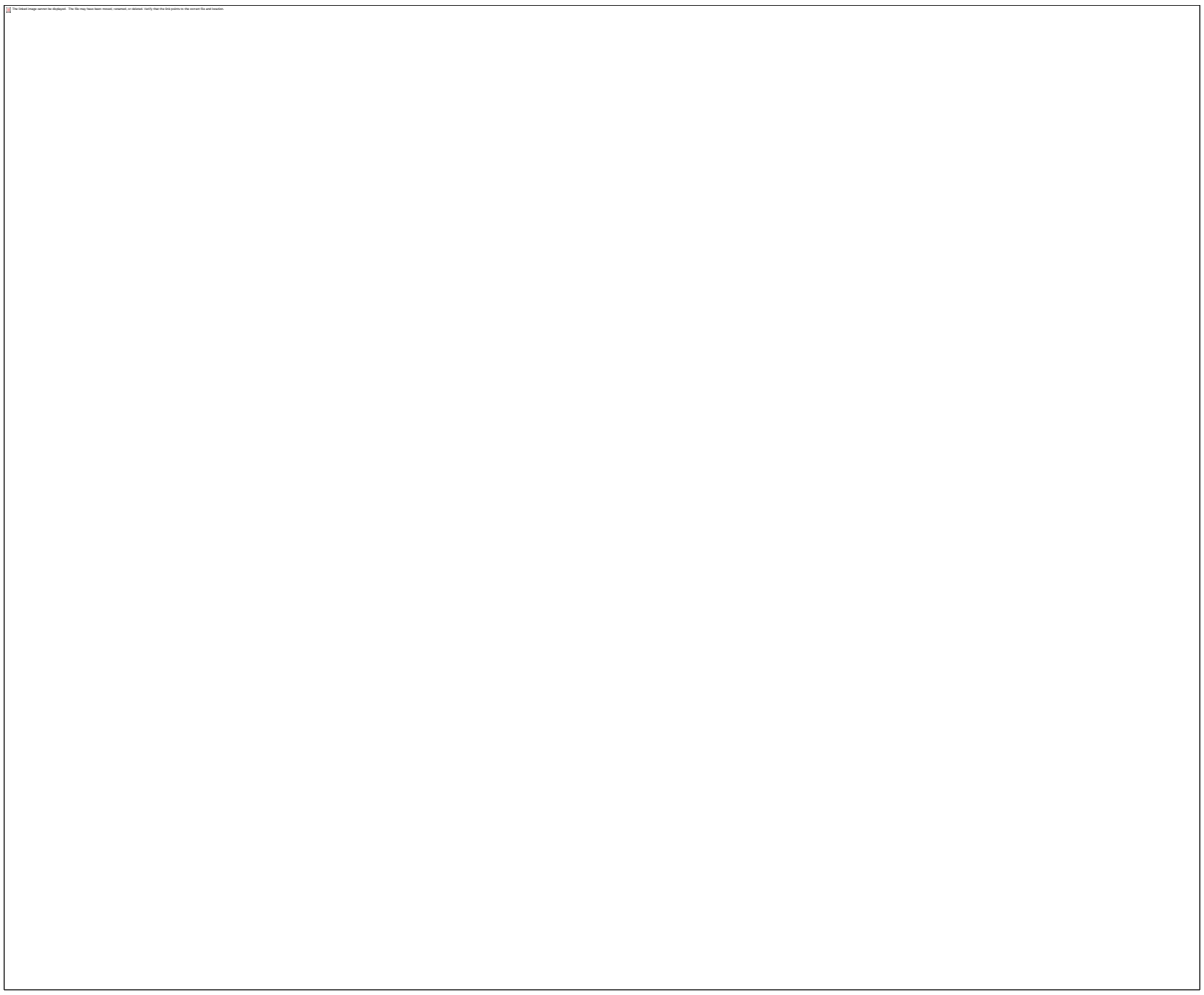


7.15.6.96 void FlowField::perturb (Real *magnitude*, Real *spectralDecay*)

References addPerturbations(), kxmax(), and kzmax().

```
1092                                        {
1093   addPerturbations(kxmax(), kzmax(), mag, decay);
1094 }
```

Here is the call graph for this function:

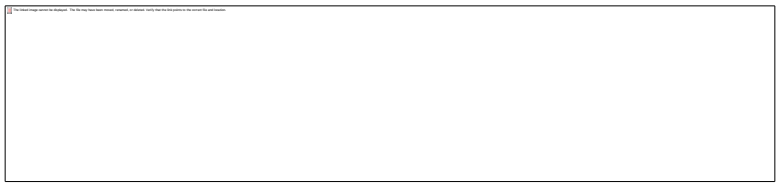


7.15.6.97 void FlowField::print (int *nx*, int *ny*, int *nz*, int *i*) const

References cdata_, complex_flatten(), flatten(), rdata_, Spectral, and xzstate_.

```
1197 {
1198   if (xzstate_ == Spectral)
1199   {
1200     cout << "FlowField::print() real view " << endl;
1201     cout << rdata_[flatten(nx,ny,nz,i)] << ' ';
1202   }
1203   else
1204   {
1205     cout << "complex view " << endl;
1206     cout << "complex_flatten=" << complex_flatten(nx,ny,nz,i) << " " << endl;
1207     cout <<" cdata= " << cdata_[complex_flatten(nx,ny,nz,i)] << ' ';
1208   }
1209 }
```

Here is the call graph for this function:



7.15.6.98 void FlowField::print () const

References cdata_, complex_flatten(), flatten(), Lx_, Lz_, Nd_, Nx_, Ny_, Nz_, Nzpad_, rdata_, Spectral, xzstate_, and ystate_.

```
1147                        {
1148
1149  //IG
1150  freopen("igor_log_out.txt","w",stdout);
1151
1152   cout << Nx_ << " x " << Ny_ << " x " << Nz_ << endl;
1153   cout << "[0, " << Lx_ << "] x [-1, 1] x [0, " << Lz_ << "]" << endl;
1154   cout << xzstate_ << " x " << ystate_ << " x " << xzstate_ << endl;
1155   cout << xzstate_ << " x " << ystate_ << " x " << xzstate_ << endl;
1156
1157   if (xzstate_ == Spectral) {
1158   cout << "FlowField::print() real view " << endl;
1159     for (int i=0; i<Nd_; ++i) {
1160       for (int ny=0; ny<Ny_; ++ny) {
1161         for (int nx=0; nx<Nx_; ++nx) {
1162           cout << "i=" << i << " ny=" << ny << " nx= " << nx << ' ';
1163           int nz; // MSVC++ FOR-SCOPE BUG
1164           for (nz=0; nz<Nz_; ++nz)
1165             cout << rdata_[flatten(nx,ny,nz,i)] << ' ';
1166           cout << " pad : ";
1167           for (nz=Nz_; nz<Nzpad_; ++nz)
1168             cout << rdata_[flatten(nx,ny,nz,i)] << ' ';
1169           cout << endl;
1170         }
1171       }
1172     }
1173   }
1174   else {
1175     cout << "complex view Nd_=" << Nd_ << "Ny_=" << Ny_ << "Nx=" <<Nx_ <<
"Nz=" << Nz_ <<endl;
1176     for (int i=0; i<Nd_; ++i) {
1177       for (int ny=0; ny<Ny_; ++ny) {
1178         for (int nx=0; nx<Nx_; ++nx) {
```

```
1179          for (int nz=0; nz<Nz_/2; ++nz)
1180          {
1181           {
1182          cout << "i=" << i << " ny=" << ny << " nx=" << nx << ' ' << "nz="<<nz << " ";
1183          cout <<" flatten="<<complex_flatten(nx,ny,nz,i) <<" cdata= "
<< cdata_[complex_flatten(nx,ny,nz,i)] << ' ';
1184          cout << endl;
1185          cout.flush();
1186          }
1187
1188          }
1189         }
1190        }
1191     }
1192   }
1193
1194 }
```

Here is the call graph for this function:

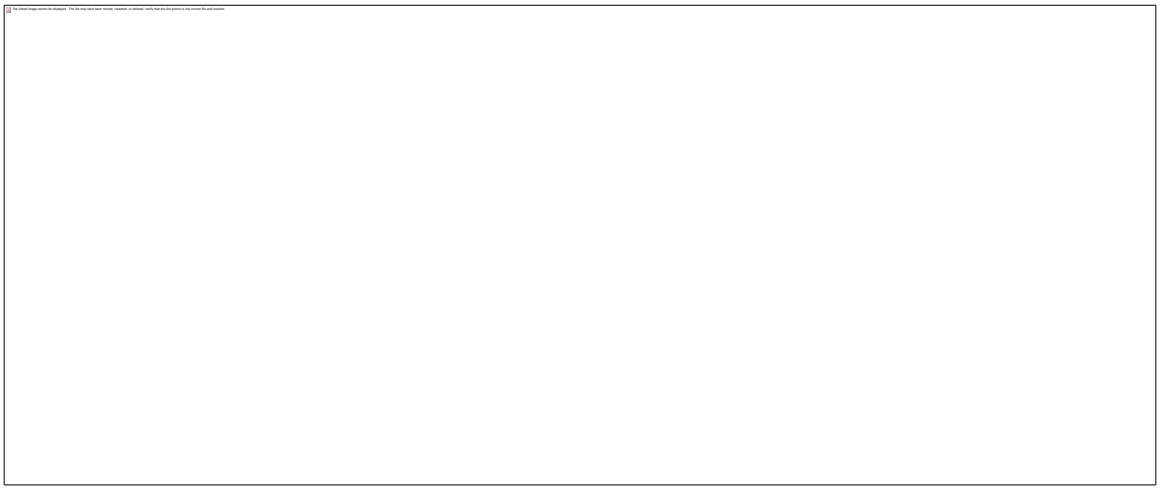
7.15.6.99 BasisFunc FlowField::profile (int *mx*, int *mz*) const

References a_, b_, cmplx(), kx(), kz(), lesser(), Lx_, Lz_, Nd(), Nd_, Ny_, Spectral, xzstate_, and ystate_.

```
756                                    {
757   if (xzstate_== Spectral) {
758     int k_x = kx(mx);
759     int k_z = kz(mz);
760     BasisFunc rtn(Ny_, k_x, k_z, Lx_, Lz_, a_, b_, ystate_);
761
762     int Nd = lesser(Nd_, 3);
763     for (int i=0; i<Nd; ++i)
764       for (int ny=0; ny<Ny_; ++ny)
765         rtn[i].set(ny, cmplx(mx, ny, mz, i));
766     return rtn;
767   }
768   else {
769     BasisFunc rtn(Ny_, 0, 0, Lx_, Lz_, a_, b_, ystate_);
770
```

```
771     int Nd = lesser(Nd_, 3);
772     for (int i=0; i<Nd; ++i)
773       for (int ny=0; ny<Ny_; ++ny)
774         rtn[i].re[ny] = (*this)(mx, ny, mz, i);
775     return rtn;
776   }
777 }
```

Here is the call graph for this function:



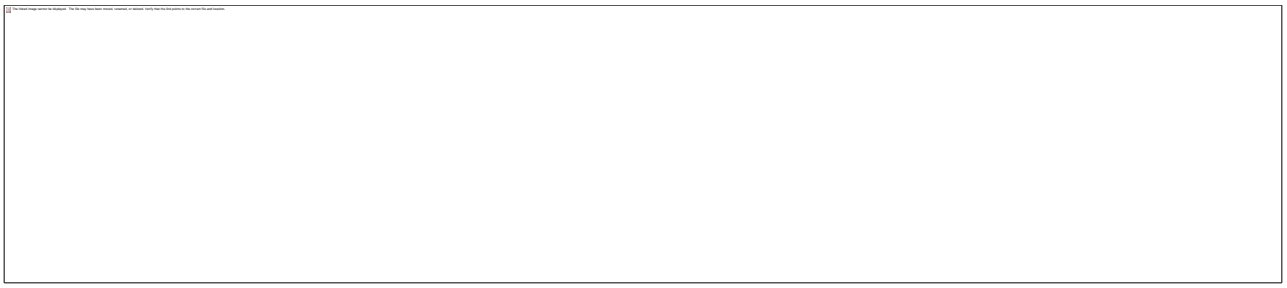
7.15.6.100 ComplexChebyCoeff FlowField::profile (int *mx*, int *mz*, int *i*) const

References a_, b_, cmplx(), Ny_, ComplexChebyCoeff::re, ComplexChebyCoeff::set(), Spectral, xzstate_, and ystate_.

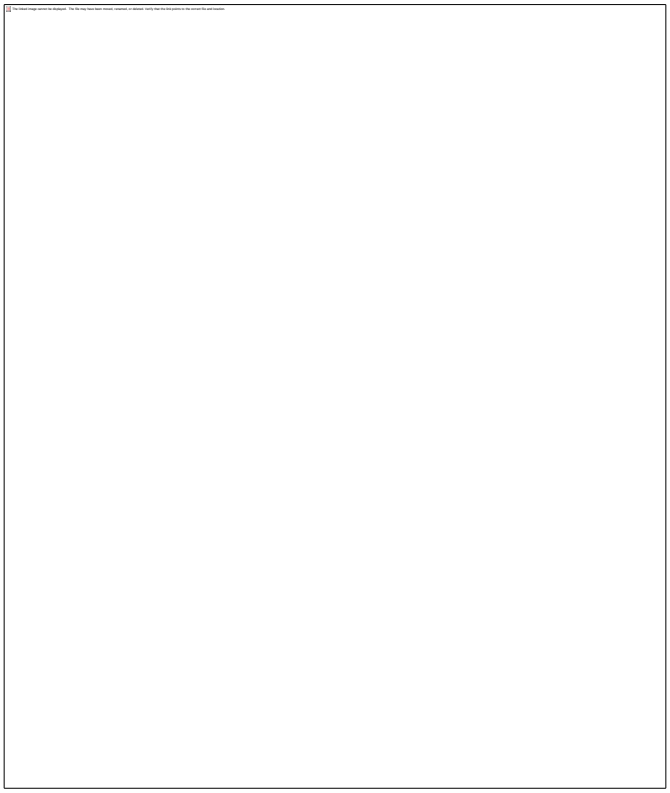
Referenced by divDist2(), divNorm2(), dudy_a(), dudy_b(), L1Dist(), L1Norm(), LinfDist(), and LinfNorm().

```
717                                                 {
718   ComplexChebyCoeff rtn(Ny_, a_, b_, ystate_);
719   if (xzstate_== Spectral) {
720     for (int ny=0; ny<Ny_; ++ny)
721       rtn.set(ny, cmplx(mx, ny, mz, i));
722   }
723   else
724     for (int ny=0; ny<Ny_; ++ny)
725       rtn.re[ny] = (*this)(mx, ny, mz, i);
726
727   return rtn;
728 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



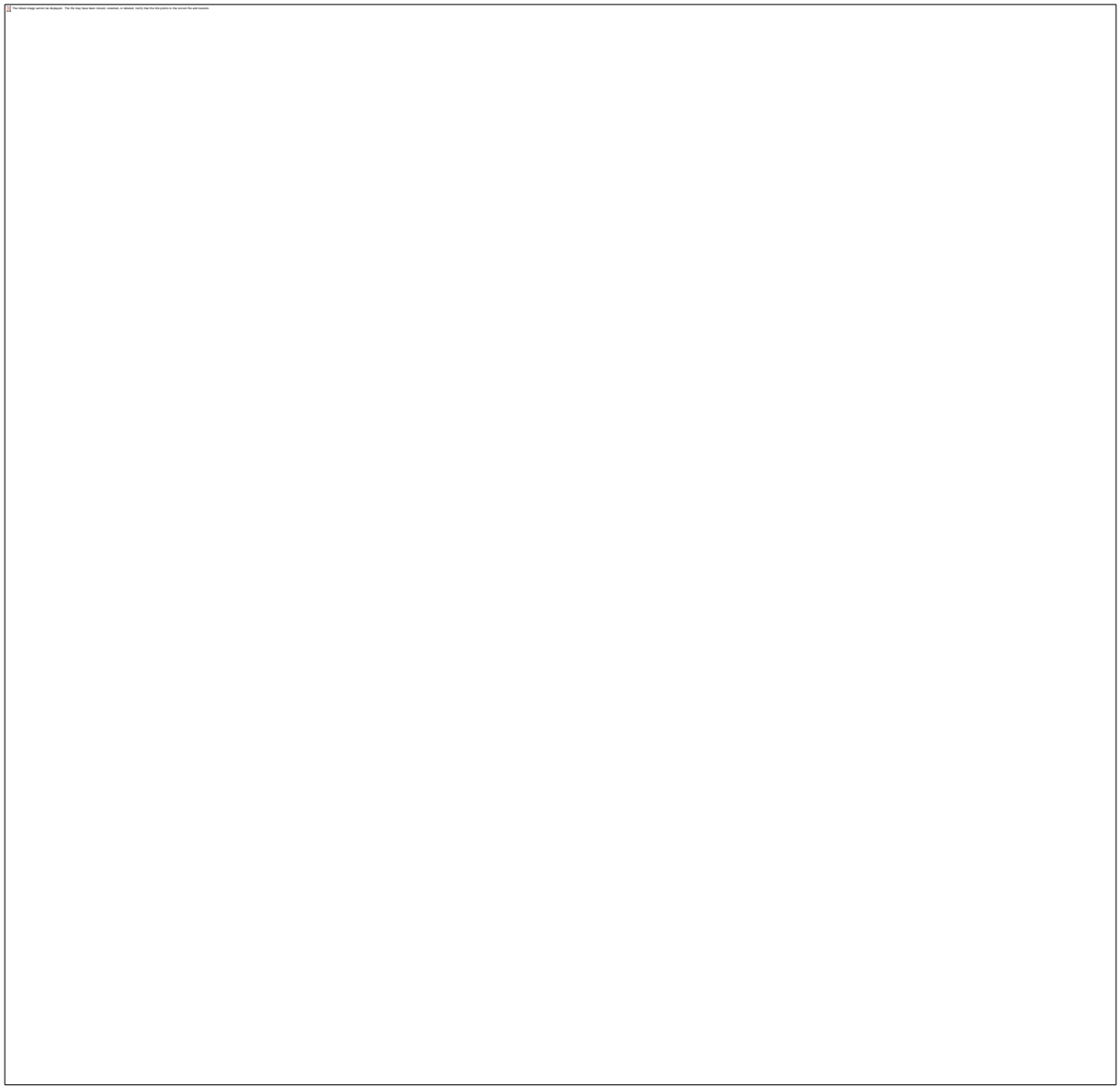
7.15.6.101 void FlowField::realfft_xz ()

References Nd_, Nx_, Ny_, Nz_, Nzpad_, Physical, rdata_, Spectral, xz_plan_, and xzstate_.
Referenced by assignOrrSommField(), and makeSpectral_xz().

```
933                     {
934   assert (xzstate_ == Physical);
935   fftw_execute(xz_plan_);
936   // args are (plan, howmany, in, istride, idist, ostride, odist)
937   //rfftwnd_real_to_complex(xz_plan_, Nd_*Ny_, rdata_, 1, Nx_*Nzpad_, 0,0,0);
938   int Ntotal = Nd_ * Nx_ * Ny_ * Nzpad_;
939   Real scale = 1.0/(Nx_*Nz_);
```

```
940   for (int i=0; i<Ntotal; ++i)
941     rdata_[i] *= scale;
942   xzstate_ = Spectral;
943 }
```

Here is the caller graph for this function:



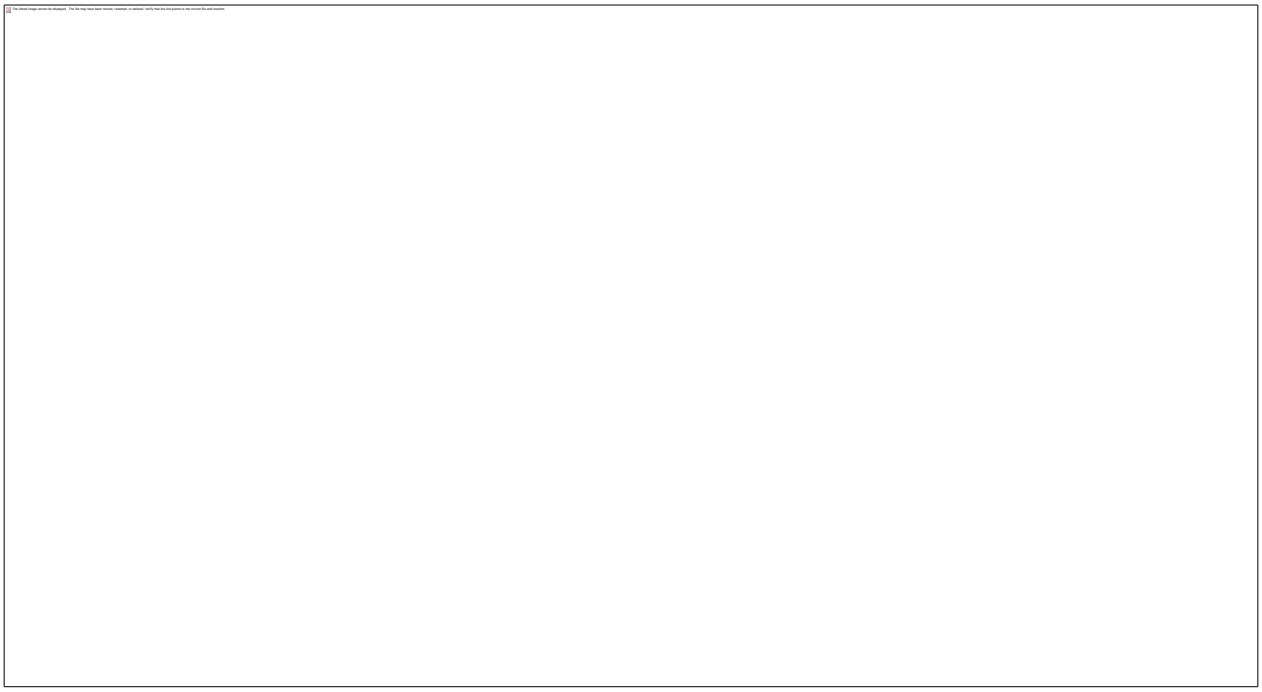
7.15.6.102 void FlowField::rescale (Real *Lx*, Real *Lz*)

References assertState(), cmplx(), Lx_, Lz_, Mx(), mx(), Mz(), mz(), numYmodes(), and Spectral.

```
495                                 {
```

```
496   assertState(Spectral, Spectral);
497   Real scalev = Lx_/Lx;
498   Real scalew = (Lx*Lz_)/(Lz*Lx_);
499
500   for (int ny=0; ny<numYmodes(); ++ ny)
501     for (int mx=0; mx<Mx(); ++mx)
502       for (int mz=0; mz<Mz(); ++ mz) {
503         cmplx(mx,ny,mz,1) *= scalev;
504         cmplx(mx,ny,mz,2) *= scalew;
505       }
506   Lx_ = Lx;
507   Lz_ = Lz;
508 }
```

Here is the call graph for this function:



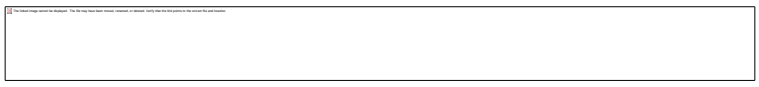
7.15.6.103 void FlowField::resize (int *Nx*, int *Ny*, int *Nz*, int *Nd*, Real *Lx*, Real *Lz*, Real *a*, Real *b*, int *fftw_flags* = FFTW_ESTIMATE)

References a_, b_, cdata_, fftw_initialize(), Lx_, Lz_, Nd_, Nx_, Ny_, Nz_, Nzpad2_, Nzpad_, rdata_, scratch_, xz_iplan_, xz_plan_, and y_plan_.

Referenced by convectionNL(), cross(), curl(), diff(), div(), divergenceNL(), DNSAlgorithm::DNSAlgorithm(), dot(), energy(), FlowField(), grad(), lapl(), linearizedNL(), norm(), norm2(), operator=(), outer(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), xdiff(), ydiff(), and zdiff().

```
402                                     {
403
404   // INEFFICIENT if geometry doesn't change. Should check.
405   if (scratch_) fftw_free(scratch_);
406   if (rdata_) fftw_free(rdata_);
407   if (y_plan_) fftw_destroy_plan(y_plan_);
408   if (xz_plan_) fftw_destroy_plan(xz_plan_);
409   if (xz_iplan_) fftw_destroy_plan(xz_iplan_);
410
411   Nx_ = Nx;
412   Ny_ = Ny;
413   Nz_ = Nz;
414   Nd_ = Nd;
415   Lx_ = Lx;
416   Lz_ = Lz;
417   a_ = a;
418   b_ = b;
419
420   assert(Nx_>=0);
421   assert(Ny_>=0);
422   assert(Nz_>=0);
423   assert(Nd_>=0);
424   assert(Lx_>=0);
425   assert(Lz_>=0);
426   assert(b_ >= a_);
427
428   Nzpad_ = 2*(Nz_/2+1);
429   Nzpad2_ = Nz_/2+1;
430   rdata_ = (Real*) fftw_malloc((Nx_*Ny_*Nzpad_*Nd_)*sizeof(Real));
431   cdata_ = (Complex*) rdata_;
432
433   int N = Nd_*Ny_*Nx_*Nzpad_;
434   for (int i=0; i<N; ++i)
435     rdata_[i] = 0.0;
436
437   scratch_ = (Real*) fftw_malloc(Ny_*sizeof(Real));
438
439   fftw_initialize(fftw_flags);
440 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.15.6.104 void FlowField::saveDivSpectrum (const std::string & *filebase*, bool *kxorder* = true) const

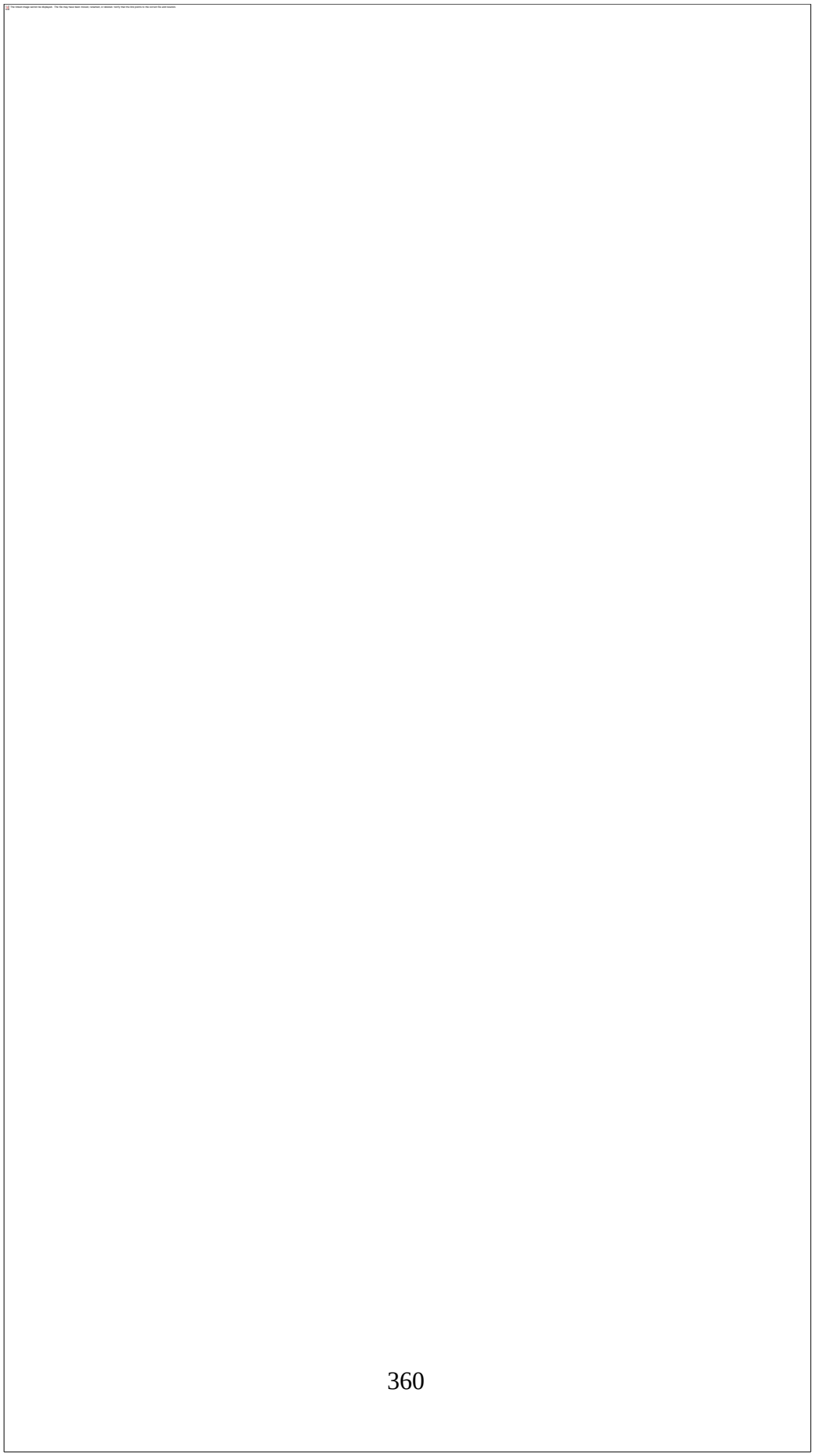
References a_, abs2(), b_, cmplx(), diff(), div(), kx(), kxmax(), kxmin(), kz(), kzmax(), kzmin(), Lx_, Lz_, Mx(), mx(), Mz(), mz(), Nd_, Ny_, pi, ComplexChebyCoeff::set(), Spectral, xzstate_, ystate_, and zero_last_mode().

```
1399                                                    {
```

```
1400  string filename(filebase);
1401  filename += string(".asc");
1402  ofstream os(filename.c_str());
1403
1404  assert(xzstate_ == Spectral && ystate_ == Spectral);
1405
1406  //assert(congruent(div));
1407  assert(Nd_ >= 3);
1408
1409  ComplexChebyCoeff v(Ny_, a_, b_, Spectral);
1410  ComplexChebyCoeff vy(Ny_, a_, b_, Spectral);
1411
1412  // Rely on compiler to pull loop invariants out of loops
1413  if (pretty) {
1414
1415    for (int kx=kxmin(); kx<kxmax(); ++kx) {
1416      Complex d_dx(0.0, 2*pi*kx/Lx_*zero_last_mode(kx,kxmax(),1));
1417
1418      for (int kz=kzmin(); kz<kzmax(); ++kz) {
1419        Complex d_dz(0.0, 2*pi*kz/Lz_*zero_last_mode(kz,kzmax(),1));
1420
1421        for (int ny=0; ny<Ny_; ++ny)
1422          v.set(ny, cmplx(mx(kx),ny,mz(kz),1));
1423        diff(v,vy);
1424
1425        Real div = 0.0;
1426        for (int ny=0; ny<Ny_; ++ny) {
1427          Complex ux = d_dx*cmplx(mx(kx),ny,mz(kz),0);
1428          Complex wz = d_dz*cmplx(mx(kx),ny,mz(kz),2);
1429          div += abs2(ux + vy[ny] + wz);
1430        }
1431        os << sqrt(div) << ' ';
1432      }
1433      os << endl;
1434    }
1435  }
1436  else {
1437    for (int mx=0; mx<Mx(); ++mx) {
1438      Complex d_dx(0.0, 2*pi*kx(mx)/Lx_*zero_last_mode(kx(mx),kxmax(),1));
1439
1440      for (int mz=0; mz<Mz(); ++mz) {
1441        Complex d_dz(0.0, 2*pi*kz(mz)/Lz_*zero_last_mode(kz(mz),kzmax(),1));
1442
1443        for (int ny=0; ny<Ny_; ++ny)
1444          v.set(ny, cmplx(mx,ny,mz,1));
1445        diff(v,vy);
```

```
1446
1447       Real div = 0.0;
1448       for (int ny=0; ny<Ny_; ++ny) {
1449         Complex ux = d_dx*cmplx(mx,ny,mz,0);
1450         Complex wz = d_dz*cmplx(mx,ny,mz,2);
1451         div += abs2(ux + vy[ny] + wz);
1452       }
1453       os << sqrt(div) << ' ';
1454     }
1455     os << endl;
1456   }
1457  }
1458  os.close();
1459 }
```

Here is the call graph for this function:

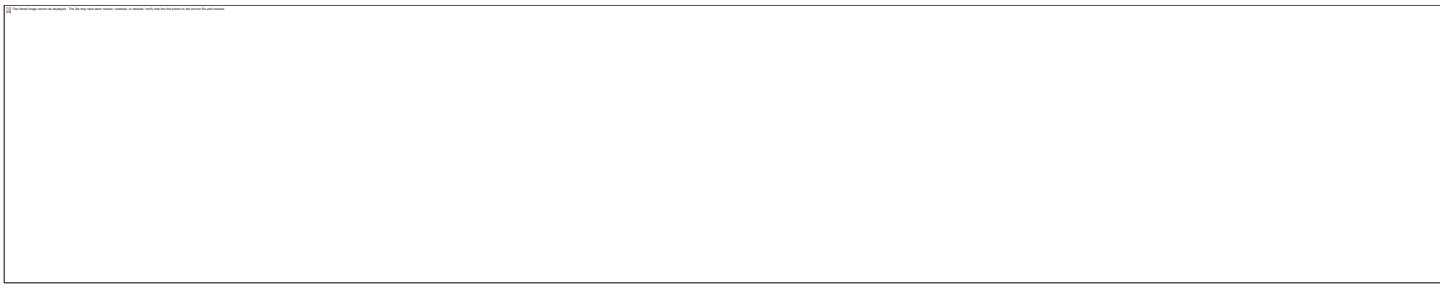


7.15.6.105 void FlowField::saveProfile (int *mx*, int *mz*, const std::string & *filebase*, const ChebyTransform & *t*) const

References a_, b_, Im(), ComplexChebyCoeff::makePhysical(), Nd_, Ny_, Physical, Re(), REAL_DIGITS, Spectral, xzstate_, and ystate_.

```
1269                                                                    {
1270   assert(xzstate_ == Spectral);
1271   string filename(filebase);
1272   if (Nd_ == 3)
1273     filename += string(".bf");  // this convention is unfortunate, need to fix
1274   else
1275     filename += string(".asc");
1276
1277   ofstream os(filename.c_str());
1278   os << setprecision(REAL_DIGITS);
1279
1280   if (ystate_ == Physical) {
1281     for (int ny=0; ny<Ny_; ++ny) {
1282       for (int i=0; i<Nd_; ++i) {
1283         Complex c = (*this).cmplx(mx, ny, mz, i);
1284         os << Re(c) << ' ' << Im(c) << ' ';
1285       }
1286       os << '\n';
1287     }
1288   }
1289   else {
1290     ComplexChebyCoeff* f = new ComplexChebyCoeff[Nd_];
1291     for (int i=0; i<Nd_; ++i) {
1292       f[i] = ComplexChebyCoeff(Ny_, a_, b_, Spectral);
1293       for (int ny=0; ny<Ny_; ++ny)
1294         f[i].set(ny, (*this).cmplx(mx, ny, mz, i));
1295       f[i].makePhysical(trans);
1296     }
1297     for (int ny=0; ny<Ny_; ++ny) {
1298       for (int i=0; i<Nd_; ++i)
1299         os << f[i].re[ny] << ' ' << f[i].im[ny] << ' ';
1300       os << '\n';
1301     }
1302   }
1303 }
```

Here is the call graph for this function:



7.15.6.106 void FlowField::saveProfile (int *mx*, int *mz*, const std::string & *filebase*) const

References Ny_.

```
1264                                                    {
1265   ChebyTransform trans(Ny_);
1266   saveProfile(mx,mz,filebase, trans);
1267 }
```

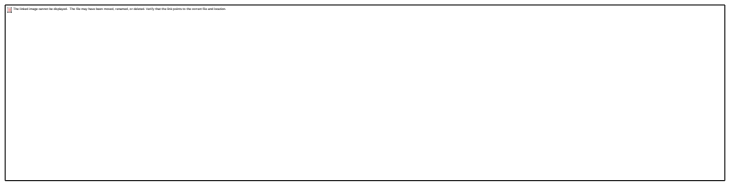
7.15.6.107 void FlowField::saveSlice (int *k*, int *i*, int *nk*, const std::string & *filebase*) const

References Nd_, Nx_, Ny_, Nz_, xzstate(), xzstate_, ystate(), and ystate_.

```
1214                                                    {
1215   assert(k>=0 && k<3);
1216   assert(i>=0 && i<Nd_);
1217
1218   string filename(filebase);
1219   filename += string(".asc");
1220   ofstream os(filename.c_str());
1221
1222   FlowField& u = (FlowField&) *this;  // cast away constness
1223   fieldstate xzstate = xzstate_;
1224   fieldstate ystate = ystate_;
1225
1226   u.makePhysical();
1227
1228   switch (k) {
1229   case 0:
1230     os << "% yz slice\n";
1231     os << "% (i,j)th elem is field at (x_n, y_i, z_j)\n";
1232     for (int ny=0; ny<Ny_; ++ny) {
1233       for (int nz=0; nz<Nz_; ++nz)
1234         os << u(nk, ny, nz, i) << ' ';
```

```
1235       os << u(nk, ny, 0, i) << '\n';  // repeat nz=0 to complete [0, Lz]
1236     }
1237     break;
1238   case 1: // xz slice
1239     os << "% xz slice\n";
1240     os << "% (i,j)th elem is field at (x_j, y_n, z_i)\n";
1241     for (int nz=0; nz<Nz_; ++nz) {
1242       for (int nx=0; nx<Nx_; ++nx)
1243         os << u(nx, nk, nz, i) << ' ';
1244       os << u(0, nk, nz, i) << '\n'; // repeat nx=0 to complete [0, Lx]
1245     }
1246     for (int nx=0; nx<Nx_; ++nx)   // repeat nz=0 to complete [0, Lz]
1247       os << u(nx, nk, 0, i) << ' ';
1248     os << u(0, nk, 0, i) << '\n';  // repeat nx=0 to complete [0, Lx]
1249
1250     break;
1251   case 2:
1252     os << "% yz slice\n";
1253     os << "% (i,j)th elem is field at (x_j, y_i, z_n)\n";
1254     for (int ny=0; ny<Ny_; ++ny) {
1255       for (int nx=0; nx<Nx_; ++nx)
1256         os << u(nx, ny, nk, i) << ' ';
1257       os << u(0, ny, nk, i) << '\n';
1258     }
1259     break;
1260   }
1261   u.makeState(xzstate, ystate);
1262 }
```

Here is the call graph for this function:



7.15.6.108 void FlowField::saveSpectrum (const std::string & *filebase*, bool *kxorder* = true) const

References a_, b_, kx(), kxmax(), kxmin(), kz(), kzmax(), kzmin(), L2Norm2(), Mx(), mx(), Mz(), mz(), Nd_, Ny_, ComplexChebyCoeff::set(), Spectral, xzstate_, and ystate_.

```
1341                                                {
1342   string filename(filebase);
1343   filename += string(".asc");
```

```
1344   ofstream os(filename.c_str());
1345
1346   assert(xzstate_ == Spectral && ystate_ == Spectral);
1347
1348   ComplexChebyCoeff u(Ny_,a_,b_,Spectral);
1349
1350   if (pretty) {
1351     for (int kx=kxmin(); kx<kxmax(); ++kx) {
1352       for (int kz=kzmin(); kz<kzmax(); ++kz) {
1353         Real e = 0.0;
1354         for (int i=0; i<Nd_; ++i) {
1355           for (int ny=0; ny<Ny_; ++ny)
1356             u.set(ny,this->cmplx(mx(kx),ny,mz(kz),i));
1357           e += L2Norm2(u);
1358         }
1359         os << sqrt(e) << ' ';
1360       }
1361       os << endl;
1362     }
1363   }
1364   else {
1365     for (int mx=0; mx<Mx(); ++mx) {
1366       for (int mz=0; mz<Mz(); ++mz) {
1367         Real e = 0.0;
1368         for (int i=0; i<Nd_; ++i) {
1369           for (int ny=0; ny<Ny_; ++ny)
1370             u.set(ny,this->cmplx(mx,ny,mz,i));
1371           e += L2Norm2(u);
1372         }
1373         os << sqrt(e) << ' ';
1374       }
1375       os << endl;
1376     }
1377   }
1378   os.close();
1379 }
```

Here is the call graph for this function:

7.15.6.109 void FlowField::saveSpectrum (const std::string & *filebase*, int *i*, int *ny* = -1, bool *kxorder* = true) const

References cmplx(), energy(), Im(), kx(), kxmax(), kxmin(), kz(), kzmax(), kzmin(), Mx(), mx(), Mz(), mz(), Re(), Spectral, and xzstate_.

```
1304                                                    {
1305   string filename(filebase);
1306   filename += string(".asc");
1307   ofstream os(filename.c_str());
```

```
1308
1309   bool sum = (ny == -1) ? true : false;
1310   assert(xzstate_ == Spectral);
1311
1312   if (pretty) {
1313     for (int kx=kxmin(); kx<=kxmax(); ++kx) {
1314       for (int kz=kzmin(); kz<=kzmax(); ++kz) {
1315         if (sum)
1316           os << sqrt(energy(mx(kx),mz(kz))) << ' ';
1317         else {
1318           Complex f = this->cmplx(mx(kx), ny, mz(kz), i);
1319           os << Re(f) << ' ' << Im(f) << ' ';
1320         }
1321       }
1322       os << endl;
1323     }
1324   }
1325   else {
1326     for (int mx=0; mx<Mx(); ++mx) {
1327       for (int mz=0; mz<Mz(); ++mz) {
1328         if (sum)
1329           os << sqrt(energy(mx,mz)) << ' ';
1330         else {
1331           Complex f = this->cmplx(mx, ny, mz, i);
1332           os << Re(f) << ' ' << Im(f) << ' ';
1333         }
1334       }
1335       os << endl;
1336     }
1337   }
1338   os.close();
1339 }
```

Here is the call graph for this function:

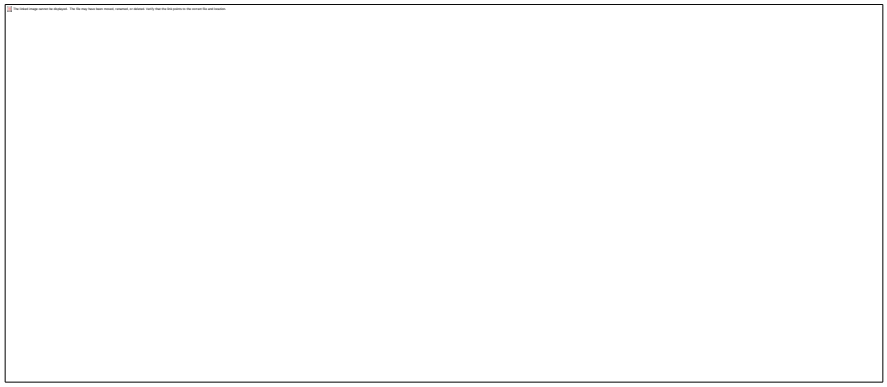
7.15.6.110 void FlowField::setDealiased (bool *b*)

References dealiased_.

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), and MultistepDNS::advance().

```
1137 {dealiased_ = b;}
```

Here is the caller graph for this function:



7.15.6.111 void FlowField::setState (fieldstate *xz*, fieldstate *y*)

References xzstate_, and ystate_.

Referenced by TurbStats::addData(), assignOrrSommField(), convectionNL(), cross(), curl(), diff(), div(), dot(), energy(), FlowField(), grad(), interpolate(), lapl(), linearizedNL(), norm(), norm2(), operator=(), outer(), rotationalNL(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), PressureSolver::solve(), PoissonSolver::solve(), xdiff(), ydiff(), and zdiff().

```
1668                                    {
1669   xzstate_ = xz;
1670   ystate_ = y;
1671 }
```

Here is the caller graph for this function:

7.15.6.112 void FlowField::setToZero ()

References Nd_, Nx_, Ny_, Nzpad_, and rdata_.

Referenced by TurbStats::addData(), assignOrrSommField(), CNABstyleDNS::CNABstyleDNS(), convectionNL(), cross(), curl(), diff(), div(), dot(), energy(), grad(), interpolate(), lapl(), linearizedNL(), MultistepDNS::MultistepDNS(), norm(), norm2(), outer(), PPEDNS::PPEDNS(), rotationalNL(), RungeKuttaDNS::RungeKuttaDNS(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), PressureSolver::solve(), PoissonSolver::solve(), xdiff(), ydiff(), and zdiff().

```
1139                        {
1140   int Ntotal = Nx_*Ny_*Nzpad_*Nd_;
1141   for (int i=0; i<Ntotal; ++i)
1142     rdata_[i] = 0.0;
1143 }
```

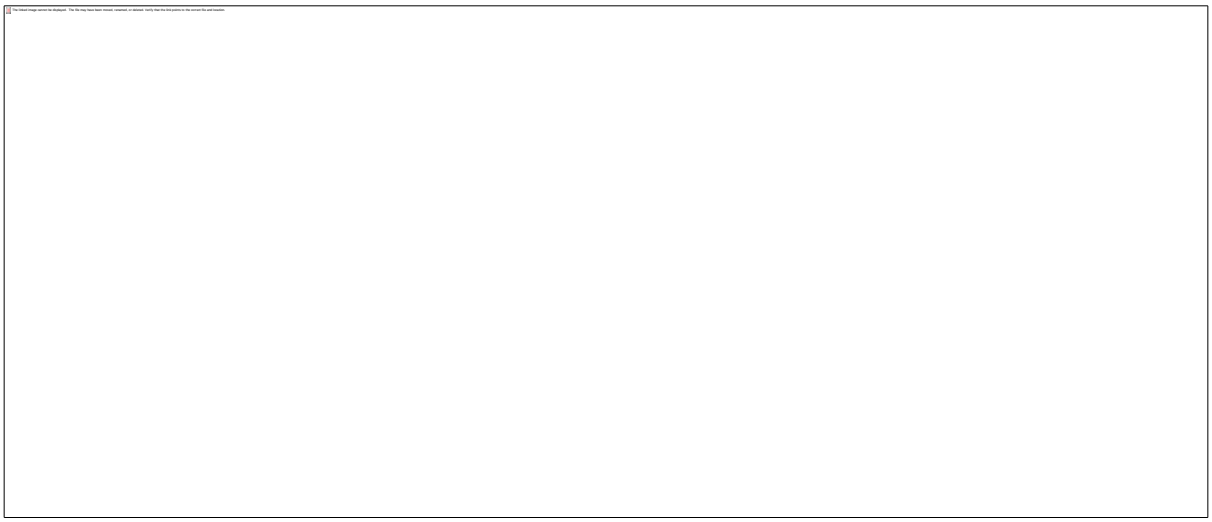
Here is the caller graph for this function:

7.15.6.113 void FlowField::translate (Real *delta_x*, Real *delta_z*)

References cmplx(), exp(), kx(), kz(), Lx_, Lz_, makeState(), Mx(), mx(), Mz(), mz(), Nd_, Ny_, pi, Spectral, xzstate_, and ystate_.

```
797                                     {
798   fieldstate xzs = xzstate_;
799   makeState(Spectral, ystate_);
800   for (int i=0; i<Nd_; ++i)
801     for (int ny=0; ny<Ny_; ++ny)
802       for (int mx=0; mx<Mx(); ++mx) {
803         Complex cx = exp(Complex(0.0, 2*pi*kx(mx)*delta_x/Lx_));
804         for (int mz=0; mz<Mz(); ++mz) {
805           Complex cz = exp(Complex(0.0, 2*pi*kz(mz)*delta_z/Lz_));
806           cmplx(mx,ny,mz,i) *= cx*cz;
807         }
808       }
809   makeState(xzs, ystate_);
810 }
```

Here is the call graph for this function:



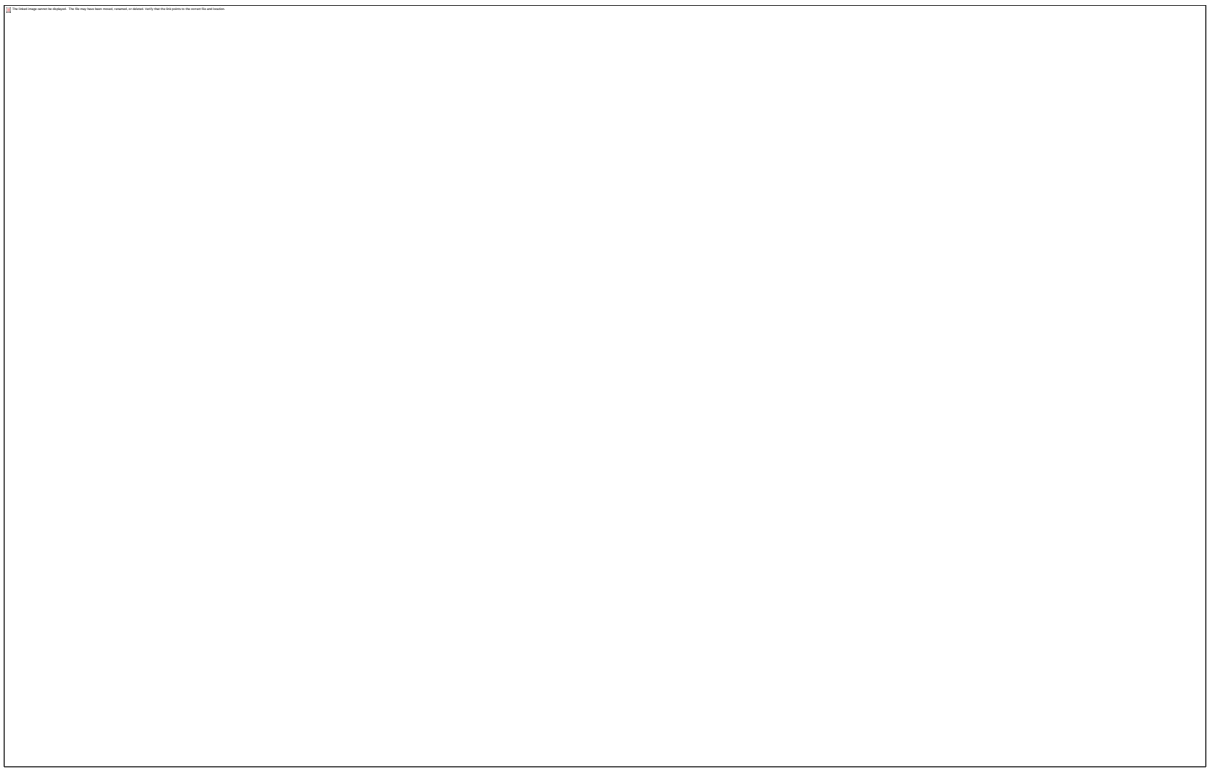
7.15.6.114 int FlowField::vectorDim () const [inline]

References Nd_.

Referenced by curl(), DNSAlgorithm::DNSAlgorithm(), grad(), L1Dist(), L1Norm(), L2Dist2(), L2InnerProduct(), L2Norm2(), LinfDist(), LinfNorm(), norm(), and norm2().

```
495 {return Nd_;}
```

Here is the caller graph for this function:



7.15.6.115 Real FlowField::x (int *nx*) const [inline]

References Lx_, and Nx_.

Referenced by xgridpts().

```
457                                 {
458   //assert(xzstate_ == Physical);
459   return nx*Lx_/Nx_;
460 }
```

Here is the caller graph for this function:

7.15.6.116 Vector FlowField::xgridpts () const

References Nx_, and x().

```
357                                 {
358   Vector xpts(Nx_);
359   for (int nx=0; nx<Nx_; ++nx)
360     xpts[nx] = x(nx);
```

```
361   return xpts;
362 }
```

Here is the call graph for this function:

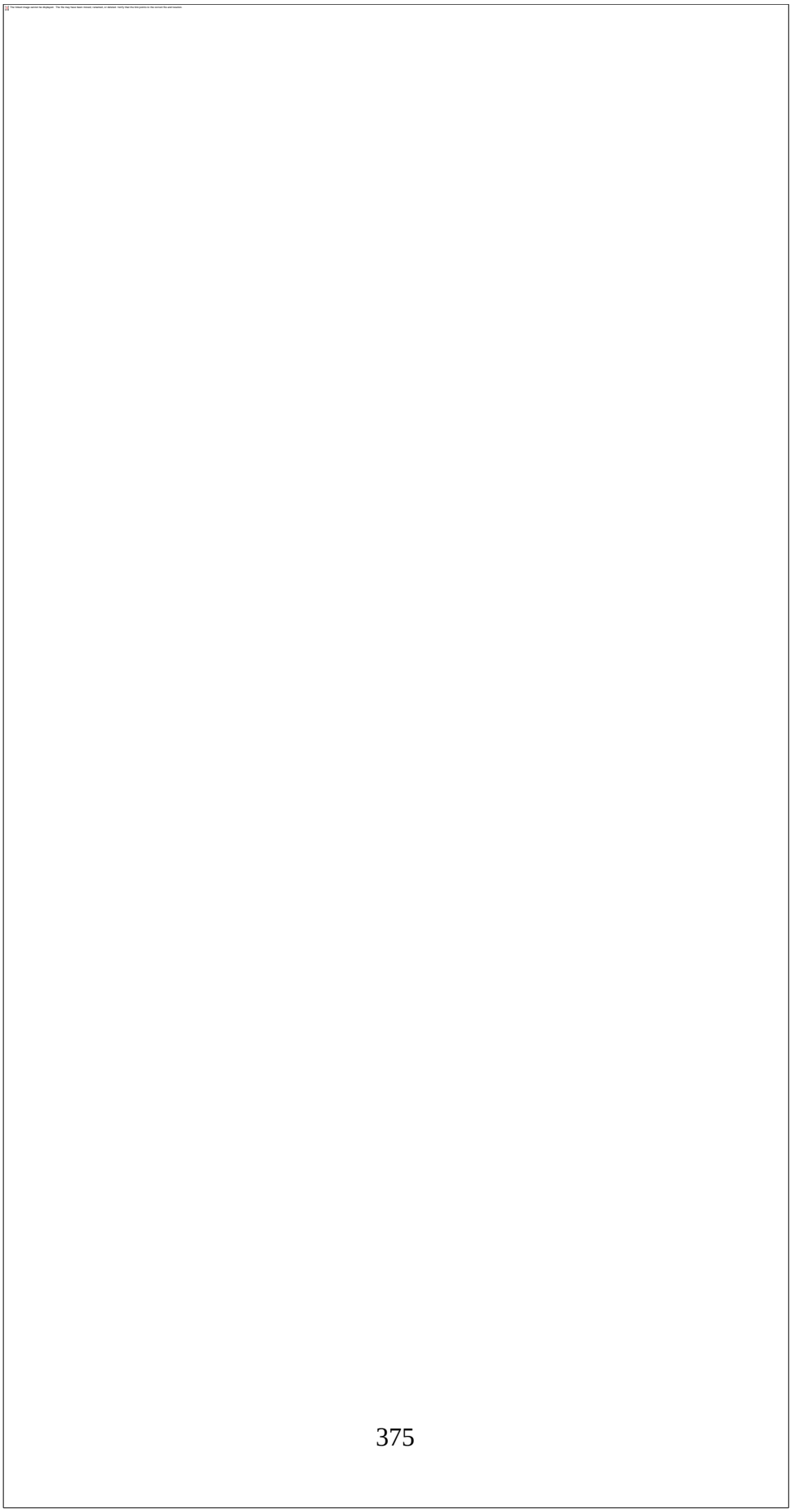
7.15.6.117 fieldstate FlowField::xzstate () const [inline]

References xzstate_.

Referenced by TurbStats::addData(), bcDist2(), bcNorm2(), CFLfactor(), changeBaseFlow(), convectionNL(), cross(), curl(), diff(), div(), divDist2(), divergenceNL(), divNorm2(), dot(), energy(), grad(), interpolate(), L1Dist(), L1Norm(), L2Dist2(), L2InnerProduct(), L2Norm2(), lapl(), linearizedNL(), LinfDist(), LinfNorm(), norm(), norm2(), outer(), rotationalNL(), saveSlice(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), PressureSolver::solve(), PoissonSolver::solve(), up2q(), uq2p(), PressureSolver::verify(), xdiff(), ydiff(), and zdiff().

```
498 {return xzstate_;}
```

Here is the caller graph for this function:


The inkwell image named has disappeared. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

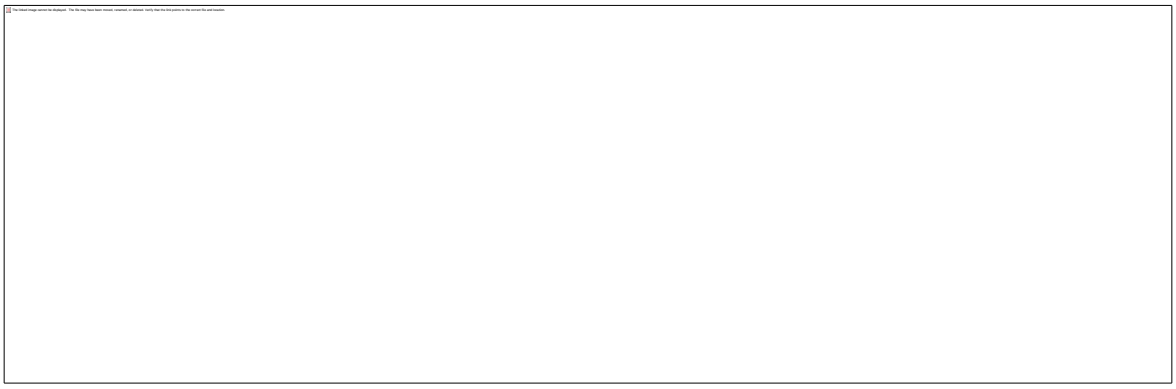
7.15.6.118 Real FlowField::y (int *ny*) const [inline]

References a_, b_, Ny_, and pi.

Referenced by assignOrrSommField(), and CFLfactor().

```
461                                {
462   //assert(ystate_ == Physical);
463   return 0.5*((b_+a_) + (b_-a_)*cos(pi*ny/(Ny_-1)));
464 }
```

Here is the caller graph for this function:



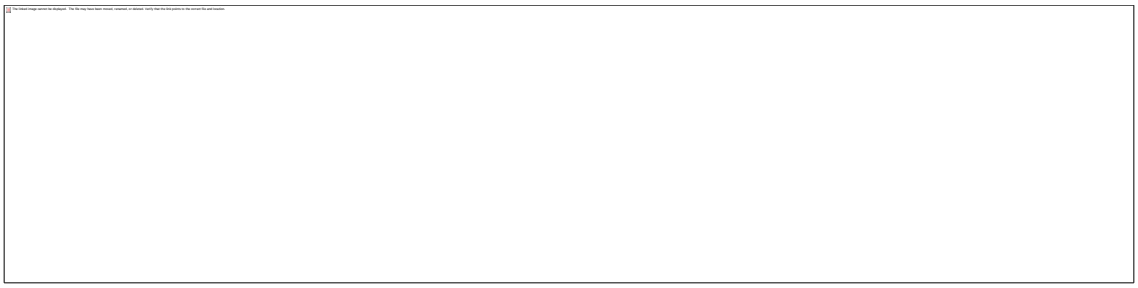
7.15.6.119 Vector FlowField::ygridpts () const

References a_, b_, Ny_, and pi.

Referenced by PressureSolver::solve().

```
363                                {
364   Vector ypts(Ny_);
365   Real c = 0.5*(b_+a_);
366   Real r = 0.5*(b_-a_);
367   Real piN = pi/(Ny_-1);
368   for (int ny=0; ny<Ny_; ++ny)
369     ypts[ny] = c + r*cos(piN*ny);
370   return ypts;
371 }
```

Here is the caller graph for this function:



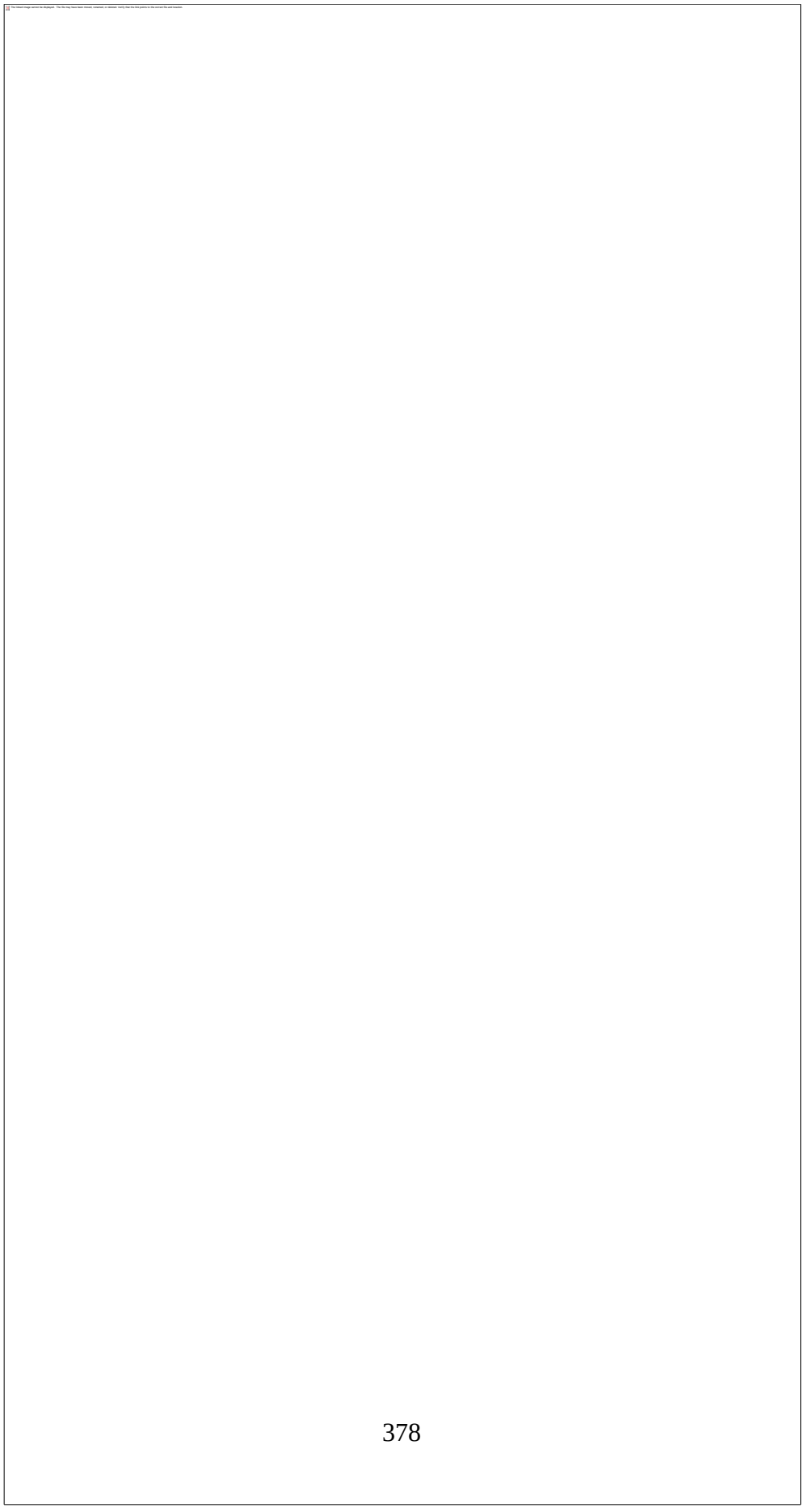
7.15.6.120 fieldstate FlowField::ystate () const [inline]

References ystate_.

Referenced by TurbStats::addData(), bcDist2(), bcNorm2(), CFLfactor(), changeBaseFlow(), convectionNL(), cross(), curl(), diff(), div(), divDist2(), divergenceNL(), divNorm2(), dot(), energy(), grad(), interpolate(), L1Dist(), L1Norm(), L2Dist2(), L2InnerProduct(), L2Norm2(), lapl(), linearizedNL(), LinfDist(), LinfNorm(), norm(), norm2(), outer(), rotationalNL(), saveSlice(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), PressureSolver::solve(), up2q(), uq2p(), PressureSolver::verify(), xdiff(), ydiff(), and zdiff().

```
499 {return ystate_;}
```

Here is the caller graph for this function:



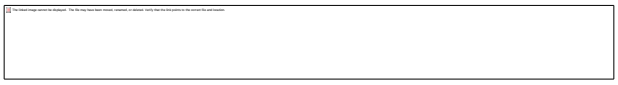
7.15.6.121 Real FlowField::z (int *nz*) const [inline]

References Lz_, and Nz_.

Referenced by zgridpts().

```
465                                {
466   //assert(zstate_ == Physical);
467   return nz*Lz_/Nz_;
468 }
```

Here is the caller graph for this function:

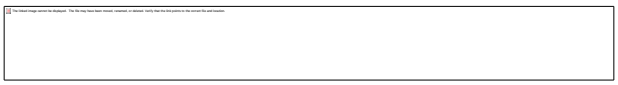


7.15.6.122 Vector FlowField::zgridpts () const

References Nz_, and z().

```
372                                {
373   Vector zpts(Nz_);
374   for (int nz=0; nz<Nz_; ++nz)
375     zpts[nz] = z(nz);
376   return zpts;
377 }
```

Here is the call graph for this function:



7.15.7 Friends And Related Function Documentation

7.15.7.1 void swap (FlowField & *f*, FlowField & *g*) [friend]

7.15.8 Member Data Documentation

7.15.8.1 Real FlowField::a_ [private]

Referenced by a(), addPerturbation(), asciiSave(), binarySave(), CFLfactor(), congruent(), energy(), FlowField(), geomCongruent(), Ly(), operator=(), profile(), resize(), saveDivSpectrum(), saveProfile(), saveSpectrum(), y(), and ygridpts().

7.15.8.2 Real FlowField::b_ [private]

Referenced by addPerturbation(), asciiSave(), b(), binarySave(), CFLfactor(), congruent(), energy(), FlowField(), geomCongruent(), Ly(), operator=(), profile(), resize(), saveDivSpectrum(), saveProfile(), saveSpectrum(), y(), and ygridpts().

7.15.8.3 Complex* FlowField::cdata_ [private]

Referenced by cmplx(), coeff(), fftw_initialize(), operator*=(), print(), resize(), and swap().

7.15.8.4 bool FlowField::dealiased_ [private]

Referenced by binarySave(), FlowField(), operator=(), and setDealiased().

7.15.8.5 Real FlowField::Lx_ [private]

Referenced by addPerturbation(), addPerturbations(), asciiSave(), binarySave(), CFLfactor(), congruent(), Dx(), energy(), FlowField(), geomCongruent(), Lx(), operator=(), print(), profile(), rescale(), resize(), saveDivSpectrum(), translate(), and x().

7.15.8.6 Real FlowField::Lz_ [private]

Referenced by addPerturbation(), addPerturbations(), asciiSave(), binarySave(), CFLfactor(), congruent(), Dz(), energy(), FlowField(), geomCongruent(), Lz(), operator=(), print(), profile(), rescale(), resize(), saveDivSpectrum(), translate(), and z().

7.15.8.7 int FlowField::Nd_ [private]

Referenced by addPerturbation(), addProfile(), asciiSave(), binarySave(), CFLfactor(), chebyfft_y(), complex_flatten(), congruent(), dump(), energy(), fftw_initialize(), flatten(), FlowField(), ichebyfft_y(), isNull(), Nd(), operator*=(), operator+=(), operator-=(), operator=(), print(), profile(), realfft_xz(), resize(), saveDivSpectrum(), saveProfile(), saveSlice(), saveSpectrum(), setToZero(), translate(), and vectorDim().

7.15.8.8 int FlowField::Nx_ [private]

Referenced by asciiSave(), binarySave(), CFLfactor(), chebyfft_y(), complex_flatten(), congruent(), dump(), fftw_initialize(), flatten(), FlowField(), geomCongruent(), ichebyfft_y(), isNull(), kx(), kxmax(), kxmaxDealiased(), kxmin(), Mx(), mx(), numXgridpts(), numXmodes(), Nx(), operator*=(), operator+=(), operator-=(), operator=(), print(), realfft_xz(), resize(), saveSlice(), setToZero(), x(), and xgridpts().

7.15.8.9 int FlowField::Ny_ [private]

Referenced by addPerturbation(), addProfile(), asciiSave(), binarySave(), CFLfactor(), chebyfft_y(), complex_flatten(), congruent(), dump(), energy(), fftw_initialize(), flatten(), FlowField(), geomCongruent(), ichebyfft_y(), isNull(), My(), numYgridpts(), numYmodes(), Ny(), operator*=(), operator+=(), operator-=(), operator=(), print(), profile(), realfft_xz(), resize(), saveDivSpectrum(), saveProfile(), saveSlice(), saveSpectrum(), setToZero(), translate(), y(), and ygridpts().

7.15.8.10 int FlowField::Nz_ [private]

Referenced by asciiSave(), binarySave(), CFLfactor(), congruent(), fftw_initialize(), FlowField(), geomCongruent(), isNull(), kzmax(), kzmaxDealiased(), numZgridpts(), Nz(), operator=(), print(), realfft_xz(), resize(), saveSlice(), z(), and zgridpts().

7.15.8.11 int FlowField::Nzpad2_ [private]

Referenced by complex_flatten(), fftw_initialize(), kz(), Mz(), mz(), numZmodes(), operator*=(), and resize().

7.15.8.12 int FlowField::Nzpad_ [private]

Referenced by binarySave(), chebyfft_y(), dump(), fftw_initialize(), flatten(), FlowField(), ichebyfft_y(), operator*=(), operator+=(), operator-=(), operator=(), print(), realfft_xz(), resize(), and setToZero().

7.15.8.13 Real* FlowField::rdata_ [private]

Referenced by binarySave(), chebyfft_y(), dump(), fftw_initialize(), FlowField(), ichebyfft_y(), operator()(), operator*=(), operator+=(), operator-=(), operator=(), print(), realfft_xz(), resize(), setToZero(), swap(), and ~FlowField().

7.15.8.14 Real* FlowField::scratch_ [private]

Referenced by chebyfft_y(), fftw_initialize(), ichebyfft_y(), resize(), and ~FlowField().

7.15.8.15 fftw_plan FlowField::xz_iplan_ [private]

Referenced by fftw_initialize(), irealfft_xz(), optimizeFFTW(), resize(), swap(), and ~FlowField().

7.15.8.16 fftw_plan FlowField::xz_plan_ [private]

Referenced by fftw_initialize(), optimizeFFTW(), realfft_xz(), resize(), swap(), and ~FlowField().

7.15.8.17 fieldstate FlowField::xzstate_ [private]

Referenced by addProfile(), asciiSave(), assertState(), binarySave(), CFLfactor(), cmplx(), coeff(), congruent(), energy(), FlowField(), irealfft_xz(), makePhysical_xz(), makeSpectral_xz(), operator()(), operator*=(), operator+=(), operator-=(), operator=(), print(), profile(), realfft_xz(), saveDivSpectrum(), saveProfile(), saveSlice(), saveSpectrum(), setState(), translate(), and xzstate().

7.15.8.18 fftw_plan FlowField::y_plan_ [private]

Referenced by chebyfft_y(), fftw_initialize(), ichebyfft_y(), optimizeFFTW(), resize(), swap(), and ~FlowField().

7.15.8.19 fieldstate FlowField::ystate_ [private]

Referenced by addProfile(), asciiSave(), assertState(), binarySave(), CFLfactor(), chebyfft_y(), congruent(), dudy_a(), dudy_b(), energy(), FlowField(), ichebyfft_y(), makePhysical_y(), makeSpectral_y(), operator+=(), operator-=(), operator=(), print(), profile(), saveDivSpectrum(), saveProfile(), saveSlice(), saveSpectrum(), setState(), translate(), and ystate().

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

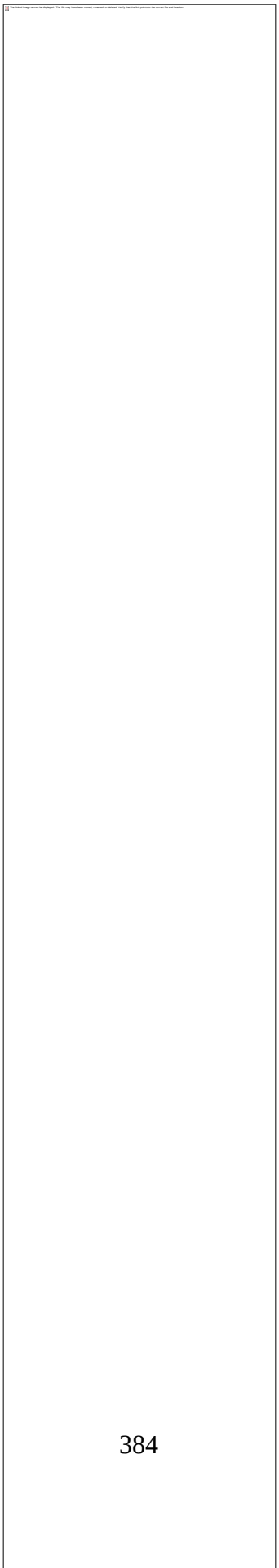
- /_cuda/channelflow-0.9.22/channelflow/flowfield.h
- /_cuda/channelflow-0.9.22/channelflow/flowfield.cpp

7.15.8.21

7.16 HelmholtzSolver Class Reference

#include <helmholtz.h>

Collaboration diagram for HelmholtzSolver:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.16.1 Public Member Functions

- HelmholtzSolver ()
- HelmholtzSolver (int numberModes, Real a, Real b, Real lambda, Real nu=1.0)
- void solve (ChebyCoeff &u, const ChebyCoeff &f, Real ua, Real ub) const
- void verify (const ChebyCoeff &u, const ChebyCoeff &f, Real ua, Real ub, bool verbose=false) const
- Real residual (const ChebyCoeff &u, const ChebyCoeff &f, Real ua, Real ub) const
- void solve (ChebyCoeff &u, Real &mu, const ChebyCoeff &f, Real umean, Real ua, Real ub) const
- void verify (ChebyCoeff &u, Real &mu, const ChebyCoeff &f, Real umean, Real ua, Real ub) const
- Real residual (const ChebyCoeff &u, Real mu, const ChebyCoeff &f, Real umean, Real ua, Real ub) const
- Real lambda () const

7.16.2 Private Member Functions

- int beta (int n) const
- int c (int n) const

7.16.3 Private Attributes

- int N_
- int nModes_
- int nEvenModes_
- int nOddModes_
- Real a_
- Real b_
- Real lambda_
- Real nu_
- BandedTridiag Ae_
- BandedTridiag Ao_
- BandedTridiag Be_
- BandedTridiag Bo_

7.16.4 Constructor & Destructor Documentation

7.16.4.1 HelmholtzSolver::HelmholtzSolver ()

```
37  :
38  N_(0),
39  nModes_(0),
40  nEvenModes_(0),
```

```
41   nOddModes_(0),
42   a_(0),
43   b_(0),
44   lambda_(0),
45   nu_(0),
46   Ae_(),
47   Ao_(),
48   Be_(),
49   Bo_()
50 {}
```

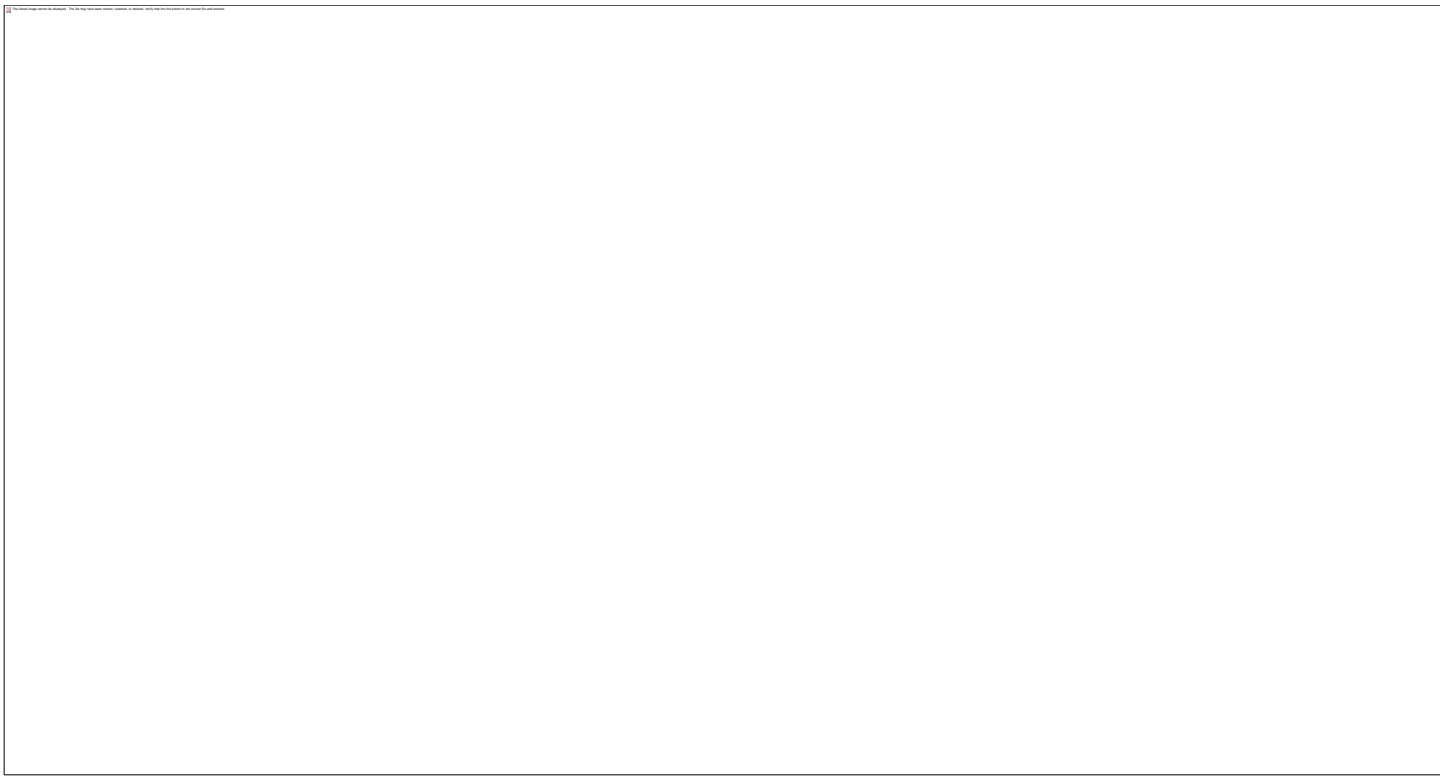
7.16.4.2 HelmholtzSolver::HelmholtzSolver (int *numberModes*, Real *a*, Real *b*, Real *lambda*, Real *nu* = 1.0)

References a_, Ae_, Ao_, b_, BandedTridiag::band(), Be_, beta(), Bo_, c(), BandedTridiag::diag(), BandedTridiag::lodiag(), nEvenModes_, nModes_, nOddModes_, square(), BandedTridiag::ULdecomp(), and BandedTridiag::updiag().

```
53   :
54   N_(numberModes - 1),
55   nModes_(numberModes),
56   nEvenModes_(N_/2 + 1),
57   nOddModes_(N_/2),
58   a_(a),
59   b_(b),
60   lambda_(lambda),
61   nu_(nu),
62   Ae_(nEvenModes_),
63   Ao_(nOddModes_),
64   Be_(nEvenModes_),
65   Bo_(nOddModes_)
66 {
67   assert(nModes_ % 2 == 1); // Is this required or assumed in C&H?
68   assert(nModes_ > 2);
69
70   Real nuscaled = nu/square((b_-a_)/2);
71
72   // Canuto & Hussaini eqn 5.1.24.
73   // Even mode matrices:
74   // Assign zeroth row of Ae_ and Be_
75   Be_.diag(0) = 1.0;
76   int i;
77   for (i=0; i<nEvenModes_; ++i)
78     Ae_.band(i) = 1.0;
79   // Assign first through rows of Ae_ and Be_
```

```
80   for (i=1; i<nEvenModes_; ++i) {
81     int n=2*i;
82     int nn=n*n;
83     Ae_.lodiag(i) =          -(c(n-2)*lambda)/(4*n*(n-1));
84     Ae_.diag(i)   = (nuscaled + (beta(n)*lambda)/(2*(nn-1)));
85     if (beta(n+2))
86       Ae_.updiag(i) =  -lambda/(4*n*(n+1));
87
88     Be_.lodiag(i) = ((Real)  c(n-2))/(4*n*(n-1));
89     Be_.diag(i)   = -((Real) beta(n))/(2*(nn-1));
90     if (beta(n+2))
91       Be_.updiag(i) = 1.0/(4*n*(n+1));
92   }
93
94   // Odd mode matrices
95   Bo_.diag(0) = 1.0;
96   for (i=0; i<nOddModes_; ++i)
97     Ao_.band(i) = 1.0;
98   for (i=1; i<nOddModes_; ++i) {
99     int n=2*i+1;
100      int nn=n*n;
101      Ao_.lodiag(i) =  -(c(n-2)*lambda)/(4*n*(n-1));
102      Ao_.diag(i)   =   (nuscaled + (beta(n)*lambda)/(2*(nn-1)));
103      if (beta(n+2))
104        Ao_.updiag(i) = -lambda/(4*n*(n+1));
105
106      Bo_.lodiag(i) = ((Real)  c(n-2))/(4*n*(n-1));
107      Bo_.diag(i)   = -((Real) beta(n))/(2*(nn-1));
108      if (beta(n+2))
109        Bo_.updiag(i) = 1.0/(4*n*(n+1));
110    }
111    Ae_.ULdecomp();
112    Ao_.ULdecomp();
113
114 }
```

Here is the call graph for this function:



7.16.5 Member Function Documentation

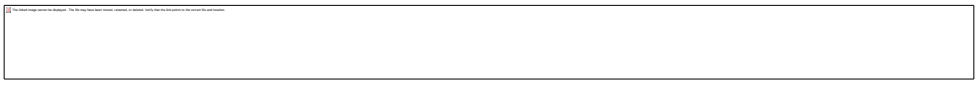
7.16.5.1 int HelmholtzSolver::beta (int *n*) const [inline, private]

References N_.

Referenced by HelmholtzSolver().

```
87 {return (n > N_-2) ? 0 : 1;}
```

Here is the caller graph for this function:



7.16.5.2 int HelmholtzSolver::c (int *n*) const [inline, private]

References N_.

Referenced by HelmholtzSolver().

```
88 {return (n==0 || n==N_) ? 2 : 1;}
```

Here is the caller graph for this function:

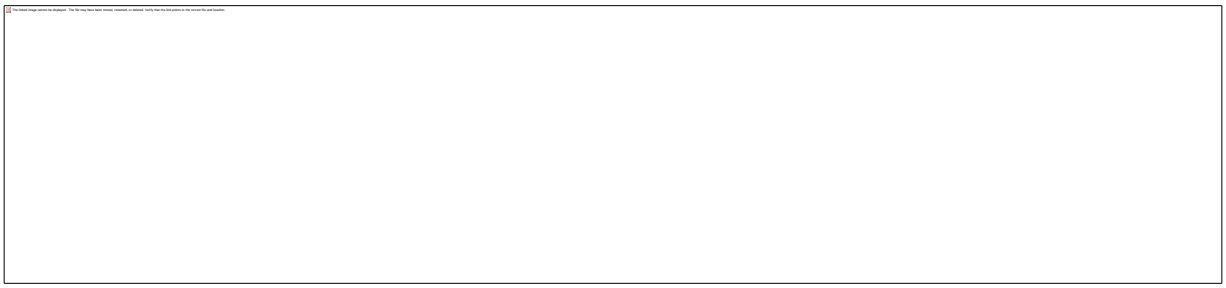
7.16.5.3 Real HelmholtzSolver::lambda () const

References lambda_.

Referenced by PressureSolver::solve().

```
281 {return lambda_;}
```

Here is the caller graph for this function:



7.16.5.4 Real HelmholtzSolver::residual (const ChebyCoeff & *u*, Real *mu*, const ChebyCoeff & *f*, Real *umean*, Real *ua*, Real *ub*) const

References abs(), ChebyCoeff::mean(), and residual().

```
272                                                {
273
274   ChebyCoeff rhs(f);
275   rhs[0] += mu;
276   Real res = residual(u, rhs, ua, ub);
277   res += abs(umean - u.mean());
278   return res;
279 }
```

Here is the call graph for this function:

7.16.5.5 Real HelmholtzSolver::residual (const ChebyCoeff & *u*, const ChebyCoeff & *f*, Real *ua*, Real *ub*) const

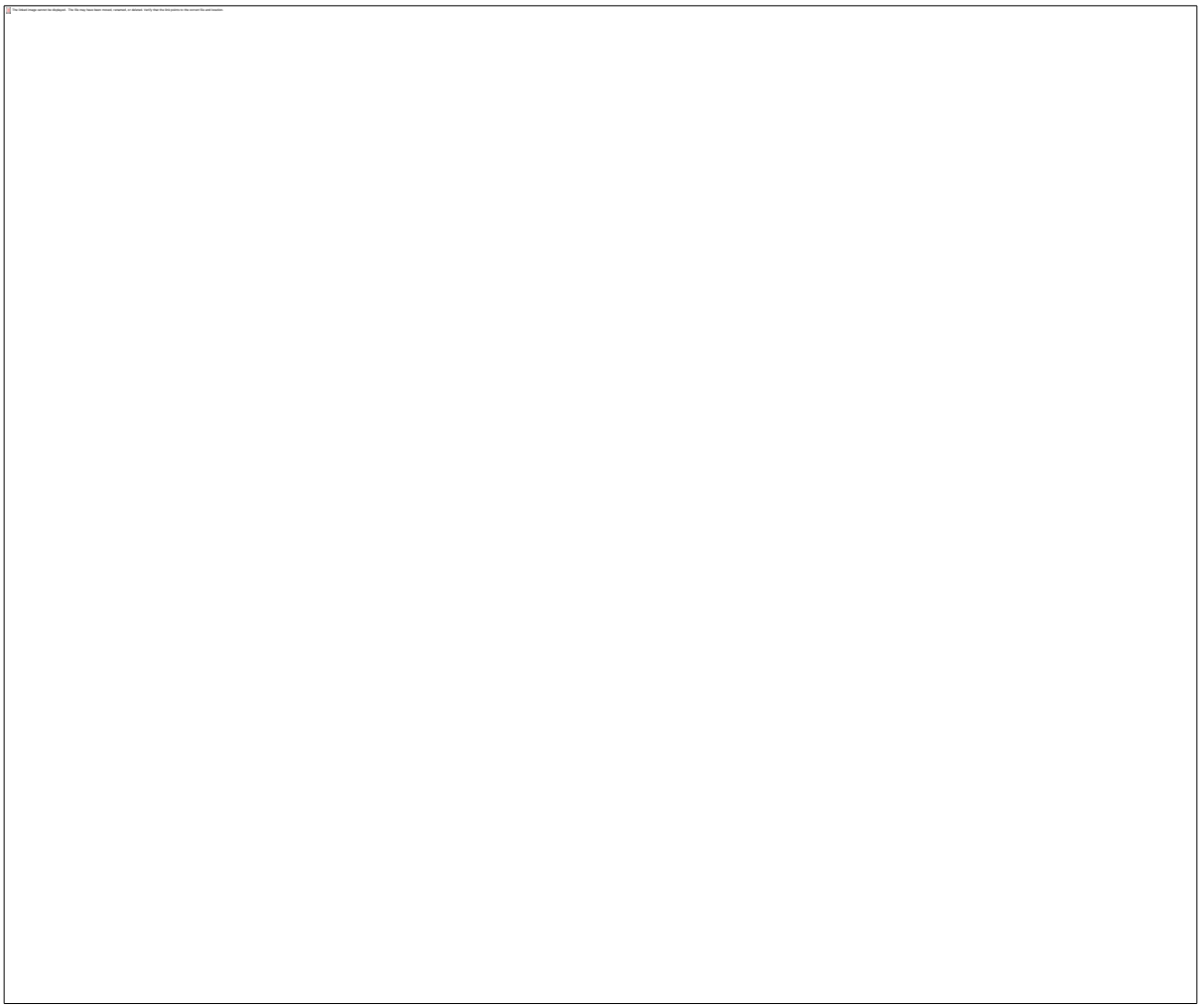
References a_, Ae_, Ao_, b_, Be_, Bo_, diff2(), L1Dist(), lambda_, Vector::length(), BandedTridiag::multiplyStrided(), nModes_, nu_, and Spectral.

Referenced by residual().

```
134                                                     {
135
136   ChebyTransform trans(nModes_);
137   assert(nModes_ == u.length());
138
139   ChebyCoeff u_yy = diff2(u);
140
141   Real errorTau = 0.0;
142   for (int i=0; i<nModes_-2; ++i)
143     errorTau += fabs(nu_*u_yy[i] - lambda_*u[i] - f[i]);
144
145   return errorTau;
146
147   // calculate || A*u - B*f ||
```

```
148   // g = B*f
149   ChebyCoeff g(nModes_, a_, b_, Spectral);
150   Be_.multiplyStrided(f, g, 0, 2);
151   Bo_.multiplyStrided(f, g, 1, 2);
152   g[0] = (ub+ua)/2;
153   g[1] = (ub-ua)/2;
154
155   // h = A*u
156   ChebyCoeff h(nModes_, a_, b_, Spectral);
157   Ae_.multiplyStrided(u, h, 0, 2);
158   Ao_.multiplyStrided(u, h, 1, 2);
159
160   return L1Dist(g,h);
161 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.16.5.6 void HelmholtzSolver::solve (ChebyCoeff & *u*, Real & *mu*, const ChebyCoeff & *f*, Real *umean*, Real *ua*, Real *ub*) const

References a_, b_, ChebyCoeff::mean(), nModes_, nu_, ChebyCoeff::setToZero(), solve(), and Spectral.

```
200                                             {
201
202   // Solve by breaking u into u = uf + um, where
203   // nu uf" - lambda uf = f  and uf(-+1) = ua,ub
204   // nu um" - lambda um = mu and um(-+1) = 0,0
205   /***************
206   ChebyCoeff ua(nModes_);
207   solve(ua, f, a, b);
208   real uamean = ua.mean();
209
210   ChebyCoeff uc(nModes_);
211   ChebyCoeff rhs(nModes_);
212   rhs[0] = nu_;
213   solve(uc, rhs, 0.0, 0.0);  // rhs == nu
214   real ucmean = uc.mean();
215
216   mu = nu_*(umean - uamean)/ucmean;
217   rhs = f;
218   rhs[0] += mu;
219   solve(u, rhs, a, b);       // rhs == f + mu
220   ********************/
221
222   // Solve nu u" - lambda u == f + mu, mean(u) == umean, u(+-1) == a,b.
223   // for v and mu.
224
225   // Solve by breaking u into u = ua + ub, where
226   // eqn1. nu ua" - lambda ua = f  and ua(-+1) = a,b. Solve
227   // eqn2. nu ub" - lambda ub = mu and ub(-+1) = 0,0  mu is O(1)
228
229   // Rescale eqn2, via uc = mu/nu ub. Results in nu on RHS, nu is O(1/Re),
230   // so eqn3 is better conditioned than eqn2.
231   // eqn3. nu uc" - lambda uc = nu and uc(-+1) = 0,0. Solve
232
233   // Then mu = nu*(umean - mean(ua))/mean(uc).
234
235   // Solve eqn1.
236   int N=nModes_;
```

```
237   ChebyCoeff utmp(N, a_, b_, Spectral); // utmp serves as uf
238   ChebyCoeff rtmp(f);
239   solve(utmp, rtmp, ua, ub);
240   Real uamean = utmp.mean();
241   //cout << "eqn1 residual " << residual(utmp, rtmp, a,b) << endl;
242
243   // Solve eqn3.
244   rtmp.setToZero(); // rtmp serves as rhs of eqn3.
245   rtmp[0] = nu_;
246   solve(utmp, rtmp, 0.0, 0.0);  // utmp serves as uc
247   //cout << "eqn3 residual " << residual(utmp,rtmp,0.0,0.0) << endl;
248   Real ucmean = utmp.mean();
249
250   // Solve eqn0 using known value of mu.
251   mu = nu_*(umean - uamean)/ucmean;
252   rtmp = f;
253   rtmp[0] += mu; // rtmp serves as f + mu
254   solve(u, rtmp, ua, ub);
255   //cout << "eqn0 residual " << residual(u,rtmp,a,b) << endl;
256 }
```

Here is the call graph for this function:

7.16.5.7 void HelmholtzSolver::solve (ChebyCoeff & *u*, const ChebyCoeff & *f*, Real *ua*, Real *ub*) const

References Ae_, Ao_, Be_, Bo_, BandedTridiag::multiplyStrided(), ChebyCoeff::setState(), Spectral, ChebyCoeff::state(), and BandedTridiag::ULsolveStrided().

Referenced by PPEDNS::advance(), TauSolver::solve(), PoissonSolver::solve(), solve(), TauSolver::solve_P_and_v(), and TauSolver::TauSolver().

```
116                                                                 {
117   assert(f.state() == Spectral);
118   Be_.multiplyStrided(f, u, 0, 2);
119   Bo_.multiplyStrided(f, u, 1, 2);
120
121   u[0] = (ub+ua)/2;
122   u[1] = (ub-ua)/2;
123
124   // Solve for A u = g
125   Ae_.ULsolveStrided(u, 0, 2);
126   Ao_.ULsolveStrided(u, 1, 2);
127
128   //#ifdef DEBUG
129   //verify(u, f, ua, ub);
130   //#endif
131   u.setState(Spectral);
132 }
```

Here is the call graph for this function:

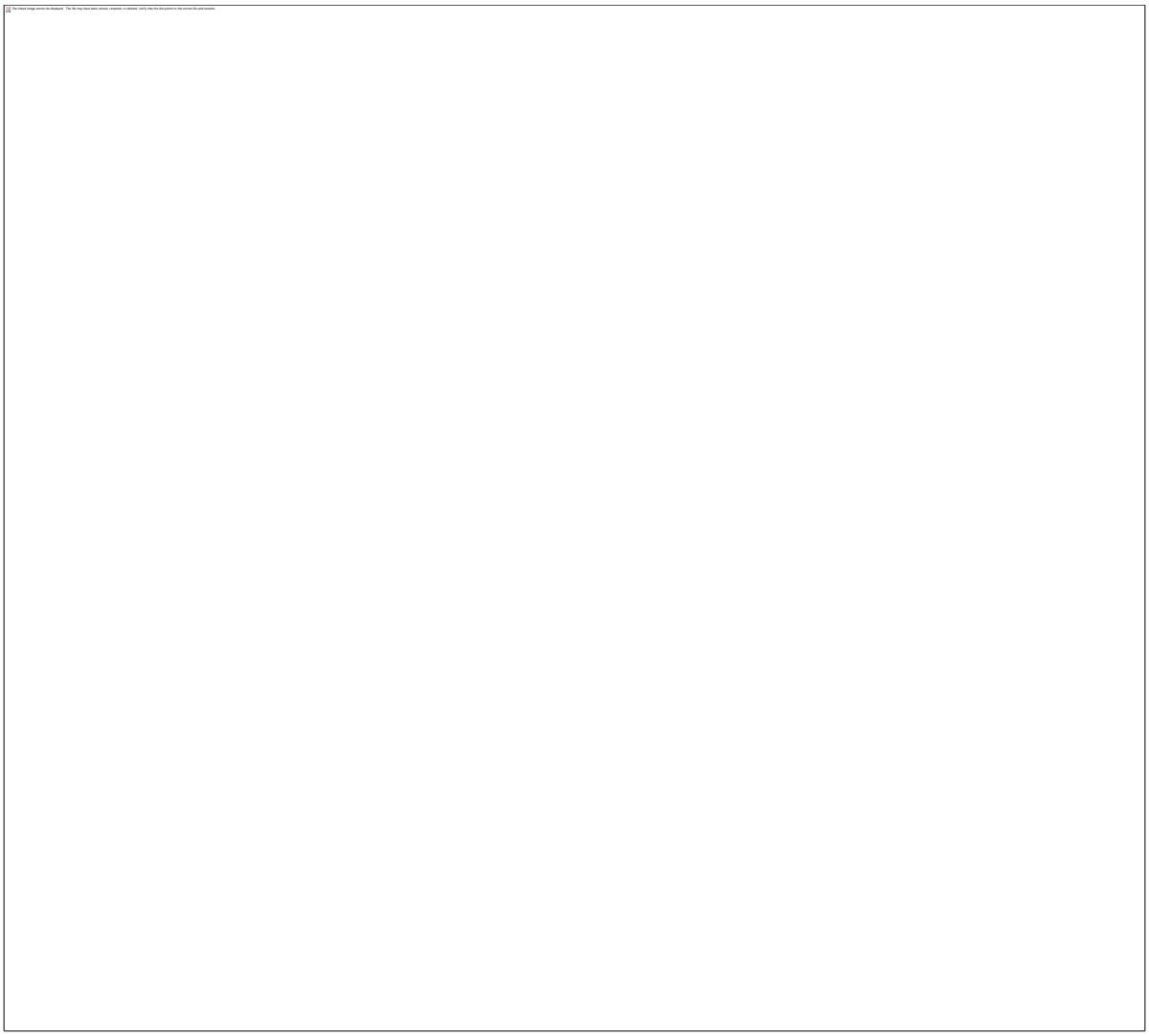
Here is the caller graph for this function:

7.16.5.8 void HelmholtzSolver::verify (ChebyCoeff & *u*, Real & *mu*, const ChebyCoeff & *f*, Real *umean*, Real *ua*, Real *ub*) const

References ChebyCoeff::mean(), and verify().

```
260                                                 {
261
262   cerr << "Helmholtz::verify(u,f,a,b,mu,umean) {" << endl;
263   ChebyCoeff rhs(f);
264   rhs[0] += mu;
265   verify(u,f,ua,ub);
266   cerr << "umean - mean(u) === " << umean - u.mean() << endl;
267   cerr << "} Helmholtz::verify(u,f,ua,ub,mu,umean)" << endl;
268 }
```

Here is the call graph for this function:



7.16.5.9 void HelmholtzSolver::verify (const ChebyCoeff & *u*, const ChebyCoeff & *f*, Real *ua*, Real *ub*, bool *verbose* = false) const

References diff2(), EPSILON, ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), Greater(), L1Norm(), lambda_, Vector::length(), N_, nModes_, norm(), and nu_.

Referenced by verify().

```
164                                                                    {
165
166   ChebyTransform trans(nModes_);
167   assert(nModes_ ==u.length());
168
169   ChebyCoeff u_yy = diff2(u);
170
171   Real errorTau = 0.0;
```

```
172   Real errorL1 = 0.0;
173   int i; // MSVC++ FOR-SCOPE BUG
174   for (i=0; i<nModes_-2; ++i)
175     errorTau += fabs(nu_*u_yy[i] - lambda_*u[i] - f[i]);
176   errorL1 = errorTau;
177   for (i=nModes_-2; i<nModes_; ++i)
178     errorL1 += fabs(nu_*u_yy[i] - lambda_*u[i] - f[i]);
179   Real norm = Greater(L1Norm(u), L1Norm(f));
180   norm = (norm > EPSILON) ? norm : 1.0;
181   Real uas = u.eval_a();
182   Real ubs = u.eval_b();
183
184
185   if (verbose) {
186     cerr << "Helmholtz::verify() { " << endl;
187     cerr << "N nu lambda == " << N_ << ' ' << nu_ << ' ' << lambda_ << endl;
188     cerr << "tauNorm(nu*uyy - lambda*u - f) == " << errorTau << endl;
189     cerr << " L1Norm(nu*uyy - lambda*u - f) == " << errorL1 << endl;
190     cerr << "fabs(uas - ua)      == "<<  fabs(uas - ua) << endl;
191     cerr << "fabs(ubs - ub)      == "<<  fabs(ubs - ub) << endl;
192     cerr << "} Helmholtz::verify()" << endl;
193   }
194   assert(fabs(errorTau/norm) < EPSILON);
195   assert(fabs((uas - ua)/norm) < EPSILON);
196   assert(fabs((ubs - ub)/norm) < EPSILON);
197 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.16.6 Member Data Documentation

7.16.6.1 Real HelmholtzSolver::a_ [private]

Referenced by HelmholtzSolver(), residual(), and solve().

7.16.6.2 BandedTridiag HelmholtzSolver::Ae_ [private]

Referenced by HelmholtzSolver(), residual(), and solve().

7.16.6.3 BandedTridiag HelmholtzSolver::Ao_ [private]

Referenced by HelmholtzSolver(), residual(), and solve().

7.16.6.4 Real HelmholtzSolver::b_ [private]

Referenced by HelmholtzSolver(), residual(), and solve().

7.16.6.5 BandedTridiag HelmholtzSolver::Be_ [private]

Referenced by HelmholtzSolver(), residual(), and solve().

7.16.6.6 BandedTridiag HelmholtzSolver::Bo_ [private]

Referenced by HelmholtzSolver(), residual(), and solve().

7.16.6.7 Real HelmholtzSolver::lambda_ [private]

Referenced by lambda(), residual(), and verify().

7.16.6.8 int HelmholtzSolver::N_ [private]

Referenced by beta(), c(), and verify().

7.16.6.9 int HelmholtzSolver::nEvenModes_ [private]

Referenced by HelmholtzSolver().

7.16.6.10 int HelmholtzSolver::nModes_ [private]

Referenced by HelmholtzSolver(), residual(), solve(), and verify().

7.16.6.11 int HelmholtzSolver::nOddModes_ [private]

Referenced by HelmholtzSolver().

7.16.6.12 Real HelmholtzSolver::nu_ [private]

Referenced by residual(), solve(), and verify().

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

- /_cuda/channelflow-0.9.22/channelflow/helmholtz.h
- /_cuda/channelflow-0.9.22/channelflow/helmholtz.cpp

7.16.6.14

7.17 Limits Class Reference

#include <dns.h>

7.17.1 Public Member Functions

- Limits (int maxI, int maxJ, int maxNx, int maxNy, int maxNz)
- Limits ()
- void operator= (const Limits &lim)
- void setLimits (int maxI, int maxJ, int maxNx, int maxNy, int maxNz)

7.17.2 Public Attributes

- int m_maxI
- int m_maxJ
- int m_maxNx
- int m_maxNy
- int m_maxNz

7.17.3 Constructor & Destructor Documentation

7.17.3.1 Limits::Limits (int *maxI*, int *maxJ*, int *maxNx*, int *maxNy*, int *maxNz*)

References m_maxI, m_maxJ, m_maxNx, m_maxNy, and m_maxNz.

```
351 {
352  m_maxI = maxI;
353  m_maxJ = maxJ;
354  m_maxNx = maxNx;
355  m_maxNy = maxNy;
356  m_maxNz = maxNz;
357
358 }
```

7.17.3.2 Limits::Limits ()

References m_maxI, m_maxJ, m_maxNx, m_maxNy, and m_maxNz.

```
362 {
363   m_maxI=1;
364   m_maxJ=1;
365   m_maxNx=1;
366   m_maxNy=1;
367   m_maxNz=1;
368
```

```
369 }
```

7.17.4 Member Function Documentation

7.17.4.1 void Limits::operator= (const Limits & *lim*)

References m_maxI, m_maxJ, m_maxNx, m_maxNy, and m_maxNz.

```
372 {
373   m_maxI = lim.m_maxI;
374   m_maxJ = lim.m_maxJ;
375   m_maxNx = lim.m_maxNx;
376   m_maxNy = lim.m_maxNy;
377   m_maxNz = lim.m_maxNz;
378 }
```

7.17.4.2 void Limits::setLimits (int *maxI*, int *maxJ*, int *maxNx*, int *maxNy*, int *maxNz*)

References m_maxI, m_maxJ, m_maxNx, m_maxNy, and m_maxNz.

```
381 {
382   m_maxI = maxI;
383   m_maxJ = maxJ;
384   m_maxNx = maxNx;
385   m_maxNy = maxNy;
386   m_maxNz = maxNz;
387 }
```

7.17.5 Member Data Documentation

7.17.5.1 int Limits::m_maxI

Referenced by Limits(), operator=(), and setLimits().

7.17.5.2 int Limits::m_maxJ

Referenced by Limits(), operator=(), and setLimits().

7.17.5.3 int Limits::m_maxNx

Referenced by Limits(), operator=(), and setLimits().

7.17.5.4 int Limits::m_maxNy

Referenced by Limits(), operator=(), and setLimits().

7.17.5.5 int Limits::m_maxNz

Referenced by Limits(), operator=(), and setLimits().

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

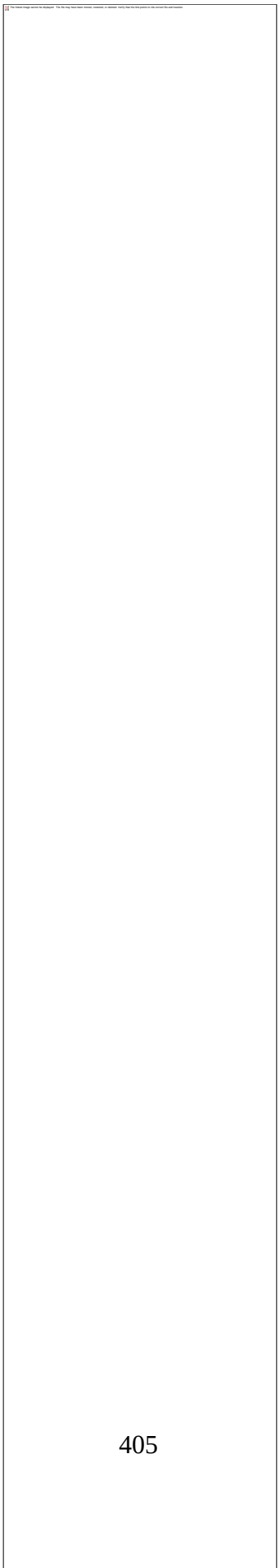
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.17.5.7

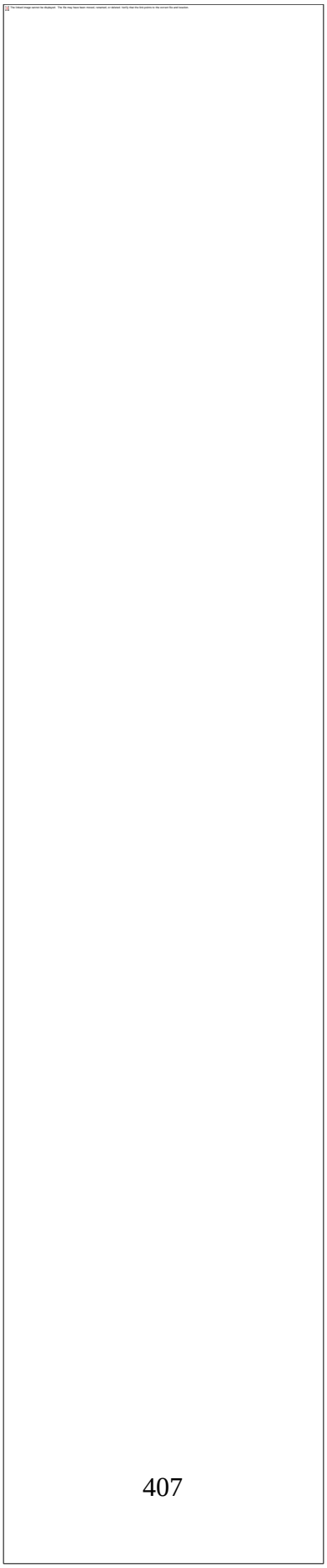
7.18 MultistepDNS Class Reference

#include <dns.h>

Inheritance diagram for MultistepDNS:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

Collaboration diagram for MultistepDNS:


The inked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.18.1 Public Member Functions

- MultistepDNS ()
- MultistepDNS (const MultistepDNS &dns)
- MultistepDNS (FlowField &u, const ChebyCoeff &Ubase, Real nu, Real dt, const DNSFlags &flags, Real t=0)
- ~MultistepDNS ()
- MultistepDNS & operator= (const MultistepDNS &dns)
- virtual void advance (FlowField &u, FlowField &q, int nSteps=1)
- virtual void reset_dt (Real dt)
- virtual bool push (const FlowField &u)
- virtual bool full () const
- virtual DNSAlgorithm * clone () const

7.18.2 Protected Attributes

- Real eta_
- array< Real > alpha_
- array< Real > beta_
- array< FlowField > u_
- array< FlowField > f_
- TauSolver ** tausolver_
- int countdown_

7.18.3 Constructor & Destructor Documentation

7.18.3.1 MultistepDNS::MultistepDNS ()

Referenced by clone().

```
1069   :
1070   DNSAlgorithm()
1071 {}
```

Here is the caller graph for this function:

7.18.3.2 MultistepDNS::MultistepDNS (const MultistepDNS & *dns*)

References DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, and tausolver_.

```
1074   :
1075   DNSAlgorithm(dns),
1076   eta_(dns.eta_),
1077   alpha_(dns.alpha_),
1078   beta_(dns.beta_),
1079   u_(dns.u_),
1080   f_(dns.f_)
1081 {
1082   // Copy tausolvers
1083   tausolver_ = new TauSolver*[Mx_];       // new #1
1084   for (int mx=0; mx<Mx_; ++mx) {
1085     tausolver_[mx] = new TauSolver[Mz_];  // new #2
1086     for (int mz=0; mz<Mz_; ++mz)
1087       tausolver_[mx][mz] = dns.tausolver_[mx][mz];
1088   }
1089 }
```

7.18.3.3 MultistepDNS::MultistepDNS (FlowField & *u*, const ChebyCoeff & *Ubase*, Real *nu*, Real *dt*, const DNSFlags & *flags*, Real *t* = 0)

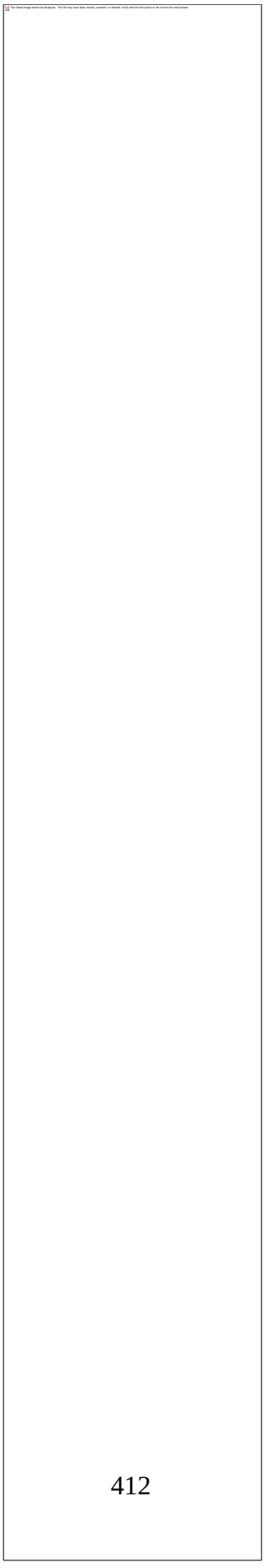
References alpha_, beta_, DNSAlgorithm::cfl_, CNFE1, countdown_, DNSFlags::dealias_xz(), DNSAlgorithm::dt_, eta_, f_, DNSAlgorithm::flags_, DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, navierstokesNL(), DNSAlgorithm::Ninitsteps_, DNSFlags::nonlinearity, DNSAlgorithm::order_, pi, reset_dt(), array< T >::resize(), SBDF1, SBDF2, SBDF3, SBDF4, FlowField::setToZero(), tausolver_, DNSFlags::timestepping, DNSAlgorithm::tmp_, u_, and DNSAlgorithm::Ubase_.

```
1094   :
1095   DNSAlgorithm(u,Ubase,nu,dt,flags,t)
1096 {
1097
1098   TimeStepMethod algorithm = flags.timestepping;
1099   switch (algorithm) {
1100   case CNFE1:
1101   case SBDF1:
1102     order_ = 1;
1103     eta_ = 1.0;
1104     alpha_.resize(order_);
1105     beta_.resize(order_);
1106     alpha_[0] = -1.0;
1107     beta_[0]  =  1.0;
1108     break;
1109   case SBDF2:
1110     order_ = 2;
1111     alpha_.resize(order_);
```

```
1112    beta_.resize(order_);
1113    eta_ = 1.5;
1114    alpha_[0] = -2.0; alpha_[1] =  0.5;
1115    beta_[0]  =  2.0;  beta_[1] = -1.0;
1116    break;
1117   case SBDF3:
1118    order_ = 3;
1119    alpha_.resize(order_);
1120    beta_.resize(order_);
1121    eta_ = 11.0/6.0;
1122    alpha_[0] = -3.0;  alpha_[1] = 1.5; alpha_[2] = -1.0/3.0;
1123    beta_[0]  =  3.0;   beta_[1] = -3.0; beta_[2] = 1.0;
1124    break;
1125   case SBDF4:
1126    order_ = 4;
1127    alpha_.resize(order_);
1128    beta_.resize(order_);
1129    eta_ = 25.0/12.0;
1130    alpha_[0] = -4.0; alpha_[1] =  3.0; alpha_[2] = -4.0/3.0; alpha_[3] = 0.25;
1131    beta_[0]  =  4.0;  beta_[1] = -6.0;  beta_[2] =  4.0;      beta_[3] = -1.0;
1132    break;
1133   default:
1134    cerr << "MultistepDNS::MultistepDNS(un,Ubase,nu,dt,flags,t0)\n"
1135         << "error: flags.timestepping == " << algorithm
1136         << "is a non-multistepping algorithm" << endl;
1137    exit(1);
1138   }
1139
1140   // Configure tausolvers
1141   tausolver_ = new TauSolver*[Mx_];       // new #1
1142   for (int mx=0; mx<Mx_; ++mx)
1143    tausolver_[mx] = new TauSolver[Mz_];  // new #2
1144
1145   reset_dt(dt_);
1146
1147   // Initialize arrays of previous u's and f's
1148   FlowField tmp(u);
1149   tmp.setToZero();
1150
1151   u_.resize(order_);
1152   f_.resize(order_);
1153   for (int j=0; j<order_; ++j) {
1154    u_[j] = tmp;
1155    f_[j] = tmp;
1156   }
1157   if (order_ > 0) { // should always be true
```

```
1158     u_[0] = u;
1159     //cout << "IG: call navierstokesNL 1" <<endl;
1160 
1161     navierstokesNL(u_[0], Ubase_, f_[0], tmp_, flags_.nonlinearity);
1162 
1163     cfl_ = u_[0].CFLfactor(Ubase_);
1164     cfl_ *= flags_.dealias_xz() ? 2.0*pi/3.0*dt_ : pi*dt_;
1165   }
1166 
1167   Ninitsteps_ = order_-1;
1168   countdown_ = Ninitsteps_;
1169 }
```

Here is the call graph for this function:



7.18.3.4 MultistepDNS::~MultistepDNS ()

References DNSAlgorithm::Mx_, and tausolver_.

```
1171                         {
1172   if (tausolver_) {
1173     for (int mx=0; mx<Mx_; ++mx)
1174       delete[] tausolver_[mx];  // undo new #2
1175     delete[] tausolver_;        // undo new #1
1176   }
1177   tausolver_ = 0;
1178 }
```

7.18.4 Member Function Documentation

7.18.4.1 void MultistepDNS::advance (FlowField & *u*, FlowField & *q*, int *nSteps* = 1) [virtual]

Implements DNSAlgorithm.

References DNSAlgorithm::a(), ComplexChebyCoeff::add(), alpha_, DNSAlgorithm::b(), beta_, DNSAlgorithm::cfl_, FlowField::cmplx(), DNSFlags::constraint, DNSFlags::dealias_xz(), DNSAlgorithm::dPdxAct_, DNSAlgorithm::dPdxRef_, DNSAlgorithm::dt_, f_, DNSAlgorithm::flags_, DNSAlgorithm::isAliasedMode(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), Vector::length(), ChebyCoeff::mean(), DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, navierstokesNL(), DNSFlags::nonlinearity, DNSAlgorithm::nu_, DNSAlgorithm::Nx_, DNSAlgorithm::Ny_, DNSAlgorithm::Nyd_, DNSAlgorithm::Nz_, DNSAlgorithm::order_, pi, DNSAlgorithm::Pk_, PressureGradient, PrintAll, PrintTicks, PrintTime, Re(), ComplexChebyCoeff::re, DNSAlgorithm::Rxk_, DNSAlgorithm::Ryk_, DNSAlgorithm::Rzk_, FlowField::setDealiased(), ComplexChebyCoeff::setToZero(), TauSolver::solve(), swap(), DNSAlgorithm::t_, tausolver_, DNSAlgorithm::tmp_, u_, DNSAlgorithm::Ubase_, DNSAlgorithm::Ubaseyy_, DNSAlgorithm::UbulkAct_, DNSAlgorithm::UbulkBase_, DNSAlgorithm::UbulkRef_, DNSAlgorithm::uk_, DNSFlags::verbosity, DNSAlgorithm::vk_, and DNSAlgorithm::wk_.

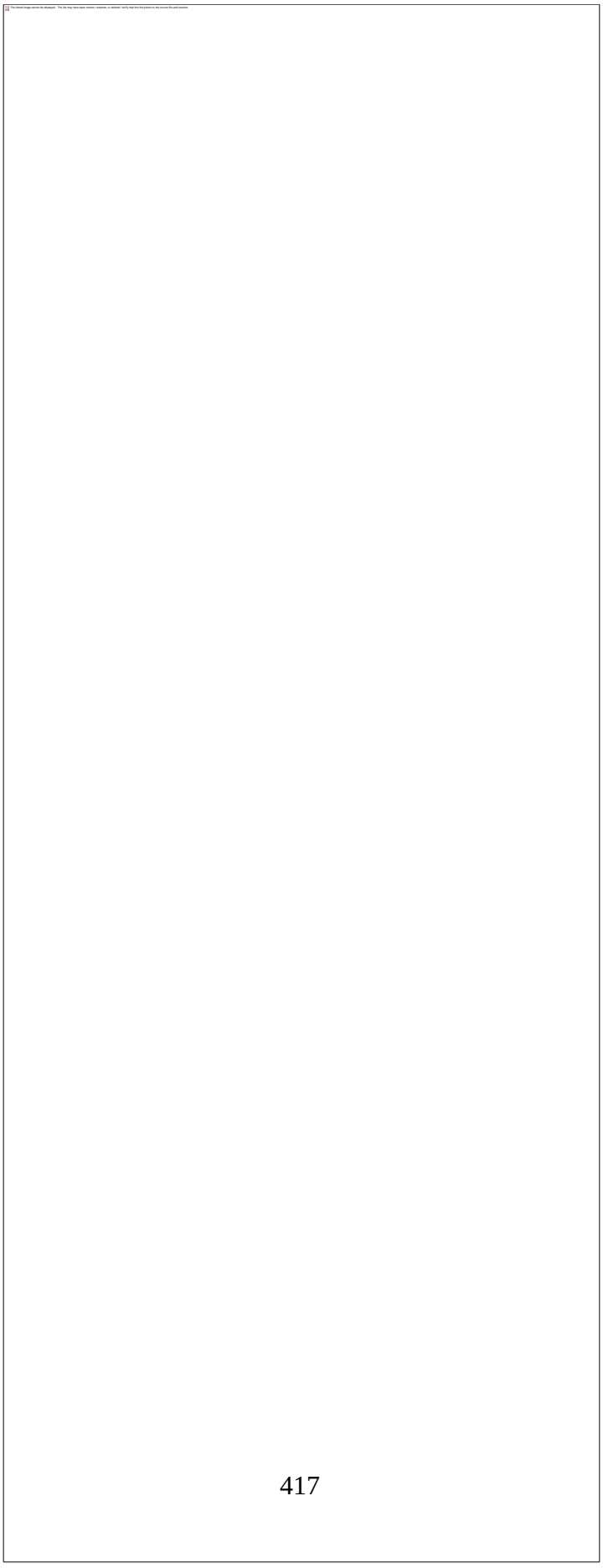
```
1214                                                 {
1215
1216   const int kxmax = un.kxmax();
1217   const int kzmax = un.kzmax();
1218   for (int step=0; step<Nsteps; ++step) {
1219
1220     //cout << "MultistepDNS::advance(u,q,N) stack : " << endl;
1221     // Update each Fourier mode with time-stepping algorithm
```

```
1222    for (int mx=0; mx<Mx_; ++mx) {
1223      const int kx = un.kx(mx);
1224
1225      for (int mz=0; mz<Mz_; ++mz) {
1226        const int kz = un.kz(mz);
1227
1228        // Zero out the aliased modes and break to next kx,kz
1229        if ((kx == kxmax || kz == kzmax) ||
1230            flags_.dealias_xz() && isAliasedMode(kx,kz)) {
1231          for (int ny=0; ny<Nyd_; ++ny) {
1232            u_[0].cmplx(mx,ny,mz,0) = 0.0;
1233            u_[0].cmplx(mx,ny,mz,1) = 0.0;
1234            u_[0].cmplx(mx,ny,mz,2) = 0.0;
1235            qn.cmplx(mx,ny,mz,0) = 0.0;
1236          }
1237          break;
1238        }
1239
1240        // For nonaliased modes
1241        Rxk_.setToZero();
1242        Ryk_.setToZero();
1243        Rzk_.setToZero();
1244
1245        for (int j=0; j<order_; ++j) {
1246          const Real a = -alpha_[j]/dt_;
1247          const Real b = -beta_[j];
1248          for (int ny=0; ny<Nyd_; ++ny) {
1249            Rxk_.add(ny, a*u_[j].cmplx(mx,ny,mz,0)+b*f_[j].cmplx(mx,ny,mz,0));
1250            Ryk_.add(ny, a*u_[j].cmplx(mx,ny,mz,1)+b*f_[j].cmplx(mx,ny,mz,1));
1251            Rzk_.add(ny, a*u_[j].cmplx(mx,ny,mz,2)+b*f_[j].cmplx(mx,ny,mz,2));
1252          }
1253        }
1254
1255        // Solve the tau solutions
1256        if (kx!=0 || kz!=0)
1257          tausolver_[mx][mz].solve(uk_,vk_,wk_,Pk_, Rxk_,Ryk_,Rzk_);
1258
1259        else { // kx,kz == 0,0
1260
1261          if (Ubaseyy_.length() > 0)
1262            for (int ny=0; ny<Ny_; ++ny)
1263              Rxk_.re[ny] += nu_*Ubaseyy_[ny]; // Rx has additional term
1264
1265          if (flags_.constraint == PressureGradient) {
1266            // pressure is supplied, put on RHS of tau eqn
1267            Rxk_.re[0] -= dPdxRef_;
```

```
1268
1269          // Solve the tau equations
1270          tausolver_[mx][mz].solve(uk_, vk_, wk_, Pk_, Rxk_,Ryk_,Rzk_);
1271
1272          // Bulk vel is free variable determined from soln of tau eqn
1273          UbulkAct_ = UbulkBase_ + uk_.re.mean();
1274          dPdxAct_ = dPdxRef_;
1275        }
1276        else { // const bulk velocity
1277          // bulk velocity is supplied, put on RHS of tau eqn
1278          // dPdxAct (i.e. at next time step is solved for)
1279          // constraint:    UbulkBase + mean(u) = UbulkRef.
1280          tausolver_[mx][mz].solve(uk_, vk_, wk_, Pk_, dPdxAct_,
1281                                   Rxk_, Ryk_, Rzk_,
1282                                   UbulkRef_ - UbulkBase_);
1283          UbulkAct_ = UbulkBase_ + uk_.re.mean(); // should == UbulkRef_
1284        }
1285      }
1286
1287      // Load solutions back into the external 3d data arrays.
1288      // Because of FFTW complex symmetries
1289      // The 0,0 mode must be real.
1290      // For Nx even, the kxmax,0 mode must be real
1291      // For Nz even, the 0,kzmax mode must be real
1292      // For Nx,Nz even, the kxmax,kzmax mode must be real
1293      if ((kx == 0 && kz == 0) ||
1294          (Nx_%2 == 0 && kx == kxmax && kz == 0) ||
1295          (Nz_%2 == 0 && kz == kzmax && kx == 0) ||
1296          (Nx_%2 == 0 && Nz_%2 == 0 && kx == kxmax && kz == kzmax)) {
1297
1298        for (int ny=0; ny<Nyd_; ++ny) {
1299          un.cmplx(mx,ny,mz,0) = Complex(Re(uk_[ny]), 0.0);
1300          un.cmplx(mx,ny,mz,1) = Complex(Re(vk_[ny]), 0.0);
1301          un.cmplx(mx,ny,mz,2) = Complex(Re(wk_[ny]), 0.0);
1302          qn.cmplx(mx,ny,mz,0) = Complex(Re(Pk_[ny]), 0.0);
1303        }
1304      }
1305      // The normal case, for general kx,kz
1306      else
1307        for (int ny=0; ny<Nyd_; ++ny) {
1308          un.cmplx(mx,ny,mz,0) = uk_[ny];
1309          un.cmplx(mx,ny,mz,1) = vk_[ny];
1310          un.cmplx(mx,ny,mz,2) = wk_[ny];
1311          qn.cmplx(mx,ny,mz,0) = Pk_[ny];
1312        }
1313
```

```
1314        // And now set the y-aliased modes to zero.
1315        for (int ny=Nyd_; ny<Ny_; ++ny) {
1316          un.cmplx(mx,ny,mz,0) = 0.0;
1317          un.cmplx(mx,ny,mz,1) = 0.0;
1318          un.cmplx(mx,ny,mz,2) = 0.0;
1319          qn.cmplx(mx,ny,mz,0) = 0.0;
1320        }
1321      }
1322    }
1323
1324    // The solution is stored in un. Shift entire u and f arrays in time
1325    // and push un into u_[0]. Ie shift u_[K] <- u_[K-1] <- ... <- u_[0] <- un
1326    for (int j=order_-1; j>0; --j) {
1327      swap(f_[j], f_[j-1]);
1328      swap(u_[j], u_[j-1]);
1329    }
1330    if (order_ > 0) {
1331      u_[0] = un;
1332    //cout << "IG: call navierstokesNL 2" <<endl;
1333
1334      navierstokesNL(u_[0], Ubase_, f_[0], tmp_, flags_.nonlinearity);
1335    }
1336    t_ += dt_;
1337
1338    if (flags_.verbosity == PrintTime || flags_.verbosity == PrintAll)
1339      cout << t_ << ' ' << flush;
1340    else if (flags_.verbosity == PrintTicks)
1341      cout << '.' << flush;
1342  }
1343
1344  cfl_ = u_[0].CFLfactor(Ubase_);
1345  cfl_ *= flags_.dealias_xz() ? 2.0*pi/3.0*dt_ : pi*dt_;
1346
1347  // If using dealiasing, set flag in FlowField that compactifies binary IO
1348  un.setDealiased(flags_.dealias_xz());
1349  qn.setDealiased(flags_.dealias_xz());
1350
1351  if (flags_.verbosity == PrintTime || flags_.verbosity == PrintAll)
1352    cout << endl;
1353
1354  return;
1355 }
```

Here is the call graph for this function:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.18.4.2 DNSAlgorithm * MultistepDNS::clone () const [virtual]

Implements DNSAlgorithm.

References MultistepDNS().

```
1180                                    {
1181   return new MultistepDNS(*this);
1182 }
```

Here is the call graph for this function:

7.18.4.3 bool MultistepDNS::full () const [virtual]

Reimplemented from DNSAlgorithm.

References countdown_.

Referenced by push().

```
1397                                    {
1398   return (countdown_ == 0) ? true : false;
1399 }
```

Here is the caller graph for this function:

7.18.4.4 MultistepDNS& MultistepDNS::operator= (const MultistepDNS & *dns*)

Reimplemented from DNSAlgorithm.

7.18.4.5 bool MultistepDNS::push (const FlowField & *u*) [virtual]

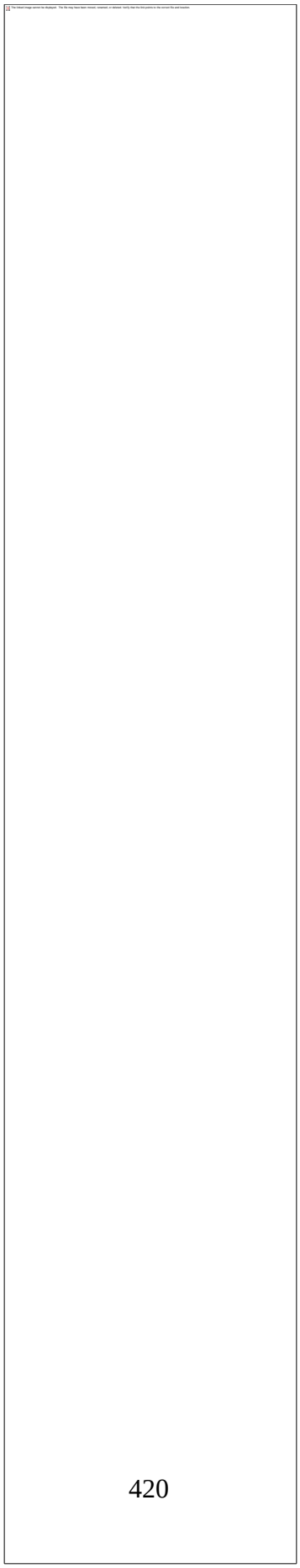
Reimplemented from DNSAlgorithm.

References DNSAlgorithm::cfl_, countdown_, DNSFlags::dealias_xz(), DNSAlgorithm::dt_, f_, DNSAlgorithm::flags_, full(), navierstokesNL(), DNSFlags::nonlinearity, DNSAlgorithm::order_, pi, swap(), DNSAlgorithm::t_, DNSAlgorithm::tmp_, u_, and DNSAlgorithm::Ubase_.

```
1373                                    {
1374
```

```
1375   // Let K = order-1. Arrays are then u_[0:K], f_[0:K]
1376   // Shift u_[K] <- u_[K-1] <- ... <- u_[0] <- un
1377   for (int j=order_-1; j>0; --j) {
1378     swap(u_[j], u_[j-1]);
1379     swap(f_[j], f_[j-1]);
1380   }
1381
1382   if (order_ > 0) {
1383     u_[0] = un;
1384     //cout << "IG call navierstokesNL 3" <<endl;
1385
1386     navierstokesNL(u_[0], Ubase_, f_[0], tmp_, flags_.nonlinearity);
1387     cfl_ = u_[0].CFLfactor(Ubase_);
1388     cfl_ *= flags_.dealias_xz() ? 2.0*pi/3.0*dt_ : pi*dt_;
1389   }
1390
1391   t_ += dt_;
1392   --countdown_;
1393
1394   return full();
1395 }
```

Here is the call graph for this function:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.18.4.6 void MultistepDNS::reset_dt (Real *dt*) [virtual]

Implements DNSAlgorithm.

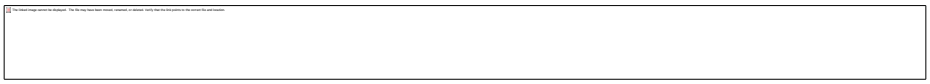
References DNSAlgorithm::a_, DNSAlgorithm::b_, DNSAlgorithm::cfl_, countdown_, DNSFlags::dealias_xz(), DNSAlgorithm::dt_, eta_, DNSAlgorithm::flags_, DNSAlgorithm::isAliasedMode(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), DNSAlgorithm::Lx_, DNSAlgorithm::Lz_, DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, DNSAlgorithm::Ninitsteps_, DNSAlgorithm::nu_, DNSAlgorithm::Nyd_, pi, square(), DNSFlags::taucorrection, tausolver_, and DNSAlgorithm::tmp_.

Referenced by MultistepDNS().

```
1184                                {
1185   cfl_ *= dt/dt_;
1186   //nu_ = nu;
1187   dt_ = dt;
1188   const Real c = 4.0*square(pi)*nu_;
1189   const int kxmax = tmp_.kxmax();
1190   const int kzmax = tmp_.kzmax();
1191 
1192   // This loop replaces the TauSolver objects at tausolver_[substep][mx][mz]
1193   // with new TauSolver objects, with the given parameters.
1194   for (int mx=0; mx<Mx_; ++mx) {
1195     int kx = tmp_.kx(mx);
1196     for (int mz=0; mz<Mz_; ++mz) {
1197       int kz = tmp_.kz(mz);
1198       Real lambda = eta_/dt_ + c*(square(kx/Lx_) + square(kz/Lz_));
1199 
1200       // When using dealiasing, some modes get set to zero, rather than
1201       // updated with momentum eqns. Don't initialize TauSolvers for these.
1202       if ((kx != kxmax && kz != kzmax) &&
1203           (!flags_.dealias_xz() || !isAliasedMode(kx,kz)))
1204 
1205         tausolver_[mx][mz] = TauSolver(kx, kz, Lx_, Lz_, a_, b_, lambda,
1206                                        nu_, Nyd_, flags_.taucorrection);
1207     }
1208   }
1209   // Start from beginning on initialization
1210   countdown_ = Ninitsteps_;
1211 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



7.18.5 Member Data Documentation

7.18.5.1 array<Real> MultistepDNS::alpha_ [protected]

Referenced by advance(), and MultistepDNS().

7.18.5.2 array<Real> MultistepDNS::beta_ [protected]

Referenced by advance(), and MultistepDNS().

7.18.5.3 int MultistepDNS::countdown_ [protected]

Referenced by full(), MultistepDNS(), push(), and reset_dt().

7.18.5.4 Real MultistepDNS::eta_ [protected]

Referenced by MultistepDNS(), and reset_dt().

7.18.5.5 array<FlowField> MultistepDNS::f_ [protected]

Referenced by advance(), MultistepDNS(), and push().

7.18.5.6 TauSolver** MultistepDNS::tausolver_ [protected]

Referenced by advance(), MultistepDNS(), reset_dt(), and ~MultistepDNS().

7.18.5.7 array<FlowField> MultistepDNS::u_ [protected]

Referenced by advance(), MultistepDNS(), and push().

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

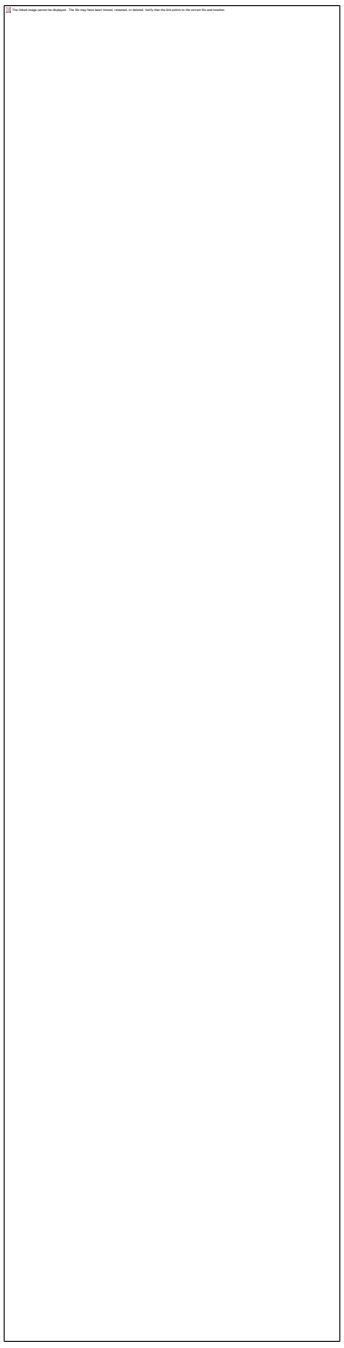
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.18.5.9

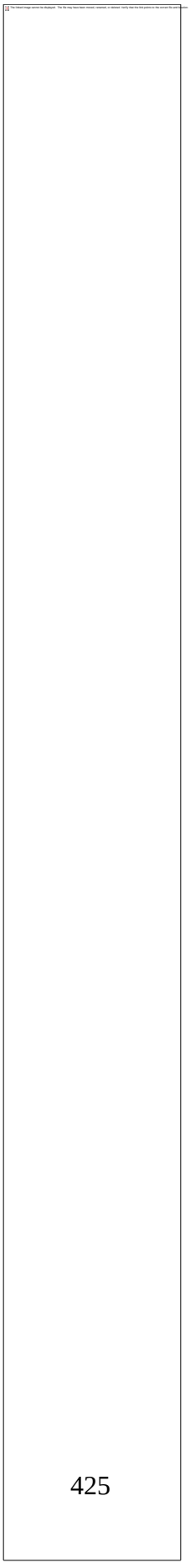
7.19 PoissonSolver Class Reference

#include <poissonsolver.h>

Inheritance diagram for PoissonSolver:



Collaboration diagram for PoissonSolver:



7.19.1 Public Member Functions

- PoissonSolver ()
- PoissonSolver (const FlowField &u)
- PoissonSolver (int Nx, int Ny, int Nz, int Nd, Real Lx, Real Lz, Real a, Real b)
- PoissonSolver (const PoissonSolver &ps)
- PoissonSolver & operator= (const PoissonSolver &ps)
- ~PoissonSolver ()
- void solve (FlowField &u, const FlowField &f) const
- void solve (FlowField &u, const FlowField &f, const FlowField &bc) const
- Real verify (const FlowField &u, const FlowField &f) const
- Real verify (const FlowField &u, const FlowField &f, const FlowField &bc) const
- bool geomCongruent (const FlowField &u) const
- bool congruent (const FlowField &u) const

7.19.2 Protected Attributes

- int Mx_
- int My_
- int Mz_
- int Nd_
- Real Lx_
- Real Lz_
- Real a_
- Real b_
- HelmholtzSolver ** helmholtz_

7.19.3 Constructor & Destructor Documentation

7.19.3.1 PoissonSolver::PoissonSolver ()

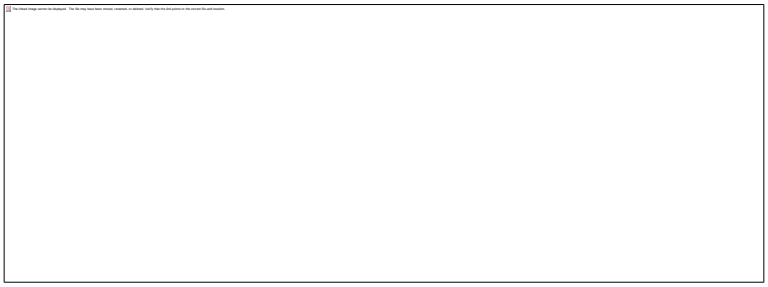
```
34   :
35   Mx_(0),
36   My_(0),
37   Mz_(0),
38   Nd_(0),
39   Lx_(0),
40   Lz_(0),
41   a_(0),
42   b_(0),
43   helmholtz_(0)
44 {}
```

7.19.3.2 PoissonSolver::PoissonSolver (const FlowField & *u*)

References a_, b_, helmholtz_, FlowField::kx(), FlowField::kz(), Lx_, Lz_, Mx_, My_, Mz_, pi, and square().

```
101   :
102   Mx_(u.Mx()),
103   My_(u.My()),
104   Mz_(u.Mz()),
105   Nd_(u.Nd()),
106   Lx_(u.Lx()),
107   Lz_(u.Lz()),
108   a_(u.a()),
109   b_(u.b()),
110   helmholtz_(0)
111 {
112
113   // Allocate Mx x Mz array of HelmholtzSolvers
114   helmholtz_ = new HelmholtzSolver*[Mx_]; // new #1
115   for (int mx=0; mx<Mx_; ++mx)
116     helmholtz_[mx] = new HelmholtzSolver[Mz_];  // new #2
117
118   // Assign Helmholtz solvers into array. Each solves p" - lambda p = f
119   for (int mx=0; mx<Mx_; ++mx) {
120     int kx = u.kx(mx);
121     for (int mz=0; mz<Mz_; ++mz) {
122       int kz = u.kz(mz);
123       Real lambda = 4.0*pi*pi*(square(kx/Lx_) + square(kz/Lz_));
124       helmholtz_[mx][mz] = HelmholtzSolver(My_, a_, b_, lambda);
125     }
126   }
127 }
```

Here is the call graph for this function:



7.19.3.3 PoissonSolver::PoissonSolver (int *Nx*, int *Ny*, int *Nz*, int *Nd*, Real *Lx*, Real *Lz*, Real *a*, Real *b*)

References a_, b_, helmholtz_, Lx_, Lz_, Mx_, My_, Mz_, pi, and square().

```
49   :
50   Mx_(Nx),
51   My_(Ny),
52   Mz_(Nz/2+1),
53   Nd_(Nd),
54   Lx_(Lx),
55   Lz_(Lz),
56   a_(a),
57   b_(b),
58   helmholtz_(0)
59 {
60   // Allocate Mx x Mz array of HelmholtzSolvers
61   helmholtz_ = new HelmholtzSolver*[Mx_]; // new #1
62   for (int mx=0; mx<Mx_; ++mx)
63     helmholtz_[mx] = new HelmholtzSolver[Mz_];  // new #2
64
65   // Assign Helmholtz solvers into array. Each solves u" - lambda u = f
66   for (int mx=0; mx<Mx_; ++mx) {
67     int kx = (mx <= Nx/2) ? mx : mx - Nx; // same as FlowField::kx(mx)
68     for (int mz=0; mz<Mz_; ++mz) {
69       int kz = mz; // same as FlowField::kz(mz)
70       Real lambda = 4.0*pi*pi*(square(kx/Lx_) + square(kz/Lz_));
71       helmholtz_[mx][mz] = HelmholtzSolver(My_, a_, b_, lambda);
72     }
73   }
74 }
```

Here is the call graph for this function:

7.19.3.4 PoissonSolver::PoissonSolver (const PoissonSolver & *ps*)

References helmholtz_, Mx_, and Mz_.

```
77   :
78   Mx_(ps.Mx_),
79   My_(ps.My_),
80   Mz_(ps.Mz_),
```

```
81   Nd_(ps.Nd_),
82   Lx_(ps.Lx_),
83   Lz_(ps.Lz_),
84   a_(ps.a_),
85   b_(ps.b_),
86   helmholtz_(0)
87 {
88
89   // Allocate Mx x Mz array of HelmholtzSolvers
90   helmholtz_ = new HelmholtzSolver*[Mx_]; // new #1
91   for (int mx=0; mx<Mx_; ++mx)
92     helmholtz_[mx] = new HelmholtzSolver[Mz_];  // new #2
93
94   // Assign Helmholtz solvers into array. Each solves p" - lambda p = f
95   for (int mx=0; mx<Mx_; ++mx)
96     for (int mz=0; mz<Mz_; ++mz)
97       helmholtz_[mx][mz] = ps.helmholtz_[mx][mz];
98 }
```

7.19.3.5 PoissonSolver::~PoissonSolver ()

References helmholtz_, and Mx_.

```
161                              {
162   for (int mx=0; mx<Mx_; ++mx)
163     delete[] helmholtz_[mx]; // undo new #2
164   delete[] helmholtz_;       // undo new #1
165 }
```

7.19.4 Member Function Documentation

7.19.4.1 bool PoissonSolver::congruent (const FlowField & *u*) const

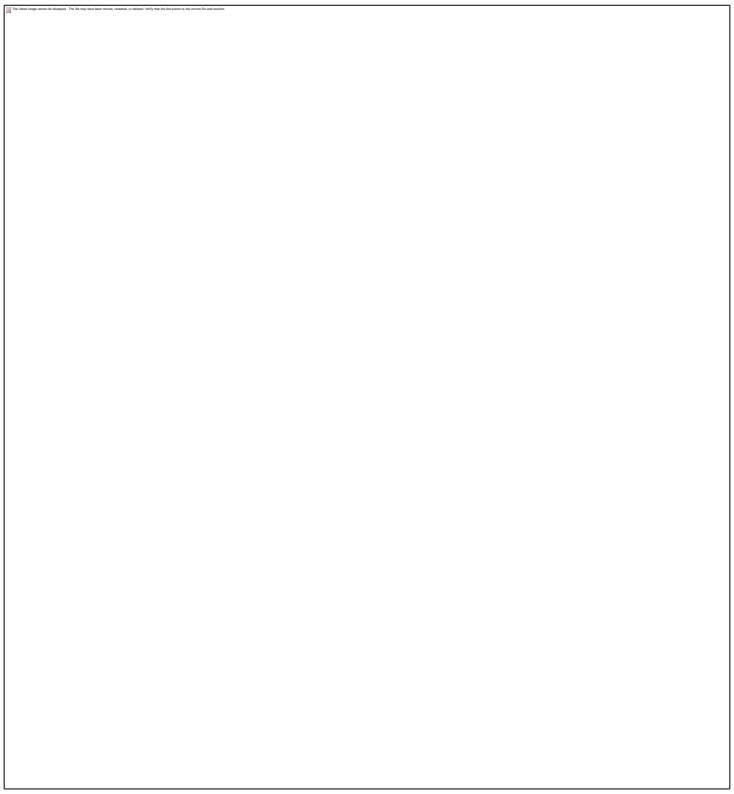
References FlowField::a(), a_, FlowField::b(), b_, FlowField::Lx(), Lx_, FlowField::Lz(), Lz_, FlowField::Mx(), Mx_, FlowField::My(), My_, FlowField::Mz(), Mz_, FlowField::Nd(), and Nd_.

Referenced by PressureSolver::solve(), solve(), PressureSolver::verify(), and verify().

```
173                                          {
174   return
175     (u.Mx() == Mx_) && (u.My() == My_) && (u.Mz()== Mz_) && (u.Nd()== Nd_)
&&
176     (u.Lx() == Lx_) && (u.Lz() == Lz_) && (u.a() == a_)  && (u.b() == b_);
```

```
177     (u.Lx() == Lx_) && (u.Lz() == Lz_) && (u.a() == a_) && (u.b() == b_);
178 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.19.4.2 bool PoissonSolver::geomCongruent (const FlowField & *u*) const

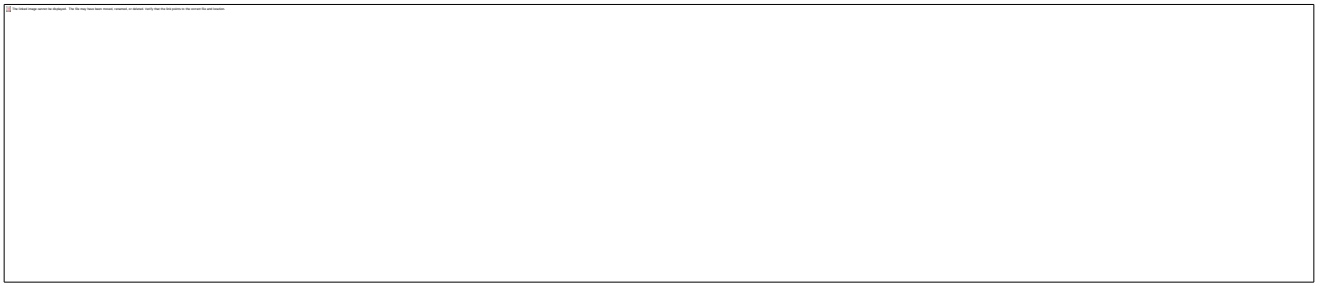
References FlowField::a(), a_, FlowField::b(), b_, FlowField::Lx(), Lx_, FlowField::Lz(), Lz_, FlowField::Mx(), Mx_, FlowField::My(), My_, FlowField::Mz(), and Mz_.

Referenced by PressureSolver::solve(), and PressureSolver::verify().

```
167                                         {
168   return
169     (u.Mx() == Mx_) && (u.My() == My_) && (u.Mz()== Mz_) &&
170     (u.Lx() == Lx_) && (u.Lz() == Lz_) && (u.a() == a_)  && (u.b() == b_);
171 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



7.19.4.3 PoissonSolver & PoissonSolver::operator= (const PoissonSolver & *ps*)

References a_, b_, helmholtz_, Lx_, Lz_, Mx_, My_, Mz_, and Nd_.

```
129                                             {
130
131   if (this == &ps)
132     return *this;
133
134   // Delete old helmholtz solvers
135   for (int mx=0; mx<Mx_; ++mx)
136     delete[] helmholtz_[mx]; // undo new #2
137   delete[] helmholtz_;       // undo new #1
138
139   Mx_ = ps.Mx_;
140   My_ = ps.My_;
141   Mz_ = ps.Mz_;
142   Nd_ = ps.Nd_;
143   Lx_ = ps.Lx_;
144   Lz_ = ps.Lz_;
145   a_ = ps.a_;
146   b_ = ps.b_;
147
148   // Allocate Mx x Mz array of HelmholtzSolvers
149   helmholtz_ = new HelmholtzSolver*[Mx_]; // new #1
150   for (int mx=0; mx<Mx_; ++mx)
151     helmholtz_[mx] = new HelmholtzSolver[Mz_];  // new #2
152
153   // Assign Helmholtz solvers into array. Each solves p" - lambda p = f
154   for (int mx=0; mx<Mx_; ++mx)
155     for (int mz=0; mz<Mz_; ++mz)
156       helmholtz_[mx][mz] = ps.helmholtz_[mx][mz];
157
158   return *this;
159 }
```

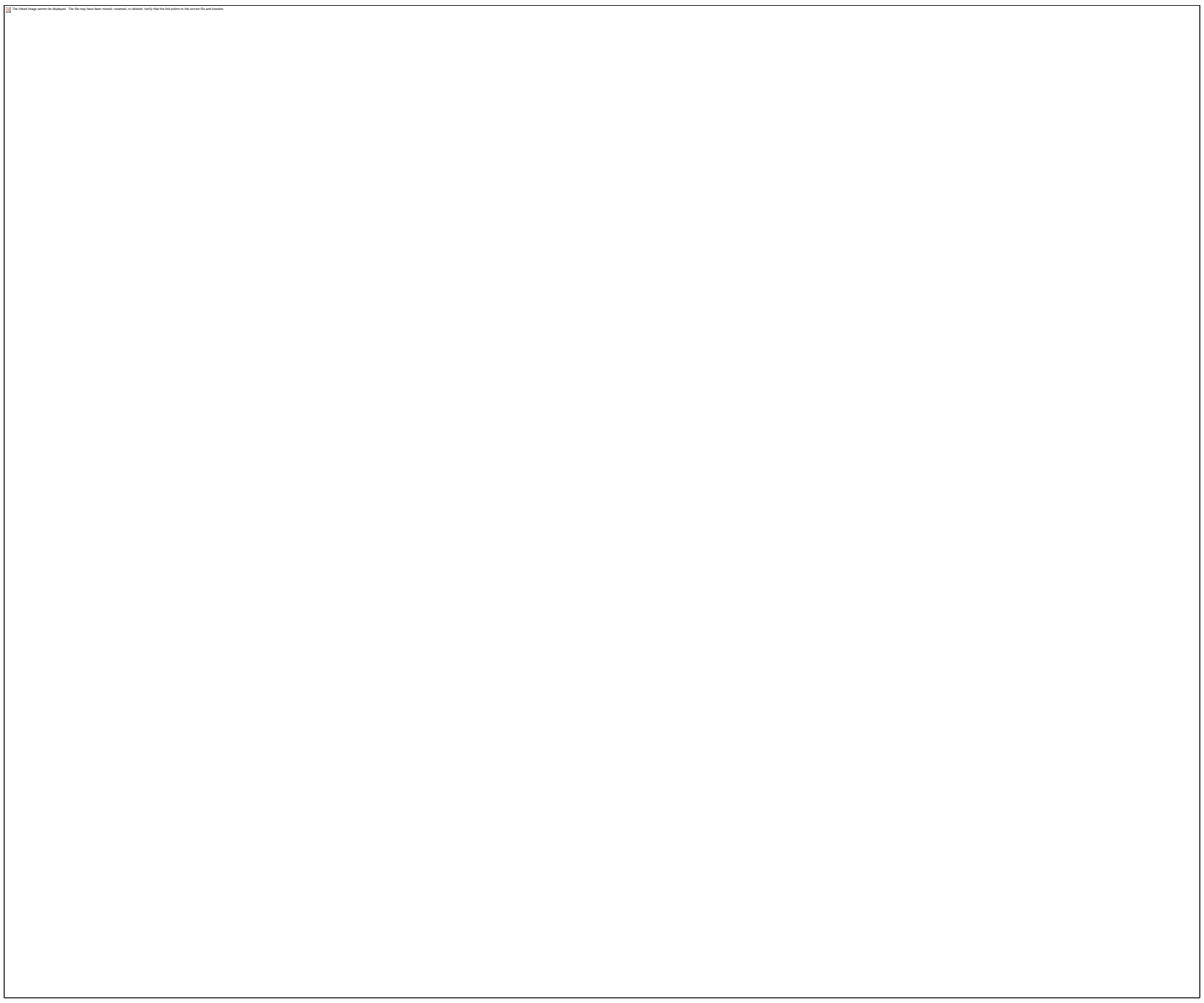
7.19.4.4 void PoissonSolver::solve (FlowField & *u*, const FlowField & *f*, const FlowField & *bc*) const

References a_, FlowField::assertState(), b_, FlowField::cmplx(), congruent(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), helmholtz_, ComplexChebyCoeff::im, Mx_, My_, Mz_, Nd_, ComplexChebyCoeff::re, ComplexChebyCoeff::set(), FlowField::setState(), FlowField::setToZero(), HelmholtzSolver::solve(), Spectral, and FlowField::xzstate().

```
203                                             {
204   assert(this->congruent(f));
205   assert(this->congruent(u));
206   assert(this->congruent(bc));
207   assert(bc.xzstate() == Spectral);
208   f.assertState(Spectral, Spectral);
```

```
209   u.setToZero();
210   u.setState(Spectral, Spectral);
211
212   ComplexChebyCoeff fk(My_, a_, b_, Spectral);
213   ComplexChebyCoeff bk(My_, a_, b_, Spectral);
214   ComplexChebyCoeff uk(My_, a_, b_, Spectral);
215
216   for (int i=0; i<Nd_; ++i)
217     for (int mx=0; mx<Mx_; ++mx)
218       for (int mz=0; mz<Mz_; ++mz) {
219         for (int my=0; my<My_; ++my) {
220           fk.set(my, f.cmplx(mx,my,mz,i));
221           bk.set(my, bc.cmplx(mx,my,mz,i));
222         }
223         helmholtz_[mx][mz].solve(uk.re, fk.re, bk.re.eval_a(), bk.re.eval_b());
224         helmholtz_[mx][mz].solve(uk.im, fk.im, bk.im.eval_a(), bk.im.eval_b());
225         for (int my=0; my<My_; ++my)
226           u.cmplx(mx,my,mz,i) = uk[my];
227       }
228 }
```

Here is the call graph for this function:



7.19.4.5 void PoissonSolver::solve (FlowField & *u*, const FlowField & *f*) const

References a_, FlowField::assertState(), b_, FlowField::cmplx(), congruent(), helmholtz_, ComplexChebyCoeff::im, Mx_, My_, Mz_, Nd_, ComplexChebyCoeff::re, ComplexChebyCoeff::set(), FlowField::setState(), FlowField::setToZero(), HelmholtzSolver::solve(), and Spectral.

```
180                                                 {
181   assert(this->congruent(u));
182   assert(this->congruent(f));
183   f.assertState(Spectral, Spectral);
184   u.setToZero();
185   u.setState(Spectral, Spectral);
186
187   ComplexChebyCoeff fk(My_, a_, b_, Spectral);
188   ComplexChebyCoeff uk(My_, a_, b_, Spectral);
189
```

```
190   Real zero = 0.0; // dirichlet BC
191   for (int i=0; i<Nd_; ++i)
192     for (int mx=0; mx<Mx_; ++mx)
193       for (int mz=0; mz<Mz_; ++mz) {
194         for (int my=0; my<My_; ++my)
195           fk.set(my, f.cmplx(mx,my,mz,i));
196         helmholtz_[mx][mz].solve(uk.re, fk.re, zero, zero);
197         helmholtz_[mx][mz].solve(uk.im, fk.im, zero, zero);
198         for (int my=0; my<My_; ++my)
199           u.cmplx(mx,my,mz,i) = uk[my];
200       }
201 }
```

Here is the call graph for this function:

7.19.4.6 Real PoissonSolver::verify (const FlowField & *u*, const FlowField & *f*, const FlowField & *bc*) const

References FlowField::assertState(), bcDist(), bcNorm(), congruent(), L2Dist(), L2Norm(), lapl(), and Spectral.

```
270                                                     {
271   assert(this->congruent(u));
272   assert(this->congruent(f));
273   u.assertState(Spectral, Spectral);
274   f.assertState(Spectral, Spectral);
275
276   FlowField lapl_u;
277   lapl(u, lapl_u);
278   Real l2err = L2Dist(lapl_u, f);
279   Real bcerr = bcDist(u,bc);
280   cout << "PoissonSolver::verify(u, f) {\n";
281   cout << "  L2Norm(u)        == " << L2Norm(u) << endl;
282   cout << "  L2Norm(f)        == " << L2Norm(f) << endl;
283   cout << "  L2Norm(lapl u)   == " << L2Norm(lapl_u) << endl;
284   cout << "  L2Dist(lapl u, f) == " << l2err << endl;
285   cout << "  bcNorm(u)        == " << bcNorm(u) << endl;
286   cout << "  bcNorm(bc)       == " << bcNorm(bc) << endl;
287   cout << "  bcDist(u,bc)     == " << bcerr << endl;
288   cout << "} // PoissonSolver::verify(u, f)\n";
289   return l2err + bcerr;
290 }
```

Here is the call graph for this function:

7.19.4.7 Real PoissonSolver::verify (const FlowField & *u*, const FlowField & *f*) const

References FlowField::assertState(), bcNorm(), congruent(), diff(), L2Dist(), L2Norm(), lapl(), and Spectral.

```
231                                                 {
232   assert(this->congruent(u));
233   assert(this->congruent(f));
234   u.assertState(Spectral, Spectral);
235   f.assertState(Spectral, Spectral);
236 
237   FlowField lapl_u;
238   lapl(u, lapl_u);
239   FlowField diff(f);
240   diff -= lapl_u;
241 
242   /*********************************************
243   // Debugging output
244   f.saveSpectrum("f");
245   lapl_u.saveSpectrum("lapl_u");
246   diff.saveSpectrum("diff");
247 
248   for (int mx=0; mx<=3; ++mx) {
249     for (int mz=0; mz<=3; ++mz) {
250       string lbl = i2s(mx)+i2s(mz);
251       u.saveProfile(mx,mz, "u"+lbl);
252       f.saveProfile(mx,mz, "f"+lbl);
253       lapl_u.saveProfile(mx,mz, "lapl_u"+lbl);
254     }
255   }
256   *********************************************/
257 
258   Real l2err = L2Dist(lapl_u, f);
259   Real bcerr = bcNorm(u);
260   cout << "PoissonSolver::verify(u, f) {\n";
261   cout << "  L2Norm(u)        == " << L2Norm(u) << endl;
262   cout << "  L2Norm(f)        == " << L2Norm(f) << endl;
263   cout << "  L2Norm(lapl u)   == " << L2Norm(lapl_u) << endl;
264   cout << "  L2Dist(lapl u, f) == " << l2err << endl;
265   cout << "  bcNorm(u)        == " << bcerr << endl;
266   cout << "} // PoissonSolver::verify(u, f)\n";
267   return l2err + bcerr;
268 }
```

Here is the call graph for this function:

7.19.5 Member Data Documentation

7.19.5.1 Real PoissonSolver::a_ [protected]

Referenced by congruent(), geomCongruent(), operator=(), PoissonSolver(), PressureSolver::solve(), solve(), and PressureSolver::verify().

7.19.5.2 Real PoissonSolver::b_ [protected]

Referenced by congruent(), geomCongruent(), operator=(), PoissonSolver(), PressureSolver::solve(), solve(), and PressureSolver::verify().

7.19.5.3 HelmholtzSolver** PoissonSolver::helmholtz_ [protected]

Referenced by operator=(), PoissonSolver(), PressureSolver::solve(), solve(), and ~PoissonSolver().

7.19.5.4 Real PoissonSolver::Lx_ [protected]

Referenced by congruent(), geomCongruent(), operator=(), and PoissonSolver().

7.19.5.5 Real PoissonSolver::Lz_ [protected]

Referenced by congruent(), geomCongruent(), operator=(), and PoissonSolver().

7.19.5.6 int PoissonSolver::Mx_ [protected]

Referenced by congruent(), geomCongruent(), operator=(), PoissonSolver(), PressureSolver::solve(), solve(), PressureSolver::verify(), and ~PoissonSolver().

7.19.5.7 int PoissonSolver::My_ [protected]

Referenced by congruent(), geomCongruent(), operator=(), PoissonSolver(), PressureSolver::solve(), solve(), and PressureSolver::verify().

7.19.5.8 int PoissonSolver::Mz_ [protected]

Referenced by congruent(), geomCongruent(), operator=(), PoissonSolver(), PressureSolver::solve(), solve(), and PressureSolver::verify().

7.19.5.9 int PoissonSolver::Nd_ [protected]

Referenced by congruent(), operator=(), and solve().

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

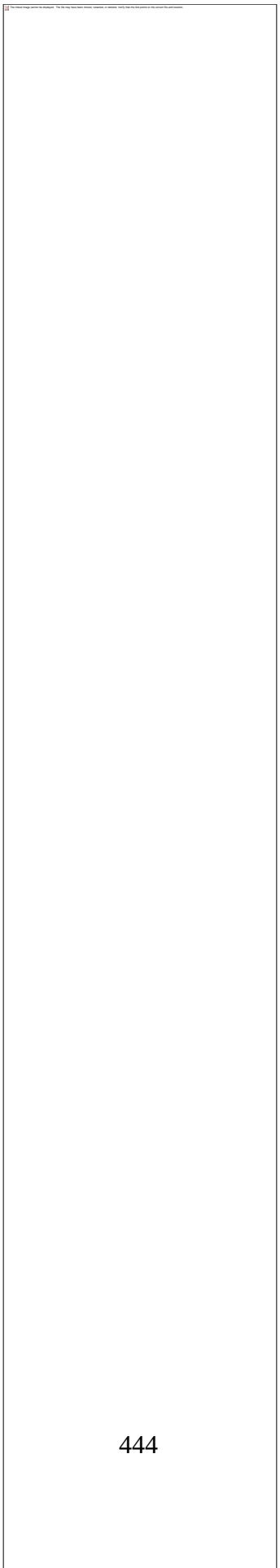
- /_cuda/channelflow-0.9.22/channelflow/poissonsolver.h
- /_cuda/channelflow-0.9.22/channelflow/poissonsolver.cpp

7.19.5.11

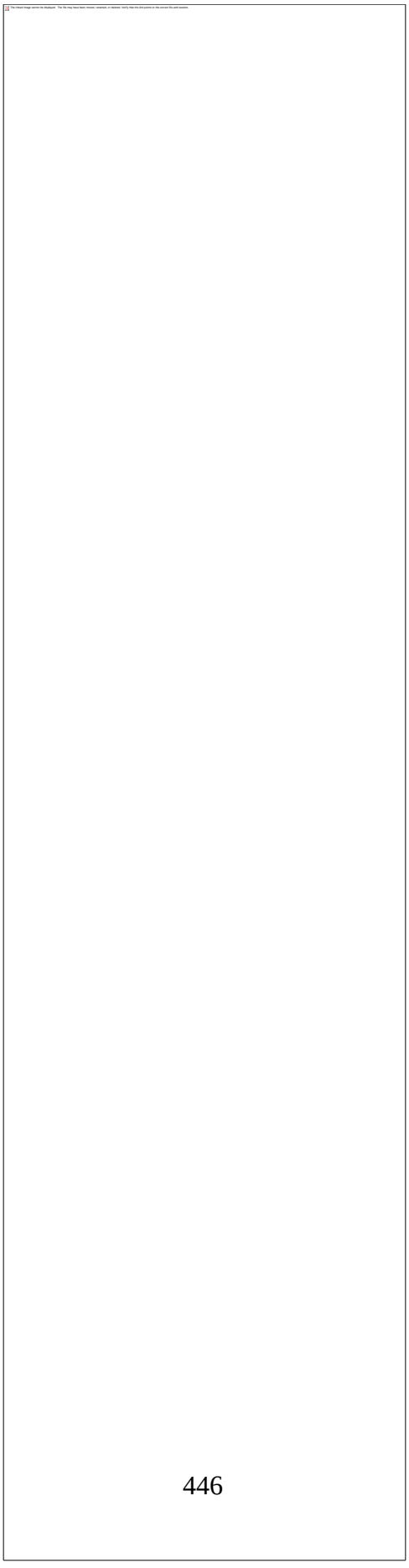
7.20 PPEDNS Class Reference

#include <dns.h>

Inheritance diagram for PPEDNS:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

Collaboration diagram for PPEDNS:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.20.1 Public Member Functions

- PPEDNS ()
- PPEDNS (const PPEDNS &dns)
- PPEDNS (FlowField &u, const ChebyCoeff &Ubase, Real nu, Real dt, const DNSFlags &flags, Real t=0)
- ~PPEDNS ()
- PPEDNS & operator= (const PPEDNS &dns)
- virtual void advance (FlowField &u, FlowField &q, int nSteps=1)
- virtual void reset_dt (Real dt)
- virtual bool push (const FlowField &u)
- virtual bool full () const

7.20.2 Private Member Functions

- virtual DNSAlgorithm * clone () const

7.20.3 Private Attributes

- Real eta_
- array< Real > alpha_
- array< Real > beta_
- array< FlowField > u_
- array< FlowField > f_
- HelmholtzSolver ** helmholtz_
- PressureSolver psolve_
- int countdown_

7.20.4 Constructor & Destructor Documentation

7.20.4.1 PPEDNS::PPEDNS ()

Referenced by clone().

```
2060   :
2061   DNSAlgorithm()
2062 {}
```

Here is the caller graph for this function:

7.20.4.2 PPEDNS::PPEDNS (const PPEDNS & *dns*)

References helmholtz_, DNSAlgorithm::Mx_, and DNSAlgorithm::Mz_.

```
2065   :
2066   DNSAlgorithm(dns),
2067   eta_(dns.eta_),
2068   alpha_(dns.alpha_),
2069   beta_(dns.beta_),
2070   u_(dns.u_),
2071   f_(dns.f_),
2072   psolve_(dns.psolve_)
2073 {
2074   // Copy tausolvers
2075   helmholtz_ = new HelmholtzSolver*[Mx_];       // new #1
2076   for (int mx=0; mx<Mx_; ++mx) {
2077     helmholtz_[mx] = new HelmholtzSolver[Mz_];  // new #2
2078     for (int mz=0; mz<Mz_; ++mz)
2079       helmholtz_[mx][mz] = dns.helmholtz_[mx][mz];
2080   }
2081 }
```

7.20.4.3 PPEDNS::PPEDNS (FlowField & *u*, const ChebyCoeff & *Ubase*, Real *nu*, Real *dt*, const DNSFlags & *flags*, Real *t* = 0)

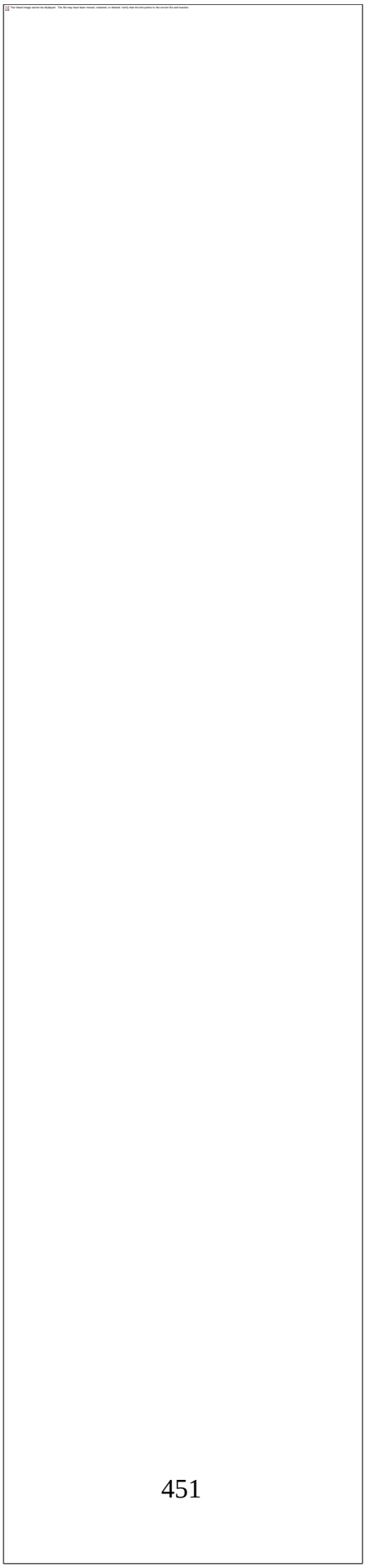
References FlowField::a(), alpha_, FlowField::b(), beta_, DNSAlgorithm::cfl_, countdown_, DNSFlags::dealias_xz(), DNSAlgorithm::dt_, eta_, f_, DNSAlgorithm::flags_, grad(), helmholtz_, FlowField::Lx(), FlowField::Lz(), DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, navierstokesNL(), DNSAlgorithm::Ninitsteps_, DNSFlags::nonlinearity, FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), DNSAlgorithm::order(), DNSAlgorithm::order_, pi, psolve_, reset_dt(), array< T >::resize(), FlowField::setToZero(), PressureSolver::solve(), DNSFlags::timestepping, DNSAlgorithm::tmp_, u_, and DNSAlgorithm::Ubase_.

```
2086   :
2087   DNSAlgorithm(u,Ubase,nu,dt,flags,t),
2088   psolve_(u, Ubase, nu, flags.nonlinearity)
2089 {
2090
2091   TimeStepMethod algorithm = flags.timestepping;
2092   int order = 2;
2093   switch (order) {
2094   case 1:
2095     order_ = 1;
2096     eta_ = 1.0;
2097     alpha_.resize(order_);
```

```
2098    beta_.resize(order_);
2099    alpha_[0] = -1.0;
2100    beta_[0]  =  1.0;
2101    break;
2102  case 2:
2103    order_ = 2;
2104    alpha_.resize(order_);
2105    beta_.resize(order_);
2106    eta_ = 1.5;
2107    alpha_[0] = -2.0; alpha_[1] =  0.5;
2108    beta_[0]  =  2.0;  beta_[1] = -1.0;
2109    break;
2110  case 3:
2111    order_ = 3;
2112    alpha_.resize(order_);
2113    beta_.resize(order_);
2114    eta_ = 11.0/6.0;
2115    alpha_[0] = -3.0;  alpha_[1] = 1.5; alpha_[2] = -1.0/3.0;
2116    beta_[0]  =  3.0;   beta_[1] = -3.0; beta_[2] = 1.0;
2117    break;
2118  case 4:
2119    order_ = 4;
2120    alpha_.resize(order_);
2121    beta_.resize(order_);
2122    eta_ = 25.0/12.0;
2123    alpha_[0] = -4.0; alpha_[1] =  3.0; alpha_[2] = -4.0/3.0; alpha_[3] = 0.25;
2124    beta_[0]  =  4.0;  beta_[1] = -6.0;  beta_[2] =  4.0;      beta_[3] = -1.0;
2125    break;
2126  default:
2127    cerr << "PPEDNS::PPEDNS(un,Ubase,nu,dt,flags,t0)\n"
2128         << "error: flags.timestepping == " << algorithm
2129         << "is a non-multistepping algorithm" << endl;
2130    exit(1);
2131  }
2132
2133  // Configure tausolvers
2134  helmholtz_ = new HelmholtzSolver*[Mx_];       // new #1
2135  for (int mx=0; mx<Mx_; ++mx)
2136    helmholtz_[mx] = new HelmholtzSolver[Mz_];  // new #2
2137
2138  reset_dt(dt_);
2139
2140  // Initialize arrays of previous u's and f's
2141  FlowField tmp(u);
2142  tmp.setToZero();
2143
```

```
2144   FlowField p(u.Nx(), u.Ny(), u.Nz(), 1, u.Lx(), u.Lz(), u.a(), u.b());
2145
2146   u_.resize(order_);
2147   f_.resize(order_);
2148   for (int j=0; j<order_; ++j) {
2149     u_[j] = tmp;
2150     f_[j] = tmp;
2151   }
2152   if (order_ > 0) { // should always be true
2153     u_[0] = u;
2154      //cout << "IG call navierstokesNL 7" <<endl;
2155     navierstokesNL(u_[0], Ubase_, f_[0], tmp_, flags_.nonlinearity);
2156     f_[0] *= -1.0;
2157
2158     psolve_.solve(p, u);
2159     grad(p, tmp);
2160     f_[0] -= tmp;
2161
2162     cfl_ = u_[0].CFLfactor(Ubase_);
2163     cfl_ *= flags_.dealias_xz() ? 2.0*pi/3.0*dt_ : pi*dt_;
2164   }
2165
2166   Ninitsteps_ = order_-1;
2167   countdown_ = Ninitsteps_;
2168 }
```

Here is the call graph for this function:



7.20.4.4 PPEDNS::~PPEDNS ()

References helmholtz_, and DNSAlgorithm::Mx_.

```
2170             {
2171   if (helmholtz_) {
2172     for (int mx=0; mx<Mx_; ++mx)
2173       delete[] helmholtz_[mx];  // undo new #2
2174     delete[] helmholtz_;       // undo new #1
2175   }
2176   helmholtz_ = 0;
2177 }
```

7.20.5 Member Function Documentation

7.20.5.1 void PPEDNS::advance (FlowField & *u*, FlowField & *q*, int *nSteps* = 1) [virtual]

Implements DNSAlgorithm.

References DNSAlgorithm::a(), ComplexChebyCoeff::add(), alpha_, DNSAlgorithm::b(), beta_, DNSAlgorithm::cfl_, FlowField::cmplx(), DNSFlags::constraint, DNSFlags::dealias_xz(), DNSAlgorithm::dPdxAct_, DNSAlgorithm::dPdxRef_, DNSAlgorithm::dt_, f_, DNSAlgorithm::flags_, grad(), helmholtz_, ComplexChebyCoeff::im, DNSAlgorithm::isAliasedMode(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), Vector::length(), ChebyCoeff::mean(), DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, navierstokesNL(), DNSFlags::nonlinearity, DNSAlgorithm::nu_, DNSAlgorithm::Nx_, DNSAlgorithm::Ny_, DNSAlgorithm::Nyd_, DNSAlgorithm::Nz_, DNSAlgorithm::order_, pi, PressureGradient, PrintAll, PrintTicks, PrintTime, psolve_, Re(), ComplexChebyCoeff::re, DNSAlgorithm::Rxk_, DNSAlgorithm::Ryk_, DNSAlgorithm::Rzk_, FlowField::setDealiased(), ComplexChebyCoeff::setToZero(), PressureSolver::solve(), HelmholtzSolver::solve(), swap(), DNSAlgorithm::t_, DNSAlgorithm::tmp_, u_, DNSAlgorithm::Ubase_, DNSAlgorithm::Ubaseyy_, DNSAlgorithm::UbulkAct_, DNSAlgorithm::UbulkBase_, DNSAlgorithm::UbulkRef_, DNSAlgorithm::uk_, DNSFlags::verbosity, DNSAlgorithm::vk_, and DNSAlgorithm::wk_.

```
2204                                          {
2205
2206   const int kxmax = un.kxmax();
2207   const int kzmax = un.kzmax();
2208   for (int step=0; step<Nsteps; ++step) {
2209
2210     //cout << "PPEDNS::advance(u,q,N) stack : " << endl;
2211     // Update each Fourier mode with time-stepping algorithm
```

```
2212    for (int mx=0; mx<Mx_; ++mx) {
2213      const int kx = un.kx(mx);
2214
2215      for (int mz=0; mz<Mz_; ++mz) {
2216        const int kz = un.kz(mz);
2217
2218        // Zero out the aliased modes
2219        if ((kx == kxmax || kz == kzmax) ||
2220            flags_.dealias_xz() && isAliasedMode(kx,kz)) {
2221          for (int ny=0; ny<Nyd_; ++ny) {
2222            u_[0].cmplx(mx,ny,mz,0) = 0.0;
2223            u_[0].cmplx(mx,ny,mz,1) = 0.0;
2224            u_[0].cmplx(mx,ny,mz,2) = 0.0;
2225            qn.cmplx(mx,ny,mz,0) = 0.0;
2226          }
2227          break;
2228        }
2229
2230        Rxk_.setToZero();
2231        Ryk_.setToZero();
2232        Rzk_.setToZero();
2233
2234        for (int j=0; j<order_; ++j) {
2235          const Real a = -alpha_[j]/dt_;
2236          const Real b = -beta_[j];
2237          for (int ny=0; ny<Nyd_; ++ny) {
2238            Rxk_.add(ny, a*u_[j].cmplx(mx,ny,mz,0)+b*f_[j].cmplx(mx,ny,mz,0));
2239            Ryk_.add(ny, a*u_[j].cmplx(mx,ny,mz,1)+b*f_[j].cmplx(mx,ny,mz,1));
2240            Rzk_.add(ny, a*u_[j].cmplx(mx,ny,mz,2)+b*f_[j].cmplx(mx,ny,mz,2));
2241          }
2242        }
2243        Rxk_ *= -1;
2244        Ryk_ *= -1;
2245        Rzk_ *= -1;
2246
2247        // Solve the Helmholtz solutions
2248        if (kx!=0 || kz!=0) {
2249          helmholtz_[mx][mz].solve(uk_.re,Rxk_.re,0,0);
2250          helmholtz_[mx][mz].solve(vk_.re,Ryk_.re,0,0);
2251          helmholtz_[mx][mz].solve(wk_.re,Rzk_.re,0,0);
2252          helmholtz_[mx][mz].solve(uk_.im,Rxk_.im,0,0);
2253          helmholtz_[mx][mz].solve(vk_.im,Ryk_.im,0,0);
2254          helmholtz_[mx][mz].solve(wk_.im,Rzk_.im,0,0);
2255        }
2256        else { // kx,kz == 0,0
2257
```

```
2258          if (Ubaseyy_.length() > 0)
2259            for (int ny=0; ny<Ny_; ++ny)
2260              Rxk_.re[ny] -= nu_*Ubaseyy_[ny]; // Rx has additional term
2261
2262          if (flags_.constraint == PressureGradient) {
2263            // pressure is supplied, put on RHS of tau eqn
2264            Rxk_.re[0] += dPdxRef_;
2265
2266            // Solve the tau equations
2267            helmholtz_[mx][mz].solve(uk_.re,Rxk_.re,0,0);
2268            helmholtz_[mx][mz].solve(vk_.re,Ryk_.re,0,0);
2269            helmholtz_[mx][mz].solve(wk_.re,Rzk_.re,0,0);
2270            helmholtz_[mx][mz].solve(uk_.im,Rxk_.im,0,0);
2271            helmholtz_[mx][mz].solve(vk_.im,Ryk_.im,0,0);
2272            helmholtz_[mx][mz].solve(wk_.im,Rzk_.im,0,0);
2273
2274            // Bulk vel is free variable determined from soln of tau eqn
2275            UbulkAct_ = UbulkBase_ + uk_.re.mean();
2276            dPdxAct_ = dPdxRef_;
2277          }
2278          else { // const bulk velocity
2279            // bulk velocity is supplied, put on RHS of tau eqn
2280            // dPdxAct (i.e. at next time step is solved for)
2281            // constraint:    UbulkBase + mean(u) = UbulkRef.
2282            helmholtz_[mx][mz].solve(uk_.re, dPdxAct_, Ryk_.re, UbulkRef_ -
 UbulkBase_,0,0);
2283            helmholtz_[mx][mz].solve(vk_.re,Ryk_.re,0,0);
2284            helmholtz_[mx][mz].solve(wk_.re,Rzk_.re,0,0);
2285            helmholtz_[mx][mz].solve(uk_.im,Rxk_.im,0,0);
2286            helmholtz_[mx][mz].solve(vk_.im,Ryk_.im,0,0);
2287            helmholtz_[mx][mz].solve(wk_.im,Rzk_.im,0,0);
2288
2289            UbulkAct_ = UbulkBase_ + uk_.re.mean(); // should == UbulkRef_
2290          }
2291        }
2292
2293        // Load solutions back into the external 3d data arrays.
2294        // Because of FFTW complex symmetries
2295        // The 0,0 mode must be real.
2296        // For Nx even, the kxmax,0 mode must be real
2297        // For Nz even, the 0,kzmax mode must be real
2298        // For Nx,Nz even, the kxmax,kzmax mode must be real
2299        if ((kx == 0 && kz == 0) ||
2300            (Nx_%2 == 0 && kx == kxmax && kz == 0) ||
2301            (Nz_%2 == 0 && kz == kzmax && kx == 0) ||
2302            (Nx_%2 == 0 && Nz_%2 == 0 && kx == kxmax && kz == kzmax)) {
```

```
2303
2304          for (int ny=0; ny<Nyd_; ++ny) {
2305            un.cmplx(mx,ny,mz,0) = Complex(Re(uk_[ny]), 0.0);
2306            un.cmplx(mx,ny,mz,1) = Complex(Re(vk_[ny]), 0.0);
2307            un.cmplx(mx,ny,mz,2) = Complex(Re(wk_[ny]), 0.0);
2308          }
2309        }
2310        // The normal case, for general kx,kz
2311        else
2312          for (int ny=0; ny<Nyd_; ++ny) {
2313            un.cmplx(mx,ny,mz,0) = uk_[ny];
2314            un.cmplx(mx,ny,mz,1) = vk_[ny];
2315            un.cmplx(mx,ny,mz,2) = wk_[ny];
2316          }
2317        // And now set the y-aliased modes to zero.
2318        for (int ny=Nyd_; ny<Ny_; ++ny) {
2319          un.cmplx(mx,ny,mz,0) = 0.0;
2320          un.cmplx(mx,ny,mz,1) = 0.0;
2321          un.cmplx(mx,ny,mz,2) = 0.0;
2322        }
2323      }
2324    }
2325
2326    // The solution is stored in un. Shift entire u and f arrays in time
2327    // and push un into u_[0]. Ie shift u_[K] <- u_[K-1] <- ... <- u_[0] <- un
2328    for (int j=order_-1; j>0; --j) {
2329      swap(f_[j], f_[j-1]);
2330      swap(u_[j], u_[j-1]);
2331    }
2332    if (order_ > 0) {
2333      u_[0] = un;
2334       //cout << "IG call navierstokesNL 8" <<endl;
2335      navierstokesNL(u_[0], Ubase_, f_[0], tmp_, flags_.nonlinearity);
2336      f_[0] *= -1.0;
2337
2338      psolve_.solve(qn, u_[0]);
2339      grad(qn, tmp_);
2340      f_[0] -= tmp_;
2341    }
2342    t_ += dt_;
2343
2344    if (flags_.verbosity == PrintTime || flags_.verbosity == PrintAll)
2345      cout << t_ << ' ' << flush;
2346    else if (flags_.verbosity == PrintTicks)
2347      cout << '.' << flush;
2348  }
```

```
2349
2350   cfl_ = u_[0].CFLfactor(Ubase_);
2351   cfl_ *= flags_.dealias_xz() ? 2.0*pi/3.0*dt_ : pi*dt_;
2352
2353   // If using dealiasing, set flag in FlowField that compactifies binary IO
2354   un.setDealiased(flags_.dealias_xz());
2355   qn.setDealiased(flags_.dealias_xz());
2356
2357   if (flags_.verbosity == PrintTime || flags_.verbosity == PrintAll)
2358     cout << endl;
2359
2360   return;
2361 }
```

Here is the call graph for this function:

7.20.5.2 DNSAlgorithm * PPEDNS::clone () const [private, virtual]

Implements DNSAlgorithm.

References PPEDNS().

```
2179                                {
2180   return new PPEDNS(*this);
2181 }
```

Here is the call graph for this function:

7.20.5.3 bool PPEDNS::full () const [virtual]

Reimplemented from DNSAlgorithm.

References countdown_.

Referenced by push().

```
2407                        {
2408   return (countdown_ == 0) ? true : false;
2409 }
```

Here is the caller graph for this function:

7.20.5.4 PPEDNS& PPEDNS::operator= (const PPEDNS & *dns*)

Reimplemented from DNSAlgorithm.

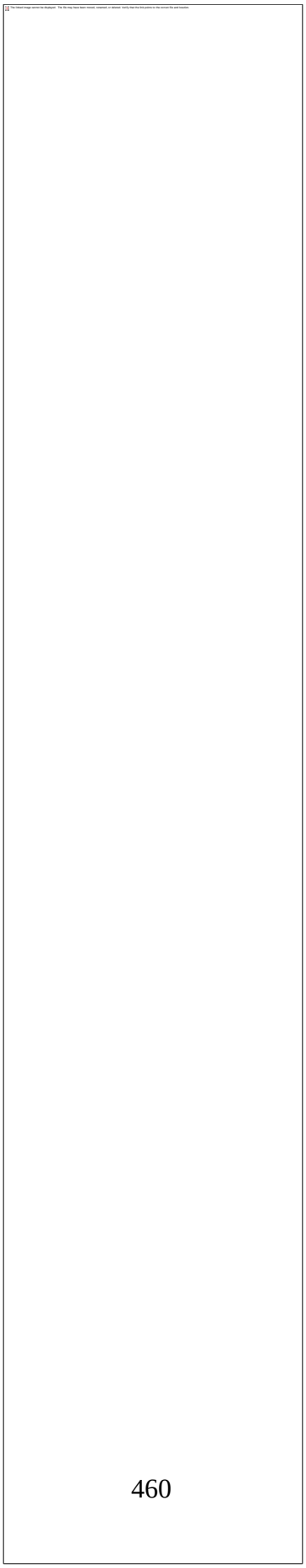
7.20.5.5 bool PPEDNS::push (const FlowField & *u*) [virtual]

Reimplemented from DNSAlgorithm.

References FlowField::a(), FlowField::b(), DNSAlgorithm::cfl_, countdown_, DNSFlags::dealias_xz(), DNSAlgorithm::dt_, f_, DNSAlgorithm::flags_, full(), grad(), FlowField::Lx(), FlowField::Lz(), navierstokesNL(), DNSFlags::nonlinearity, FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), DNSAlgorithm::order_, pi, psolve_, PressureSolver::solve(), swap(), DNSAlgorithm::t_, DNSAlgorithm::tmp_, u_, and DNSAlgorithm::Ubase_.

```
2379                                {
2380
2381   // Let K = order-1. Arrays are then u_[0:K], f_[0:K]
2382   // Shift u_[K] <- u_[K-1] <- ... <- u_[0] <- un
2383   for (int j=order_-1; j>0; --j) {
2384     swap(u_[j], u_[j-1]);
2385     swap(f_[j], f_[j-1]);
2386   }
2387
2388   FlowField p(un.Nx(), un.Ny(), un.Nz(), 1, un.Lx(), un.Lz(), un.a(), un.b());
2389
2390   if (order_ > 0) {
2391     u_[0] = un;
2392      //cout << "IG call navierstokesNL 8" <<endl;
2393     navierstokesNL(u_[0], Ubase_, f_[0], tmp_, flags_.nonlinearity);
2394     psolve_.solve(p, u_[0]);
2395     grad(p, tmp_);
2396     f_[0] -= tmp_;
2397
2398     cfl_ = u_[0].CFLfactor(Ubase_);
2399     cfl_ *= flags_.dealias_xz() ? 2.0*pi/3.0*dt_ : pi*dt_;
2400   }
2401
2402   t_ += dt_;
2403   --countdown_;
2404   return full();
2405 }
```

Here is the call graph for this function:


The mixed image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.20.5.6 void PPEDNS::reset_dt (Real *dt*) [virtual]

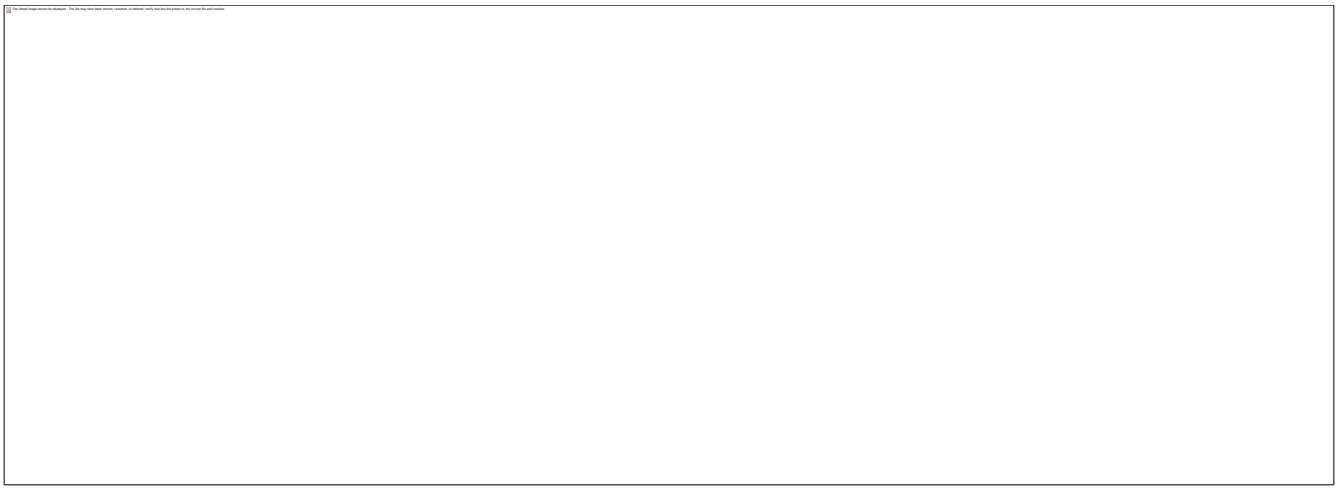
Implements DNSAlgorithm.

References DNSAlgorithm::a_, DNSAlgorithm::b_, DNSAlgorithm::cfl_, countdown_, DNSFlags::dealias_xz(), DNSAlgorithm::dt_, eta_, DNSAlgorithm::flags_, helmholtz_, DNSAlgorithm::isAliasedMode(), FlowField::kx(), FlowField::kz(), DNSAlgorithm::Lx_, DNSAlgorithm::Lz_, DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, DNSAlgorithm::Ninitsteps_, DNSAlgorithm::nu_, DNSAlgorithm::Ny_, pi, square(), and DNSAlgorithm::tmp_.

Referenced by PPEDNS().

```
2183                         {
2184   cfl_ *= dt/dt_;
2185   //nu_ = nu;
2186   dt_ = dt;
2187   const Real c = 4.0*square(pi)*nu_;
2188   for (int mx=0; mx<Mx_; ++mx) {
2189     int kx = tmp_.kx(mx);
2190     for (int mz=0; mz<Mz_; ++mz) {
2191       int kz = tmp_.kz(mz);
2192       Real lambda = eta_/dt_ + c*(square(kx/Lx_) + square(kz/Lz_));
2193       // When using dealiasing, some modes get set to zero, rather than
2194       // updated with momentum eqns. Don't initialize TauSolvers for these.
2195       if (!flags_.dealias_xz() || !isAliasedMode(kx,kz))
2196         helmholtz_[mx][mz] = HelmholtzSolver(Ny_, a_, b_, lambda, nu_);
2197     }
2198   }
2199   // Start from beginning on initialization
2200   countdown_ = Ninitsteps_;
2201 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.20.6 Member Data Documentation

7.20.6.1 array<Real> PPEDNS::alpha_ [private]

Referenced by advance(), and PPEDNS().

7.20.6.2 array<Real> PPEDNS::beta_ [private]

Referenced by advance(), and PPEDNS().

7.20.6.3 int PPEDNS::countdown_ [private]

Referenced by full(), PPEDNS(), push(), and reset_dt().

7.20.6.4 Real PPEDNS::eta_ [private]

Referenced by PPEDNS(), and reset_dt().

7.20.6.5 array<FlowField> PPEDNS::f_ [private]

Referenced by advance(), PPEDNS(), and push().

7.20.6.6 HelmholtzSolver** PPEDNS::helmholtz_ [private]

Referenced by advance(), PPEDNS(), reset_dt(), and ~PPEDNS().

7.20.6.7 PressureSolver PPEDNS::psolve_ [private]

Referenced by advance(), PPEDNS(), and push().

7.20.6.8 array<FlowField> PPEDNS::u_ [private]

Referenced by advance(), PPEDNS(), and push().

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

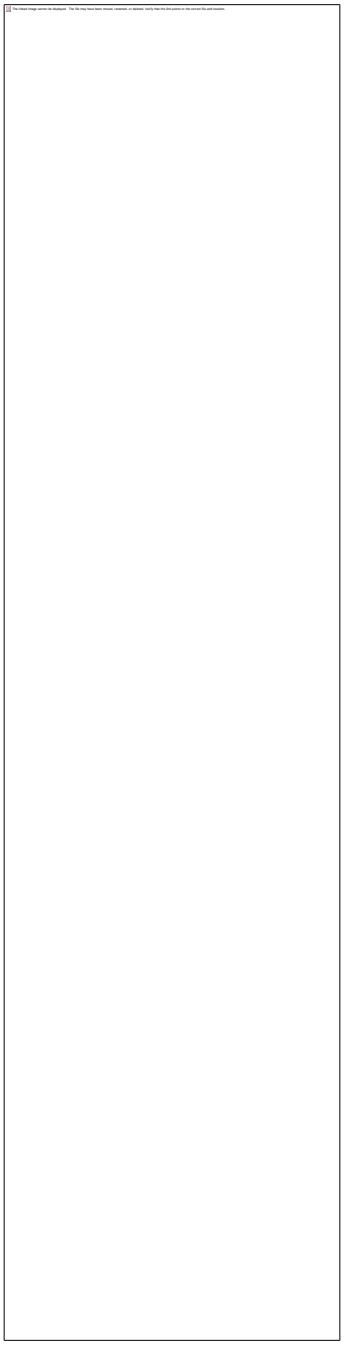
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.20.6.10

7.21 PressureSolver Class Reference

#include <poissonsolver.h>

Inheritance diagram for PressureSolver:



Collaboration diagram for PressureSolver:

7.21.1 Public Member Functions

- PressureSolver ()
- PressureSolver (int Nx, int Ny, int Nz, Real Lx, Real Lz, Real a, Real b, const ChebyCoeff &U, Real nu, NonlinearMethod nonl_method)
- PressureSolver (const FlowField &u, Real nu, NonlinearMethod nonl_method)
- PressureSolver (const FlowField &u, const ChebyCoeff &U, Real nu, NonlinearMethod nonl_method)
- ~PressureSolver ()
- void solve (FlowField &p, const FlowField &u)
- Real verify (const FlowField &p, const FlowField &u)

7.21.2 Private Attributes

- ChebyCoeff U_
- ChebyTransform trans_
- FlowField nonl_
- FlowField tmp_
- FlowField div_nonl_
- Real nu_
- NonlinearMethod nonl_method_

7.21.3 Constructor & Destructor Documentation

7.21.3.1 PressureSolver::PressureSolver ()

```
295   :
296   PoissonSolver(),
297   U_(),
298   trans_(1),
299   nonl_(),
300   tmp_(),
301   div_nonl_(),
302   nu_(0)
303 {}
```

7.21.3.2 PressureSolver::PressureSolver (int *Nx*, int *Ny*, int *Nz*, Real *Lx*, Real *Lz*, Real *a*, Real *b*, const ChebyCoeff & *U*, Real *nu*, NonlinearMethod *nonl_method*)

References ChebyCoeff::makeSpectral(), ChebyCoeff::N(), trans_, and U_.

```
309   :
```

```
310   PoissonSolver(Nx, Ny, Nz, 1, Lx, Lz, a, b),
311   U_(U),
312   trans_(Ny),
313   nonl_(Nx, Ny, Nz, 3, Lx, Lz, a, b),
314   tmp_(), // geom will be set in call to navierstokesNL
315   div_nonl_(Nx, Ny, Nz, 1, Lx, Lz, a, b),
316   nu_(nu),
317   nonl_method_(nonl)
318 {
319   assert(U_.N() == Ny);
320   U_.makeSpectral(trans_);
321 }
```

Here is the call graph for this function:

7.21.3.3 PressureSolver::PressureSolver (const FlowField & *u*, Real *nu*, NonlinearMethod *nonl_method*)

```
324   :
325   PoissonSolver(u.Nx(), u.Ny(), u.Nz(), 1, u.Lx(), u.Lz(), u.a(), u.b()),
326   U_(u.Ny(), u.a(), u.b(), Spectral),
327   trans_(u.Ny()),
328   nonl_(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b()),
329   tmp_(), // geom will be set in call to navierstokesNL
330   div_nonl_(u.Nx(), u.Ny(), u.Nz(), 1, u.Lx(), u.Lz(), u.a(), u.b()),
331   nu_(nu),
332   nonl_method_(nl)
333 {}
```

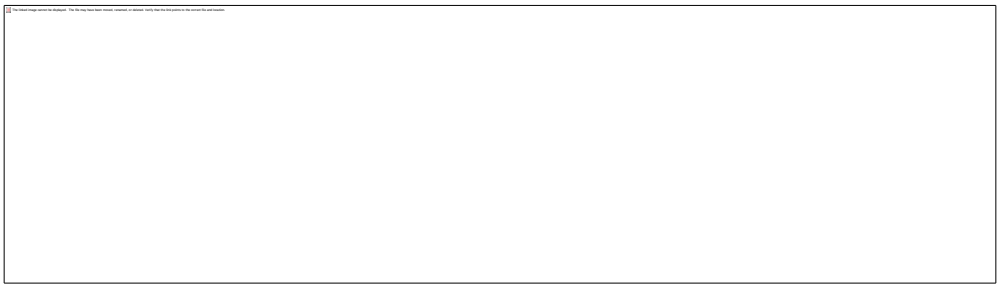
7.21.3.4 PressureSolver::PressureSolver (const FlowField & *u*, const ChebyCoeff & *U*, Real *nu*, NonlinearMethod *nonl_method*)

References ChebyCoeff::makeSpectral(), ChebyCoeff::N(), FlowField::Ny(), trans_, and U_.

```
338   :
339   PoissonSolver(u.Nx(), u.Ny(), u.Nz(), 1, u.Lx(), u.Lz(), u.a(), u.b()),
340   U_(U),
```

```
341   trans_(u.Ny()),
342   nonl_(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b()),
343   tmp_(), // geom will be set in call to navierstokesNL
344   div_nonl_(u.Nx(), u.Ny(), u.Nz(), 1, u.Lx(), u.Lz(), u.a(), u.b()),
345   nu_(nu),
346   nonl_method_(nonl_method)
347 {
348   assert(U_.N() == u.Ny());
349   U_.makeSpectral(trans_);
350 }
```

Here is the call graph for this function:



7.21.3.5 PressureSolver::~PressureSolver ()

```
352 {}
```

7.21.4 Member Function Documentation

7.21.4.1 void PressureSolver::solve (FlowField & *p*, const FlowField & *u*)

References PoissonSolver::a_, PoissonSolver::b_, FlowField::cmplx(), PoissonSolver::congruent(), diff(), diff2(), div(), div_nonl_, FlowField::Dx(), FlowField::Dz(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), exp(), PoissonSolver::geomCongruent(), PoissonSolver::helmholtz_, HelmholtzSolver::lambda(), ComplexChebyCoeff::makeSpectral(), PoissonSolver::Mx_, PoissonSolver::My_, PoissonSolver::Mz_, navierstokesNL(), nonl_, nonl_method_, nu_, Physical, ComplexChebyCoeff::set(), ComplexChebyCoeff::setState(), FlowField::setState(), FlowField::setToZero(), Spectral, tmp_, trans_, U_, FlowField::xzstate(), FlowField::ygridpts(), and FlowField::ystate().

Referenced by PPEDNS::advance(), PPEDNS::PPEDNS(), and PPEDNS::push().

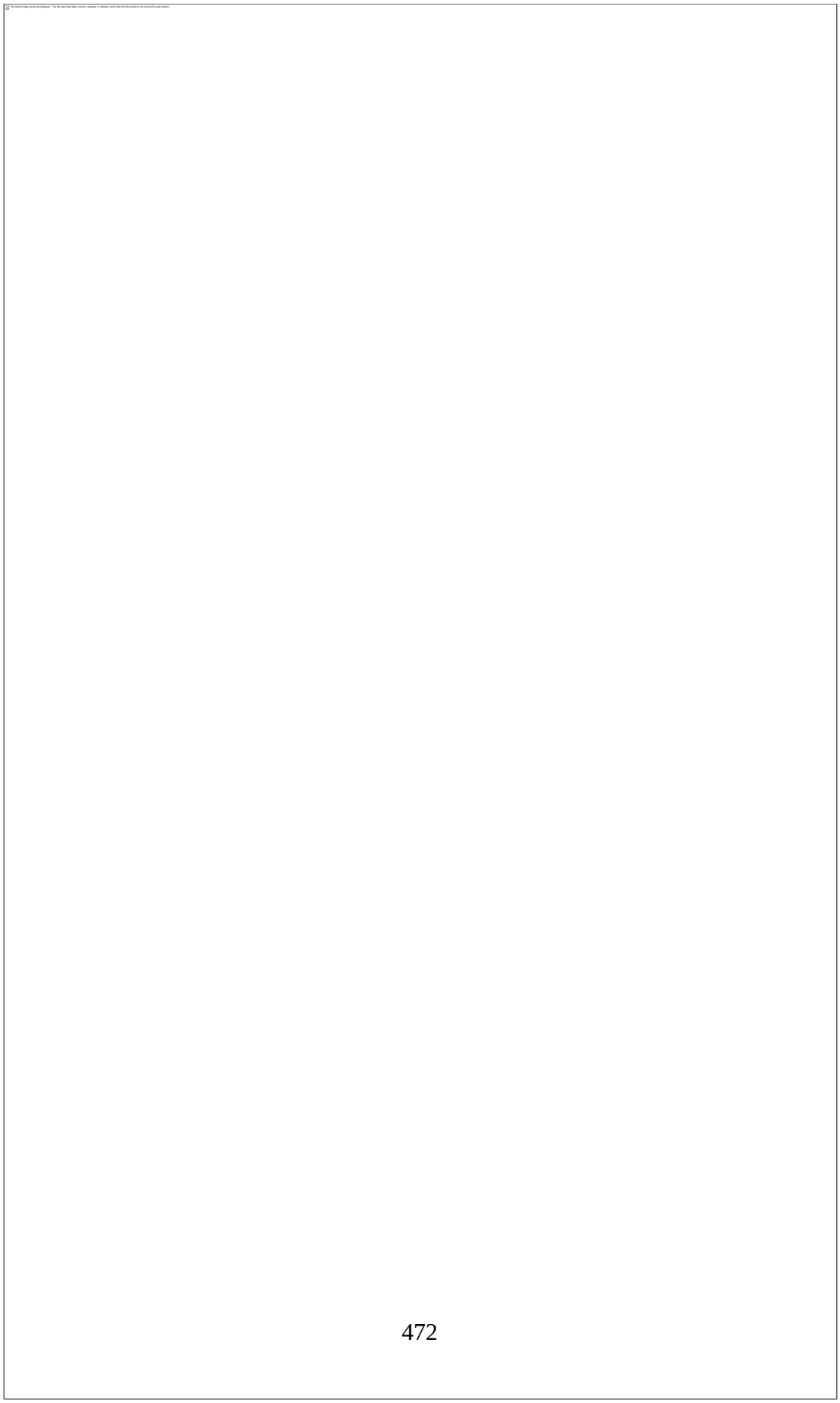
```
354                                         {
355   assert(u.xzstate() == Spectral && u.ystate() == Spectral);
```

```
356  assert(congruent(p));
357  assert(geomCongruent(u));
358
359  p.setToZero();
360  p.setState(Spectral, Spectral);
361
362  // I. Get particular solution satisfying
363  // lapl p = -div(nonl(u)) BCs p=0
364  //        = -div(u grad u) for Convection nonlinearity, for example
365
366  navierstokesNL(u, U_, nonl_, tmp_, nonl_method_);
367  div(nonl_, div_nonl_);
368  div_nonl_ *= -1.0;
369
370  PoissonSolver::solve(p, div_nonl_);
371
372  // II. Adjust solution to match von Neumann BCs
373  //   dp/dy ==  nu d^2 v/dy^2  at y=a and y=b
374  // or
375  //   dp/dy ==  nu (v_xx + v_zz - u_xy - w_yz) at y=a and y=b
376
377  // Derivation of calculation in 5/19/05 notes, jfg. Idea is that
378  //   g = c exp(lambda y) + d exp(-lambda y)
379  // satisfies
380  //   g" - lambda g = 0 with nonzero BCs.
381  // Find the particular values of c and d such that adding g to p leaves
382  //   p" - lambda p = f unchanged but sets dp/dy = nu d^v/dy^2 at y= a,b.
383
384  // These tmp variables are for v_yy form of BCs
385  ComplexChebyCoeff vk(My_, a_, b_, Spectral);
386  ComplexChebyCoeff vkyy(My_, a_, b_, Spectral);
387
388
389  // These tmp variables are for (v_xx + v_zz - u_xy - w_yz) form of BCs
390  // Moderately INEFFICIENT implementation for PPE BCs, initial testing...
391  ComplexChebyCoeff ukx(My_, a_, b_, Spectral);
392  ComplexChebyCoeff wkz(My_, a_, b_, Spectral);
393  //ComplexChebyCoeff vkxx(My_, a_, b_, Spectral);
394  //ComplexChebyCoeff vkzz(My_, a_, b_, Spectral);
395  ComplexChebyCoeff ukxy(My_, a_, b_, Spectral);
396  ComplexChebyCoeff wkyz(My_, a_, b_, Spectral);
397
398  ComplexChebyCoeff pk(My_, a_, b_, Spectral);
399  ComplexChebyCoeff pky(My_, a_, b_, Spectral);
400  ComplexChebyCoeff gk(My_, a_, b_, Physical);
401
```

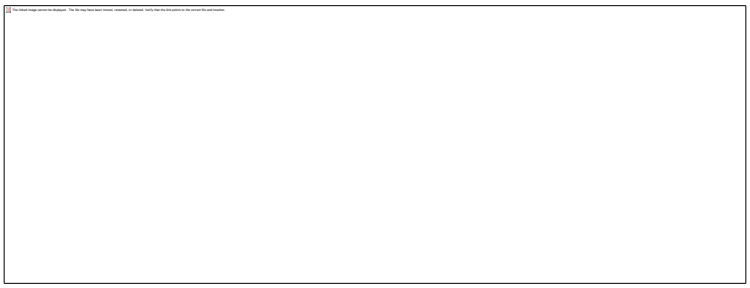
```
402  Vector y = p.ygridpts();
403  for (int mx=0; mx<Mx_; ++mx)
404    for (int mz=0; mz<Mz_; ++mz) {
405
406      // Don't modify the mx=mz=0 (kx=kz=0) solution, dirichlet BCs are fine
407      if (mx != 0 || mz != 0) {
408
409        Real lambda = helmholtz_[mx][mz].lambda();
410        assert(lambda > 0);
411        Real gamma = sqrt(lambda);
412
413        Complex Dx = u.Dx(mx);
414        Complex Dz = u.Dz(mz);
415        Complex Dx2 = Dx*Dx;
416        Complex Dz2 = Dz*Dz;
417
418        for (int my=0; my<My_; ++my) {
419          pk.set(my, p.cmplx(mx,my,mz,0));
420          vk.set(my, u.cmplx(mx,my,mz,1));
421
422          ukx.set(my, Dx*u.cmplx(mx,my,mz,0));
423          wkz.set(my, Dz*u.cmplx(mx,my,mz,2));
424          //vkxx.set(my, Dx2*u.cmplx(mx,my,mz,1));
425          //vkzz.set(my, Dz2*u.cmplx(mx,my,mz,1));
426        }
427        diff(pk,pky);
428        diff2(vk,vkyy);
429
430        diff(ukx, ukxy);
431        diff(wkz, wkyz);
432
433        //Complex alpha = nu_*vkyy.eval_a() - pky.eval_a();
434        //Complex  beta = nu_*vkyy.eval_b() - pky.eval_b();
435
436        Complex alpha = -nu_*(ukxy.eval_a() + wkyz.eval_a()) - pky.eval_a();
437        Complex beta  = -nu_*(ukxy.eval_b() + wkyz.eval_b()) - pky.eval_b();
438
439        Real delta = gamma*(exp(gamma*(a_-b_)) - exp(gamma*(b_-a_)));
440
441        Complex c = (alpha*exp(-gamma*b_) - beta*exp(-gamma*a_))/delta;
442        Complex d = (alpha*exp(gamma*b_) - beta*exp(gamma*a_))/delta;
443
444        gk.setState(Physical);
445        for (int my=0; my<My_; ++my)
446          gk.set(my, c*exp(gamma*y[my]) + d*exp(-gamma*y[my]));
447        gk.makeSpectral(trans_);
```

```
448
449         for (int my=0; my<My_; ++my)
450           p.cmplx(mx,my,mz,0) += gk[my];
451       }
452     }
453   return;
454 }
```

Here is the call graph for this function:


The inbuilt image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

Here is the caller graph for this function:



7.21.4.2 Real PressureSolver::verify (const FlowField & *p*, const FlowField & *u*)

References FlowField::a(), PoissonSolver::a_, FlowField::b(), PoissonSolver::b_, bcDist(), bcNorm(), FlowField::cmplx(), PoissonSolver::congruent(), div(), div_nonl_, PoissonSolver::geomCongruent(), L2Dist(), L2Norm(), lapl(), FlowField::Lx(), FlowField::Lz(), PoissonSolver::Mx_, PoissonSolver::My_, PoissonSolver::Mz_, navierstokesNL(), nonl_, nonl_method_, nu_, FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Spectral, tmp_, U_, FlowField::xzstate(), ydiff(), and FlowField::ystate().

```
457                                                 {
458   assert(u.xzstate() == Spectral && u.ystate() == Spectral);
459   assert(congruent(p));
460   assert(geomCongruent(u));
461
462   // I. Verify that solution satisfies lapl p = -div(nonl(u))
463   navierstokesNL(u, U_, nonl_, tmp_, nonl_method_);
464   div(nonl_, div_nonl_);
465   div_nonl_ *= -1.0;
466
467   PoissonSolver::verify(p, div_nonl_);
468
469   FlowField lapl_p;
470   lapl(p, lapl_p);
471
472   Real l2err = L2Dist(lapl_p, div_nonl_);
473   cout << "PressureSolver::verify(p,u) {\n";
474   cout << "  L2Norm(u)          == " << L2Norm(u) << endl;
475   cout << "  L2Norm(div(nonl(u)) == " << L2Norm(div_nonl_) << endl;
476   cout << "  L2Norm(lapl p)     == " << L2Norm(lapl_p) << endl;
477   cout << "  L2Dist(lapl p, div(nonl(u))) == " << l2err << endl;
478
479   // I. Verify that solution satisfies dpdy = d^2 /dy^2 (u+U) on boundary
480   FlowField dpdy;
481   ydiff(p, dpdy);
```

```
482
483   FlowField v(u.Nx(), u.Ny(), u.Nz(), 1, u.Lx(), u.Lz(), u.a(), u.b(), Spectral, Spectral);
484   for (int my=0; my<My_; ++my)
485     for (int mx=0; mx<Mx_; ++mx)
486       for (int mz=0; mz<Mz_; ++mz)
487         v.cmplx(mx,my,mz,0) = u.cmplx(mx,my,mz,1);
488
489   FlowField nu_vyy;
490   ydiff(v, nu_vyy, 2);
491   nu_vyy *= nu_;
492
493   Real bcerr = bcDist(dpdy, nu_vyy);
494   cout << "  bcNorm(dpdy)         == " << bcNorm(dpdy) << endl;
495   cout << "  bcNorm(nu_vyy)       == " << bcNorm(nu_vyy) << endl;
496   cout << "  bcDist(dpdy, nu_vyy) == " << bcerr << endl;
497
498   ComplexChebyCoeff vk(My_, a_, b_, Spectral);
499   ComplexChebyCoeff vkyy(My_, a_, b_, Spectral);
500   ComplexChebyCoeff pk(My_, a_, b_, Spectral);
501   ComplexChebyCoeff pky(My_, a_, b_, Spectral);
502
503   return l2err + bcerr;
504 }
```

Here is the call graph for this function:

7.21.5 Member Data Documentation

7.21.5.1 FlowField PressureSolver::div_nonl_ [private]

Referenced by solve(), and verify().

7.21.5.2 FlowField PressureSolver::nonl_ [private]

Referenced by solve(), and verify().

7.21.5.3 NonlinearMethod PressureSolver::nonl_method_ [private]

Referenced by solve(), and verify().

7.21.5.4 Real PressureSolver::nu_ [private]

Referenced by solve(), and verify().

7.21.5.5 FlowField PressureSolver::tmp_ [private]

Referenced by solve(), and verify().

7.21.5.6 ChebyTransform PressureSolver::trans_ [private]

Referenced by PressureSolver(), and solve().

7.21.5.7 ChebyCoeff PressureSolver::U_ [private]

Referenced by PressureSolver(), solve(), and verify().

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

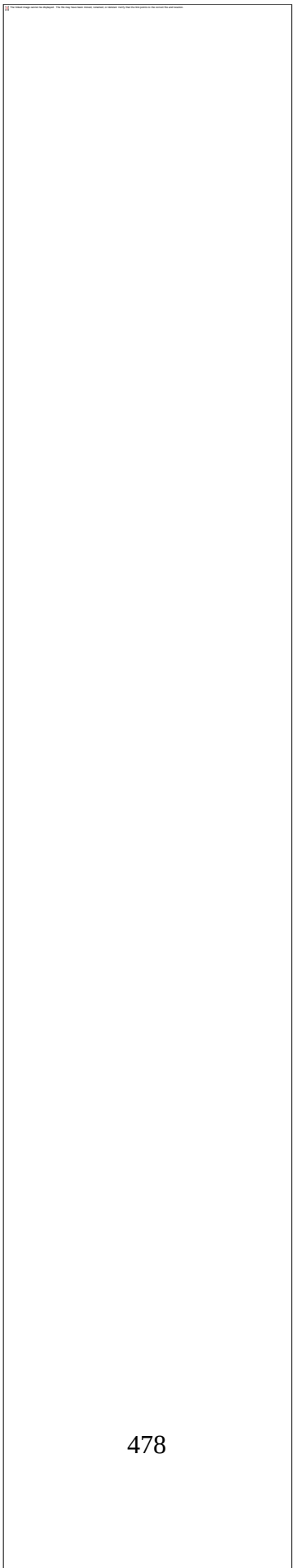
- /_cuda/channelflow-0.9.22/channelflow/poissonsolver.h
- /_cuda/channelflow-0.9.22/channelflow/poissonsolver.cpp

7.21.5.9

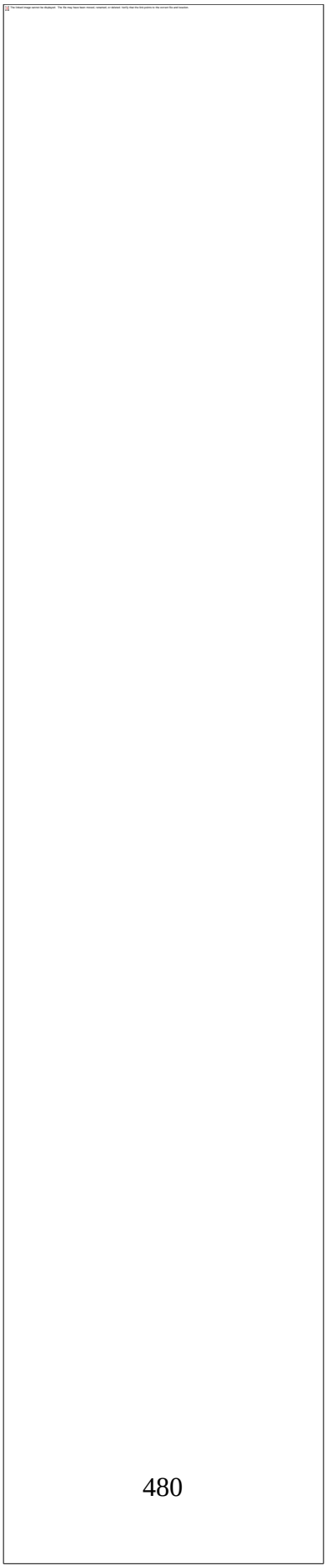
7.22 RungeKuttaDNS Class Reference

#include <dns.h>

Inheritance diagram for RungeKuttaDNS:


The inkwell image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

Collaboration diagram for RungeKuttaDNS:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.22.1 Public Member Functions

- RungeKuttaDNS ()
- RungeKuttaDNS (const RungeKuttaDNS &dns)
- RungeKuttaDNS (FlowField &u, const ChebyCoeff &Ubase, Real nu, Real dt, const DNSFlags &flags, Real t=0)
- ~RungeKuttaDNS ()
- RungeKuttaDNS & operator= (const RungeKuttaDNS &dns)
- virtual void advance (FlowField &u, FlowField &q, int nSteps=1)
- virtual void reset_dt (Real dt)
- virtual DNSAlgorithm * clone () const

7.22.2 Protected Attributes

- int Nsubsteps_
- FlowField Qj1_
- FlowField Qj_
- array< Real > A_
- array< Real > B_
- array< Real > C_
- TauSolver *** tausolver_

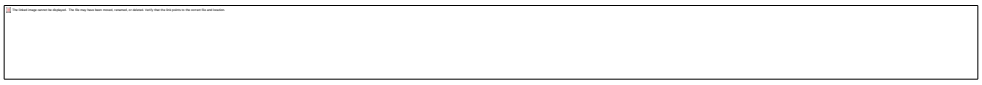
7.22.3 Constructor & Destructor Documentation

7.22.3.1 RungeKuttaDNS::RungeKuttaDNS ()

Referenced by clone().

```
1416   :
1417   DNSAlgorithm(),
1418   Qj1_(),
1419   Qj_()
1420 {}
```

Here is the caller graph for this function:



7.22.3.2 RungeKuttaDNS::RungeKuttaDNS (const RungeKuttaDNS & *dns*)

References DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, Nsubsteps_, and tausolver_.

```
1423   :
1424   DNSAlgorithm(dns),
1425   Nsubsteps_(dns.Nsubsteps_),
1426   Qj1_(dns.Qj1_),
1427   Qj_(dns.Qj_),
1428   A_(dns.A_),
1429   B_(dns.B_),
1430   C_(dns.C_)
1431 {
1432   // Allocate memory for [Nsubsteps x Mx_ x Mz_] Tausolver arrays
1433   // and copy tausolvers from dns argument
1434   tausolver_ = new TauSolver**[Nsubsteps_];    // new #1
1435   for (int j=0; j<Nsubsteps_; ++j) {
1436     tausolver_[j] = new TauSolver*[Mx_];       // new #2
1437     for (int mx=0; mx<Mx_; ++mx) {
1438       tausolver_[j][mx] = new TauSolver[Mz_];  // new #3
1439       for (int mz=0; mz<Mz_; ++mz)
1440         tausolver_[j][mx][mz] = dns.tausolver_[j][mx][mz];
1441
1442     }
1443   }
1444 }
```

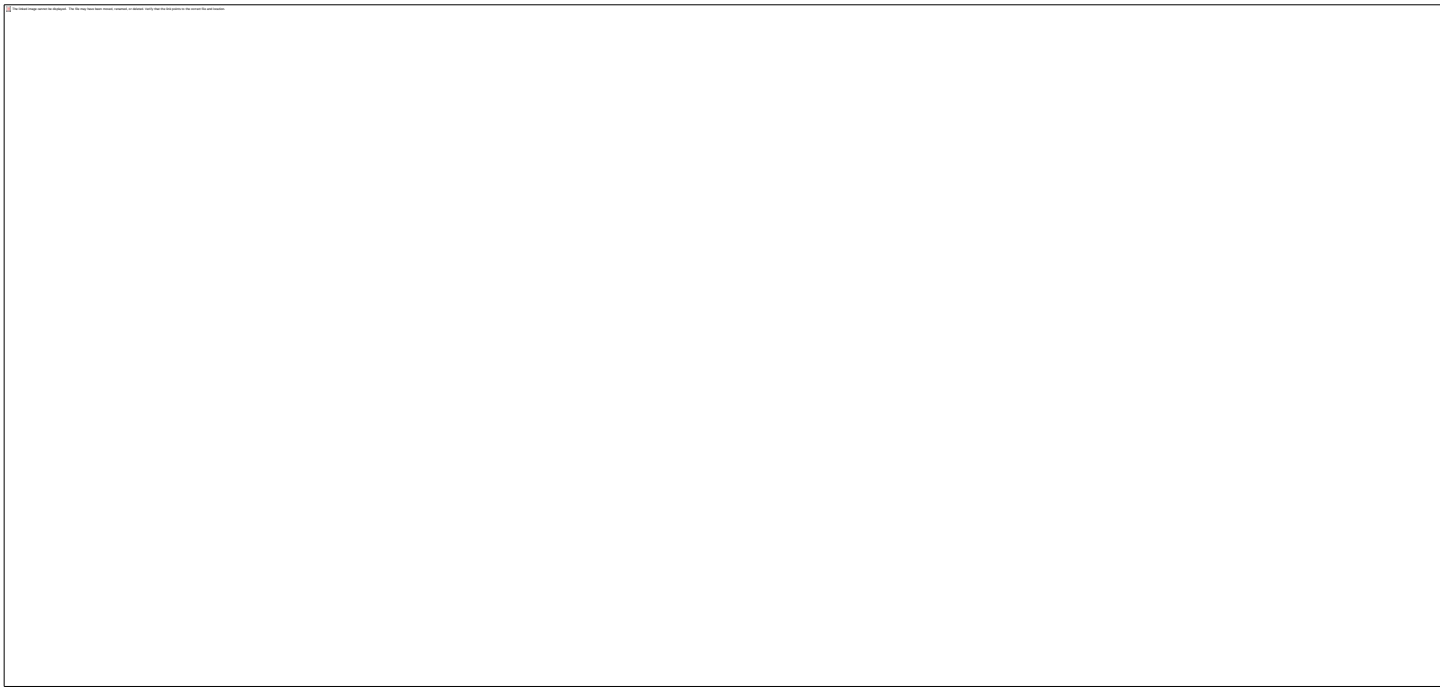
7.22.3.3 RungeKuttaDNS::RungeKuttaDNS (FlowField & *u*, const ChebyCoeff & *Ubase*, Real *nu*, Real *dt*, const DNSFlags & *flags*, Real *t* = 0)

References A_, B_, C_, CNRK2, DNSAlgorithm::dt_, DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, DNSAlgorithm::Ninitsteps_, Nsubsteps_, DNSAlgorithm::order_, Qj1_, Qj_, reset_dt(), array< T >::resize(), FlowField::setToZero(), tausolver_, and DNSFlags::timestepping.

```
1451   :
1452   DNSAlgorithm(u,Ubase,nu,dt,flags,t),
1453   Qj1_(u),
1454   Qj_(u)
1455 {
1456   Qj1_.setToZero();
1457   Qj_.setToZero();
1458
1459   TimeStepMethod algorithm = flags.timestepping;
1460   switch (algorithm) {
1461   case CNRK2:
1462     order_ = 2;
1463     Nsubsteps_ = 3;
1464     Ninitsteps_ = 0;
```

```
1465    A_.resize(Nsubsteps_);
1466    B_.resize(Nsubsteps_);
1467    C_.resize(Nsubsteps_);
1468    A_[0] = 0.0;     A_[1] = -5.0/9.0;  A_[2] = -153.0/128.0; // Peyret A
1469    B_[0] = 1.0/3.0; B_[1] = 15.0/16.0; B_[2] = 8.0/15.0;     // Peyret B
1470    C_[0] = 1.0/6.0; C_[1] = 5.0/24.0;  C_[2] = 1.0/8.0;      // Peyret B'
1471    break;
1472  default:
1473    cerr << "RungeKuttaDNS::RungeKuttaDNS(un,Ubase,nu,dt,flags,t0)\n"
1474         << "error: flags.timestepping == " << algorithm
1475         << " is a non-runge-kutta algorithm" << endl;
1476    exit(1);
1477  }
1478
1479  // Allocate memory for [Nsubsteps x Mx_ x Mz_] Tausolver array
1480  tausolver_ = new TauSolver**[Nsubsteps_];          // new #1
1481  for (int j=0; j<Nsubsteps_; ++j) {
1482    tausolver_[j] = new TauSolver*[Mx_];       // new #2
1483    for (int mx=0; mx<Mx_; ++mx)
1484      tausolver_[j][mx] = new TauSolver[Mz_];  // new #3
1485  }
1486  reset_dt(dt_);
1487 }
```

Here is the call graph for this function:



7.22.3.4 RungeKuttaDNS::~RungeKuttaDNS ()

References DNSAlgorithm::Mx_, Nsubsteps_, and tausolver_.

```
1489                          {
1490   if (tausolver_) {
1491     for (int j=0; j<Nsubsteps_; ++j) {
1492       for (int mx=0; mx<Mx_; ++mx)
1493         delete[] tausolver_[j][mx];  // undo new #3
1494       delete[] tausolver_[j];        // undo new #2
1495     }
1496     delete[] tausolver_;                 // undo new #1
1497   }
1498   tausolver_ = 0;
1499 }
```

7.22.4 Member Function Documentation

7.22.4.1 void RungeKuttaDNS::advance (FlowField & *u*, FlowField & *q*, int *nSteps* = 1) [virtual]

Implements DNSAlgorithm.

References A_, ComplexChebyCoeff::add(), B_, C_, DNSAlgorithm::cfl_, FlowField::CFLfactor(), FlowField::cmplx(), DNSFlags::constraint, DNSFlags::dealias_xz(), diff(), diff2(), DNSAlgorithm::dPdxAct_, DNSAlgorithm::dPdxRef_, DNSAlgorithm::dt_, FlowField::Dx(), FlowField::Dz(), DNSAlgorithm::flags_, DNSAlgorithm::isAliasedMode(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), Vector::length(), DNSAlgorithm::Lx_, DNSAlgorithm::Lz_, ChebyCoeff::mean(), DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, navierstokesNL(), DNSFlags::nonlinearity, Nsubsteps_, DNSAlgorithm::nu_, DNSAlgorithm::Nx_, DNSAlgorithm::Ny_, DNSAlgorithm::Nyd_, DNSAlgorithm::Nz_, pi, DNSAlgorithm::Pk_, PressureGradient, PrintAll, PrintTicks, PrintTime, DNSAlgorithm::Pyk_, Qj1_, Qj_, Re(), ComplexChebyCoeff::re, DNSAlgorithm::Rxk_, DNSAlgorithm::Ryk_, DNSAlgorithm::Rzk_, ComplexChebyCoeff::set(), FlowField::setDealiased(), ComplexChebyCoeff::setToZero(), TauSolver::solve(), square(), DNSAlgorithm::t_, tausolver_, DNSAlgorithm::tmp_, DNSAlgorithm::Ubase_, DNSAlgorithm::Ubaseyy_, DNSAlgorithm::UbulkAct_, DNSAlgorithm::UbulkBase_, DNSAlgorithm::UbulkRef_, DNSAlgorithm::uk_, DNSFlags::verbosity, DNSAlgorithm::vk_, and DNSAlgorithm::wk_.

```
1533                                                  {
1534
1535   const int kxmax = un.kxmax();
1536   const int kzmax = un.kzmax();
1537
1538   for (int n=0; n<Nsteps; ++n) {
```

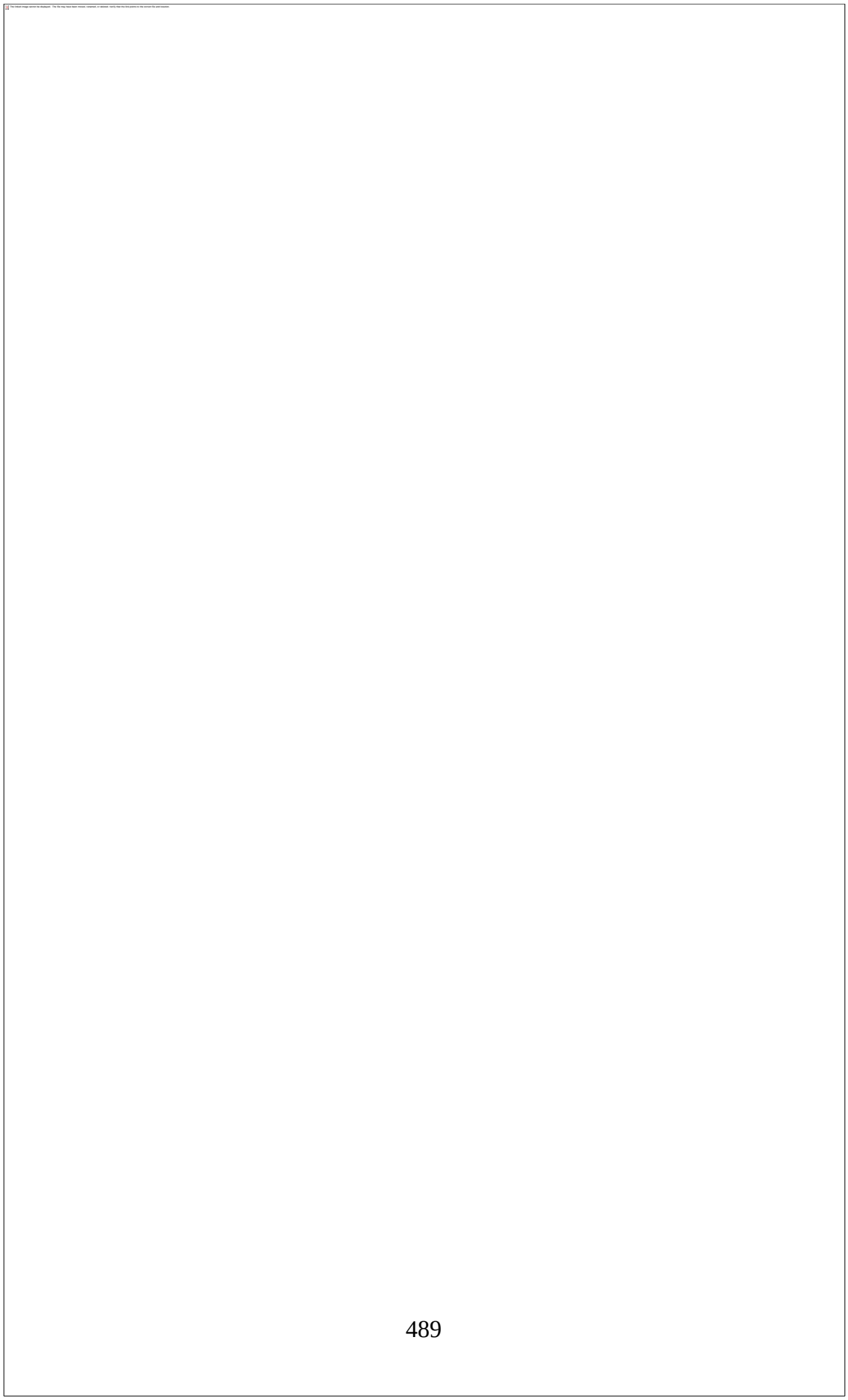
```
1539    for (int j=0; j<Nsubsteps_; ++j) {
1540
1541      FlowField& uj(un); // Store uj in un during substeps, reflect in notation
1542
1543      // Efficient implementation of
1544      // Q_{j+1} = A_j Q_j + N(u_j)}  where N = -u grad u
1545      // Q_{j+1} = A_j Q_j - f(u_j)}  where f =  u grad u
1546      // Q_j = Q_{j+1}
1547      Qj_ *= A_[j];
1548       //cout << "IG call navierstokesNL 4" <<endl;
1549
1550      navierstokesNL(uj, Ubase_, Qj1_, tmp_, flags_.nonlinearity);
1551      Qj_ -= Qj1_; // subtract because navierstokesNL(u) = u grad u = -N(u)
1552
1553      // Update each Fourier mode with time-stepping algorithm
1554      for (int mx=0; mx<Mx_; ++mx) {
1555        const int kx = un.kx(mx);
1556
1557        for (int mz=0; mz<Mz_; ++mz) {
1558          const int kz = un.kz(mz);
1559
1560          // Zero out the aliased modes
1561          if ((kx == kxmax || kz == kzmax) ||
1562              flags_.dealias_xz() && isAliasedMode(kx,kz)) {
1563            for (int ny=0; ny<Nyd_; ++ny) {
1564              un.cmplx(mx,ny,mz,0) = 0.0;
1565              un.cmplx(mx,ny,mz,1) = 0.0;
1566              un.cmplx(mx,ny,mz,2) = 0.0;
1567              qn.cmplx(mx,ny,mz,0) = 0.0;
1568            }
1569            break;
1570          }
1571
1572          Rxk_.setToZero();
1573          Ryk_.setToZero();
1574          Rzk_.setToZero();
1575
1576          // Make the following assignments in prep for computation of RHS
1577
1578          // nu (uk,vk,wk) = nu ujn(0,1,2)
1579          //           Pk = qn
1580
1581          // Goal is to compute
1582          // R = nu uj" + [1/(Cj dt) - nu kappa2]    uj  - grad qj + Bj/Cj Qj
1583          //   = nu uj" + [1/(nu Cj dt) - kappa2] nu uj  - grad qj + Bj/Cj Qj
1584
```

```
1585        // Extract relevant Fourier modes of uj and qj for computations
1586        for (int ny=0; ny<Nyd_; ++ny) {
1587          uk_.set(ny, nu_*uj.cmplx(mx,ny,mz,0));
1588          vk_.set(ny, nu_*uj.cmplx(mx,ny,mz,1));
1589          wk_.set(ny, nu_*uj.cmplx(mx,ny,mz,2));
1590          Pk_.set(ny, qn.cmplx(mx,ny,mz,0));
1591        }
1592
1593        // (1) Put nu uj" into in R. (Pyk_ is used as tmp workspace)
1594        diff2(uk_, Rxk_, Pyk_);
1595        diff2(vk_, Ryk_, Pyk_);
1596        diff2(wk_, Rzk_, Pyk_);
1597
1598        // (2) Put qn' into Pyk (compute y-comp of pressure gradient).
1599        diff(Pk_, Pyk_);
1600
1601        // (3) Add [1/(nu Cj dt)- kappa2] nu uj - grad qj + Bj/Cj Qj to R.
1602        const Real c =
1603          1.0/(nu_*C_[j]*dt_) - 4*pi*pi*(square(kx/Lx_) + square(kz/Lz_));
1604        const Real B_C = B_[j]/C_[j];
1605        const Complex Dx = un.Dx(mx);
1606        const Complex Dz = un.Dz(mz);
1607
1608        for (int ny=0; ny<Nyd_; ++ny) {
1609          Rxk_.add(ny, c*uk_[ny] + B_C*Qj_.cmplx(mx,ny,mz,0) - Dx*Pk_[ny]);
1610          Ryk_.add(ny, c*vk_[ny] + B_C*Qj_.cmplx(mx,ny,mz,1) - Pyk_[ny]);
1611          Rzk_.add(ny, c*wk_[ny] + B_C*Qj_.cmplx(mx,ny,mz,2) - Dz*Pk_[ny]);
1612        }
1613
1614        // Do the tau solutions
1615        if (kx!=0 || kz!=0)
1616          tausolver_[j][mx][mz].solve(uk_,vk_,wk_,Pk_, Rxk_,Ryk_,Rzk_);
1617
1618        else { // kx,kz == 0,0
1619          // Rx has additional terms, nu Uyy at both t=j and t=j+1
1620          const Real c = 2*nu_;
1621          if (Ubaseyy_.length() > 0)
1622            for (int ny=0; ny<Ny_; ++ny)
1623              Rxk_.re[ny] += c*Ubaseyy_[ny];
1624
1625          if (flags_.constraint == PressureGradient) {
1626            // dPdx is supplied, put dPdx at both t=j and t=j+1 on RHS
1627            Rxk_.re[0] -= dPdxAct_ + dPdxRef_;
1628
1629            // Solve the tau equations
1630            tausolver_[j][mx][mz].solve(uk_, vk_, wk_, Pk_, Rxk_, Ryk_, Rzk_);
```

```
1631
1632            // Bulk velocity is free variable on LHS solved by tau eqn
1633            UbulkAct_ = UbulkBase_ + uk_.re.mean();
1634            dPdxAct_ = dPdxRef_;
1635          }
1636          else { // const bulk velocity
1637            // Add the previous time-step's -dPdx to the RHS. The next
1638            // timestep's dPdx term appears on LHS as unknown.
1639            Rxk_.re[0] -= dPdxAct_;
1640
1641            // Use tausolver with additional variable and constraint:
1642            // free variable: dPdxAct at next time-step,
1643            // constraint:    UbulkBase + mean(u) = UbulkRef.
1644            tausolver_[j][mx][mz].solve(uk_, vk_, wk_, Pk_, dPdxAct_,
1645                                        Rxk_, Ryk_, Rzk_,
1646                                        UbulkRef_ - UbulkBase_);
1647            UbulkAct_ = UbulkBase_ + uk_.re.mean(); // should == UbulkRef_
1648          }
1649          // for kx=kz=0, constant term of pressure is arbitrary 3/19/05
1650          // Pk_.set(0, Complex(0.0, 0.0));
1651        }
1652
1653        // Load solutions back into the external 3d data arrays.
1654        // Because of FFTW complex symmetries
1655        // The 0,0 mode must be real.
1656        // For Nx even, the kxmax,0 mode must be real
1657        // For Nz even, the 0,kzmax mode must be real
1658        // For Nx,Nz even, the kxmax,kzmax mode must be real
1659        if ((kx == 0 && kz == 0) ||
1660            (Nx_%2 == 0 && kx == kxmax && kz == 0) ||
1661            (Nz_%2 == 0 && kz == kzmax && kx == 0) ||
1662            (Nx_%2 == 0 && Nz_%2 == 0 && kx == kxmax && kz == kzmax)) {
1663
1664          for (int ny=0; ny<Nyd_; ++ny) {
1665            un.cmplx(mx,ny,mz,0) = Complex(Re(uk_[ny]), 0.0);
1666            un.cmplx(mx,ny,mz,1) = Complex(Re(vk_[ny]), 0.0);
1667            un.cmplx(mx,ny,mz,2) = Complex(Re(wk_[ny]), 0.0);
1668            qn.cmplx(mx,ny,mz,0) = Complex(Re(Pk_[ny]), 0.0);
1669          }
1670        }
1671        // The normal case, for general kx,kz
1672        else
1673          for (int ny=0; ny<Nyd_; ++ny) {
1674            un.cmplx(mx,ny,mz,0) = uk_[ny];
1675            un.cmplx(mx,ny,mz,1) = vk_[ny];
1676            un.cmplx(mx,ny,mz,2) = wk_[ny];
```

```
1677             qn.cmplx(mx,ny,mz,0) = Pk_[ny];
1678           }
1679
1680         // And now set the y-aliased modes to zero.
1681         for (int ny=Nyd_; ny<Ny_; ++ny) {
1682           un.cmplx(mx,ny,mz,0) = 0.0;
1683           un.cmplx(mx,ny,mz,1) = 0.0;
1684           un.cmplx(mx,ny,mz,2) = 0.0;
1685           qn.cmplx(mx,ny,mz,0) = 0.0;
1686         }
1687       }
1688     }
1689   }
1690   t_ += dt_;
1691
1692   if (flags_.verbosity == PrintTime || flags_.verbosity == PrintAll)
1693     cout << t_ << ' ' << flush;
1694   else if (flags_.verbosity == PrintTicks)
1695     cout << '.' << flush;
1696 }
1697
1698 cfl_ = un.CFLfactor(Ubase_);
1699 cfl_ *= flags_.dealias_xz() ? 2.0*pi/3.0*dt_ : pi*dt_;
1700
1701 // If using dealiasing, set flag in FlowField that compactifies binary IO
1702 un.setDealiased(flags_.dealias_xz());
1703 qn.setDealiased(flags_.dealias_xz());
1704
1705 if (flags_.verbosity == PrintTime || flags_.verbosity == PrintAll)
1706   cout << endl;
1707
1708 return;
1709 }
```

Here is the call graph for this function:



7.22.4.2 DNSAlgorithm * RungeKuttaDNS::clone () const [virtual]

Implements DNSAlgorithm.

References RungeKuttaDNS().

```
1501                                    {
1502   return new RungeKuttaDNS(*this);
1503 }
```

Here is the call graph for this function:

7.22.4.3 RungeKuttaDNS& RungeKuttaDNS::operator= (const RungeKuttaDNS & *dns*)

Reimplemented from DNSAlgorithm.

7.22.4.4 void RungeKuttaDNS::reset_dt (Real *dt*) [virtual]

Implements DNSAlgorithm.

References DNSAlgorithm::a_, DNSAlgorithm::b_, C_, DNSAlgorithm::cfl_, DNSFlags::dealias_xz(), DNSAlgorithm::dt_, DNSAlgorithm::flags_, DNSAlgorithm::isAliasedMode(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), DNSAlgorithm::Lx_, DNSAlgorithm::Lz_, DNSAlgorithm::Mx_, DNSAlgorithm::Mz_, Nsubsteps_, DNSAlgorithm::nu_, DNSAlgorithm::Nyd_, pi, square(), DNSFlags::taucorrection, tausolver_, and DNSAlgorithm::tmp_.

Referenced by RungeKuttaDNS().

```
1505                                    {
1506   cfl_ *= dt/dt_;
1507   //nu_ = nu;
1508   dt_ = dt;
1509   const Real c = 4.0*square(pi)*nu_;
1510   const int kxmax = tmp_.kxmax();
1511   const int kzmax = tmp_.kzmax();
1512
1513   // This loop replaces the TauSolver objects at tausolver_[i][mx][mz]
1514   // with new TauSolver objects configured with appropriate parameters
1515   for (int j=0; j<Nsubsteps_; ++j) {
1516     for (int mx=0; mx<Mx_; ++mx) {
1517       int kx = tmp_.kx(mx);
1518       for (int mz=0; mz<Mz_; ++mz) {
1519         int kz = tmp_.kz(mz);
```

```
1520       Real lambda = 1.0/(C_[j]*dt_) + c*(square(kx/Lx_)+square(kz/Lz_));
1521
1522       if ((kx != kxmax || kz != kzmax) &&
1523          (!flags_.dealias_xz() || !isAliasedMode(kx,kz)))
1524
1525         tausolver_[j][mx][mz] =
1526           TauSolver(kx, kz, Lx_, Lz_, a_, b_, lambda, nu_, Nyd_,
1527                     flags_.taucorrection);
1528       }
1529     }
1530   }
1531 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.22.5 Member Data Documentation

7.22.5.1 array<Real> RungeKuttaDNS::A_ [protected]

Referenced by advance(), and RungeKuttaDNS().

7.22.5.2 array<Real> RungeKuttaDNS::B_ [protected]

Referenced by advance(), and RungeKuttaDNS().

7.22.5.3 array<Real> RungeKuttaDNS::C_ [protected]

Referenced by advance(), reset_dt(), and RungeKuttaDNS().

7.22.5.4 int RungeKuttaDNS::Nsubsteps_ [protected]

Referenced by advance(), reset_dt(), RungeKuttaDNS(), and ~RungeKuttaDNS().

7.22.5.5 FlowField RungeKuttaDNS::Qj1_ [protected]

Referenced by advance(), and RungeKuttaDNS().

7.22.5.6 FlowField RungeKuttaDNS::Qj_ [protected]

Referenced by advance(), and RungeKuttaDNS().

7.22.5.7 TauSolver*** RungeKuttaDNS::tausolver_ [protected]

Referenced by advance(), reset_dt(), RungeKuttaDNS(), and ~RungeKuttaDNS().

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

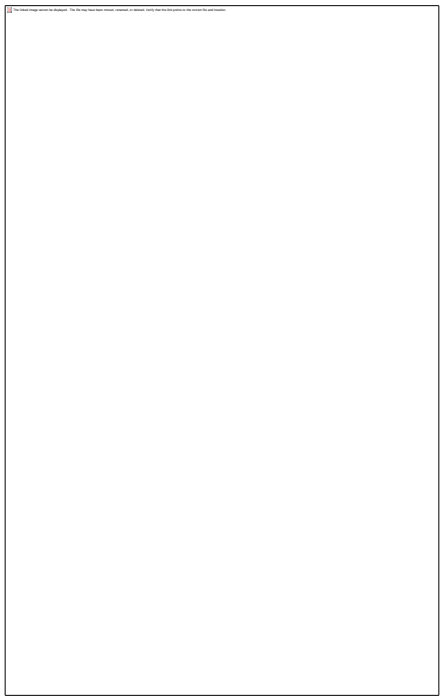
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.22.5.9

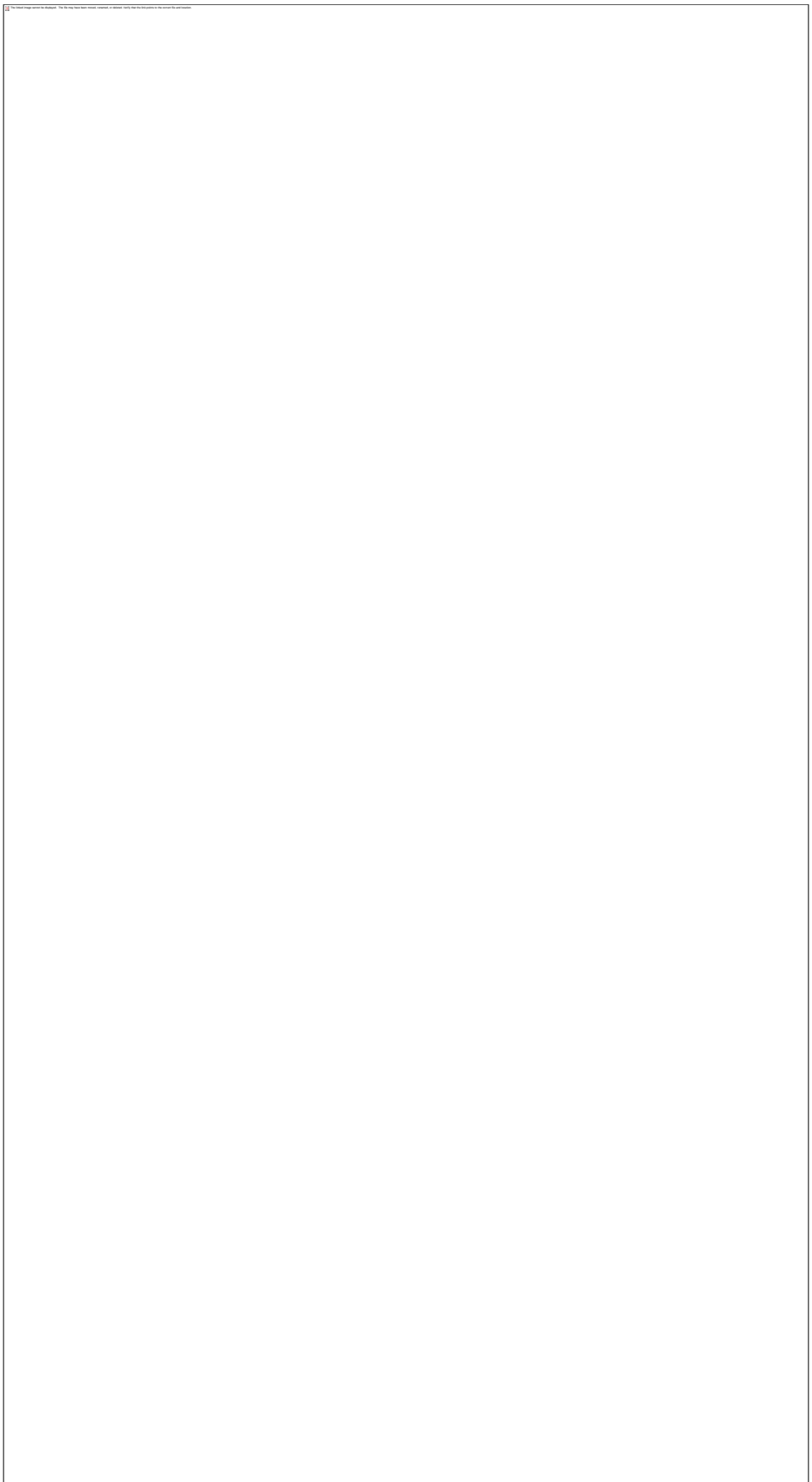
7.23 skewsymmetricNL_task Class Reference

#include <dns.h>

Inheritance diagram for skewsymmetricNL_task:



Collaboration diagram for skewsymmetricNL_task:



7.23.1 Public Member Functions

- skewsymmetricNL_task (const FlowField &_u, FlowField &_f, FlowField &_grad_u)
- void Run (int tn, int nThreads)
- void setStep (int step)
- void setAttributes (Attributes &attr)

7.23.2 Public Attributes

- pthread_mutex_t mut

7.23.3 Private Attributes

- const FlowField & u
- FlowField & f
- FlowField & grad_u
- int m_step
- Attributes m_attr

7.23.4 Constructor & Destructor Documentation

7.23.4.1 skewsymmetricNL_task::skewsymmetricNL_task (const FlowField & *_u*, FlowField & *_f*, FlowField & *_grad_u*)

```
187                                                 :
188                                     u(_u),
189                                     f(_f),
190                                     grad_u(_grad_u)
191
192 {
193   // pthread_mutex_init(&mut,NULL);
194 }
```

7.23.5 Member Function Documentation

7.23.5.1 void skewsymmetricNL_task::Run (int *tn*, int *nThreads*) [virtual]

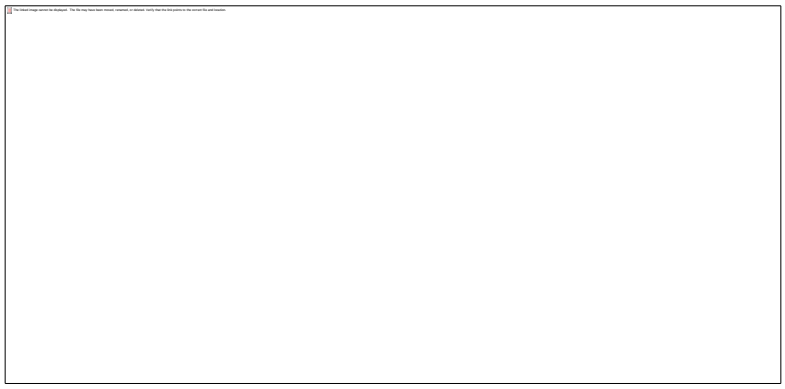
Reimplemented from BaseTask.

References f, grad_u, i3j(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), and u.

```
207 {
208  //cout << "Igor test skewsymmetricNL_step1 Run running " << endl;
209
```

```
210   int Ny = u.Ny()/nThreads;
211   int Nx = u.Nx();
212   int Nz = u.Nz();
213
214   int sty = Ny*tn;
215   int edy = Ny*(tn+1);
216
217   for (int i=0; i<3; ++i)
218     for (int ny=sty; ny < edy; ++ny)
219       for (int nx=0; nx < Nx; ++nx)
220         for (int nz=0; nz < Nz; ++nz)
221           for(int j=0; j <3; j++)
222           {
223              int ij = i3j(i,j);
224              f(nx,ny,nz,i) += 0.5*u(nx,ny,nz,j)*grad_u(nx,ny,nz,ij);
225           }
226
227
228 }
```

Here is the call graph for this function:



7.23.5.2 void skewsymmetricNL_task::setAttributes (Attributes & *attr*)

References m_attr.

```
202 {
203   m_attr = attr;
204 }
```

7.23.5.3 void skewsymmetricNL_task::setStep (int *step*)

References m_step.

```
197 {
198   m_step = step;
199 }
```

7.23.6 Member Data Documentation

7.23.6.1 FlowField& skewsymmetricNL_task::f [private]

Referenced by Run().

7.23.6.2 FlowField& skewsymmetricNL_task::grad_u [private]

Referenced by Run().

7.23.6.3 Attributes skewsymmetricNL_task::m_attr [private]

Referenced by setAttributes().

7.23.6.4 int skewsymmetricNL_task::m_step [private]

Referenced by setStep().

7.23.6.5 pthread_mutex_t skewsymmetricNL_task::mut

7.23.6.6 const FlowField& skewsymmetricNL_task::u [private]

Referenced by Run().

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

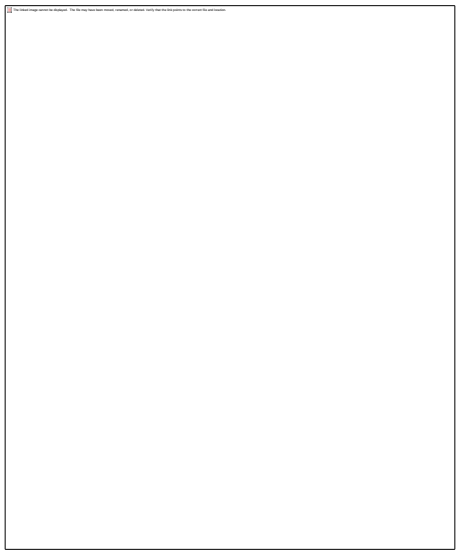
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.23.6.8

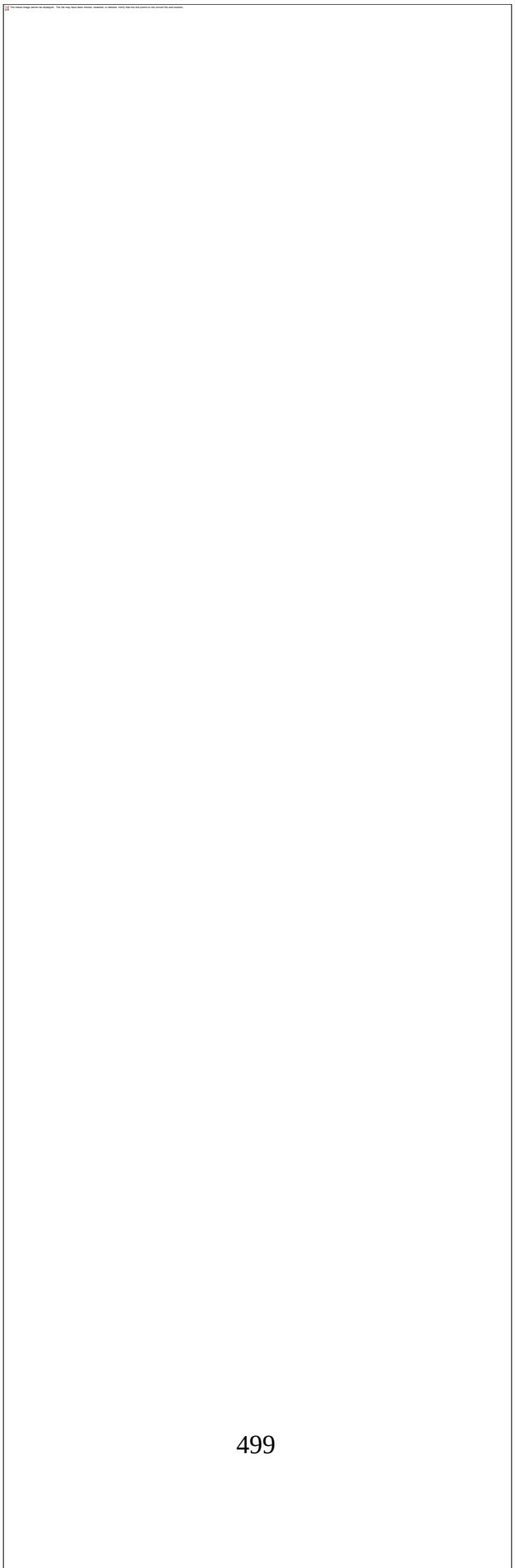
7.24 skewsymmetricNL_task2 Class Reference

#include <dns.h>

Inheritance diagram for skewsymmetricNL_task2:



Collaboration diagram for skewsymmetricNL_task2:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.24.1 Public Member Functions

- skewsymmetricNL_task2 (const FlowField &_u, FlowField &_uu, FlowField &_tmp)
- void Run (int tn, int nThreads)

7.24.2 Private Attributes

- const FlowField & u
- FlowField & uu
- FlowField & tmp

7.24.3 Constructor & Destructor Documentation

7.24.3.1 skewsymmetricNL_task2::skewsymmetricNL_task2 (const FlowField & *_u*, FlowField & *_uu*, FlowField & *_tmp*)

```
236                                     :u(_u),uu(_uu),tmp(_tmp)
237 {
238 }
```

7.24.4 Member Function Documentation

7.24.4.1 void skewsymmetricNL_task2::Run (int *tn*, int *nThreads*) [virtual]

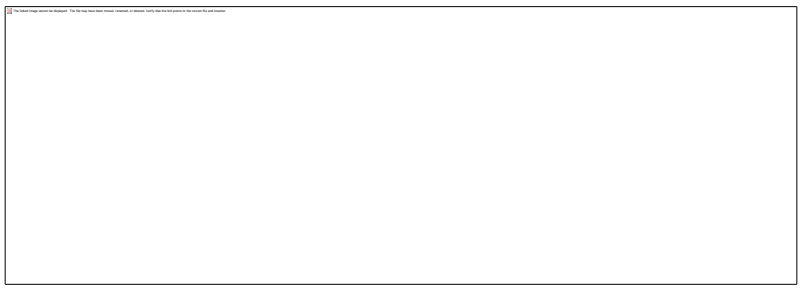
Reimplemented from BaseTask.

References FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), tmp, u, and uu.

```
240 {
241  //cout << "Igor test skewsymmetricNL_task2 Run running " << endl;
242
243   int Nx = u.Nx()/nThreads;
244   int Ny = u.Ny();
245   int Nz = u.Nz();
246
247   int stx = Nx*tn;
248   int edx = Nx*(tn+1);
249
250
251   for (int nx=stx; nx<edx; ++nx) {
252    for (int ny=0; ny<Ny; ++ny)
253     for (int nz=0; nz<Nz; ++nz) {
254       Real u0 = u(nx,ny,nz,0);
255       Real u1 = u(nx,ny,nz,1);
```

```
256         Real u2 = u(nx,ny,nz,2);
257         uu(nx,ny,nz,0) = u0*u0;
258         uu(nx,ny,nz,1) = tmp(nx,ny,nz,3) = u0*u1;
259         uu(nx,ny,nz,2) = tmp(nx,ny,nz,6) = u0*u2;
260         uu(nx,ny,nz,4) = u1*u1;
261         uu(nx,ny,nz,5) = tmp(nx,ny,nz,7) = u1*u2;
262         uu(nx,ny,nz,8) = u2*u2;
263       }
264   }
265 }
```

Here is the call graph for this function:



7.24.5 Member Data Documentation

7.24.5.1 FlowField& skewsymmetricNL_task2::tmp [private]

Referenced by Run().

7.24.5.2 const FlowField& skewsymmetricNL_task2::u [private]

Referenced by Run().

7.24.5.3 FlowField& skewsymmetricNL_task2::uu [private]

Referenced by Run().

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

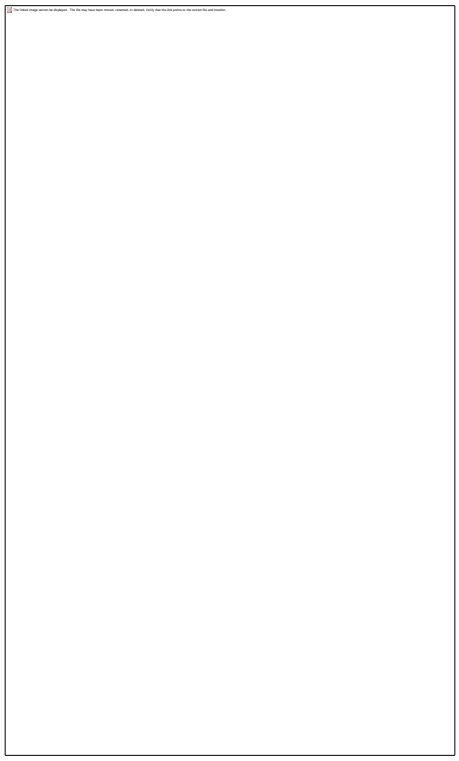
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.24.5.5

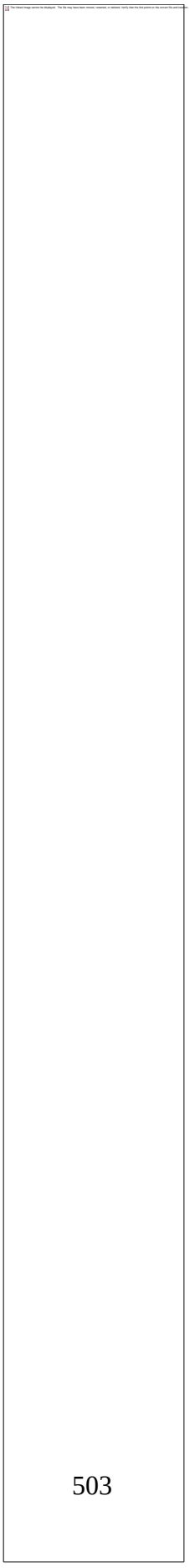
7.25 skewsymmetricNL_task3 Class Reference

#include <dns.h>

Inheritance diagram for skewsymmetricNL_task3:



Collaboration diagram for skewsymmetricNL_task3:


The inked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.25.1 Public Member Functions

- skewsymmetricNL_task3 (const FlowField &_u, FlowField &_uu, FlowField &_f, int _kxmax, int _kzmax, Real _Lx, Real _Lz, int _Mx, int _My, int _Mz)
- void Run (int tn, int nThreads)

7.25.2 Private Attributes

- const FlowField & u
- FlowField & uu
- FlowField & f
- int kxmax
- int kzmax
- Real Lx
- Real Lz
- int Mx
- int My
- int Mz

7.25.3 Constructor & Destructor Documentation

7.25.3.1 skewsymmetricNL_task3::skewsymmetricNL_task3 (const FlowField & *_u*, FlowField & *_uu*, FlowField & *_f*, int *_kxmax*, int *_kzmax*, Real *_Lx*, Real *_Lz*, int *_Mx*, int *_My*, int *_Mz*)

References kxmax, kzmax, Lx, Lz, Mx, My, and Mz.

```
269                                                                    :
270                                 u(_u),
271                                 uu(_uu),
272                                 f(_f)
273 {
274  kxmax=_kxmax;
275  kzmax=_kzmax;
276  Lx=_Lx;
277  Lz=_Lz;
278  Mx=_Mx;
279  My=_My;
280  Mz=_Mz;
281 }
```

7.25.4 Member Function Documentation

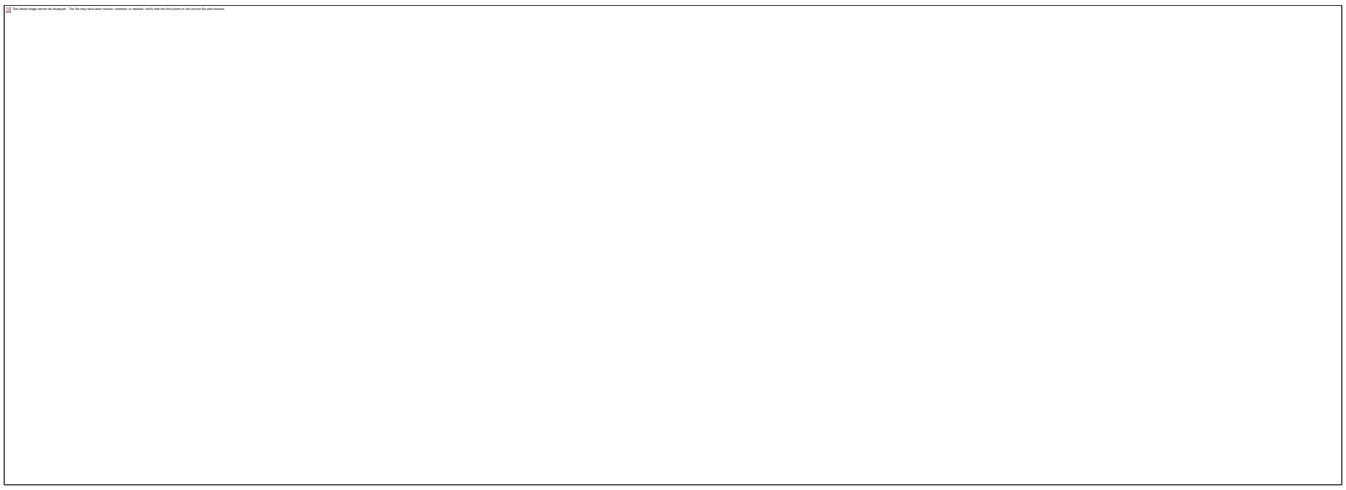
7.25.4.1 void skewsymmetricNL_task3::Run (int *tn*, int *nThreads*) [virtual]

Reimplemented from BaseTask.

References FlowField::cmplx(), f, i3j(), FlowField::kx(), kxmax, FlowField::kz(), kzmax, Lx, Lz, Mx, My, Mz, pi, u, uu, and zero_last_mode().

```
284 {
285
286   int sty = My*tn/nThreads;
287   int edy = My*(tn+1)/nThreads;
288
289   int stx = Mx*tn/nThreads;
290   int edx = Mx*(tn+1)/nThreads;
291
292  for (int i=0; i<3; ++i) {
293    int i0 = i3j(i,0);
294    int i2 = i3j(i,2);
295  // Add in du_i/dx and du_i/dz, that is, d/dx_j (u_i u_j) for j=0,2
296   for (int mx=stx; mx<edx; ++mx)
297     for (int my=0; my<My; ++my) {
298       int kx = u.kx(mx);
299       Complex d_dx(0.0, 2*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
300       for (int mz=0; mz<Mz; ++mz) {
301         int kz = u.kz(mz);
302         Complex d_dz(0.0, 2*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
303         f.cmplx(mx,my,mz,i)
304           += 0.5*(d_dx*uu.cmplx(mx,my,mz,i0)+d_dz*uu.cmplx(mx,my,mz,i2));
305       }
306     }
307   }
308 }
```

Here is the call graph for this function:



7.25.5 Member Data Documentation

7.25.5.1 FlowField& skewsymmetricNL_task3::f [private]

Referenced by Run().

7.25.5.2 int skewsymmetricNL_task3::kxmax [private]

Referenced by Run(), and skewsymmetricNL_task3().

7.25.5.3 int skewsymmetricNL_task3::kzmax [private]

Referenced by Run(), and skewsymmetricNL_task3().

7.25.5.4 Real skewsymmetricNL_task3::Lx [private]

Referenced by Run(), and skewsymmetricNL_task3().

7.25.5.5 Real skewsymmetricNL_task3::Lz [private]

Referenced by Run(), and skewsymmetricNL_task3().

7.25.5.6 int skewsymmetricNL_task3::Mx [private]

Referenced by Run(), and skewsymmetricNL_task3().

7.25.5.7 int skewsymmetricNL_task3::My [private]

Referenced by Run(), and skewsymmetricNL_task3().

7.25.5.8 int skewsymmetricNL_task3::Mz [private]

Referenced by Run(), and skewsymmetricNL_task3().

7.25.5.9 const FlowField& skewsymmetricNL_task3::u [private]

Referenced by Run().

7.25.5.10 FlowField& skewsymmetricNL_task3::uu [private]

Referenced by Run().

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

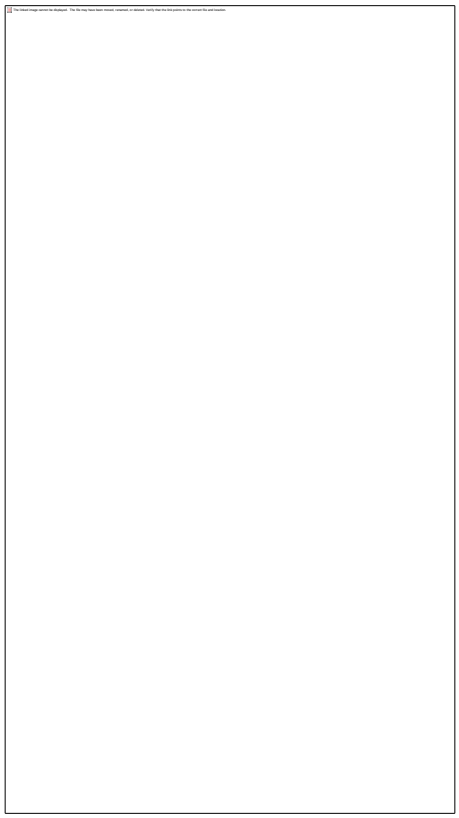
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.25.5.12

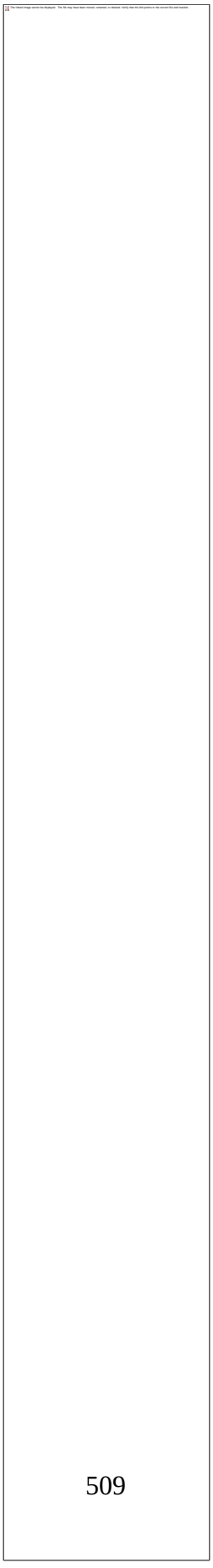
7.26 skewsymmetricNL_task4 Class Reference

#include <dns.h>

Inheritance diagram for skewsymmetricNL_task4:



Collaboration diagram for skewsymmetricNL_task4:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.26.1 Public Member Functions

- skewsymmetricNL_task4 (const FlowField &_u, FlowField &_uu, FlowField &_f, ComplexChebyCoeff &_tmpProfile, ComplexChebyCoeff &_tmpProfile_y, int _kxmax, int _kzmax, Real _Lx, Real _Lz, int _Mx, int _My, int _Mz)
- void Run (int tn, int nThreads)

7.26.2 Private Attributes

- const FlowField & u
- FlowField & uu
- FlowField & f
- ComplexChebyCoeff tmpProfile
- ComplexChebyCoeff tmpProfile_y
- int kxmax
- int kzmax
- Real Lx
- Real Lz
- int Mx
- int My
- int Mz

7.26.3 Constructor & Destructor Documentation

7.26.3.1 skewsymmetricNL_task4::skewsymmetricNL_task4 (const FlowField & *_u*, FlowField & *_uu*, FlowField & *_f*, ComplexChebyCoeff & *_tmpProfile*, ComplexChebyCoeff & *_tmpProfile_y*, int *_kxmax*, int *_kzmax*, Real *_Lx*, Real *_Lz*, int *_Mx*, int *_My*, int *_Mz*)

References kxmax, kzmax, Lx, Lz, Mx, My, and Mz.

```
313                                                                 :
314                                 u(_u),
315                                 uu(_uu),
316                                 f(_f),
317                                 tmpProfile(_tmpProfile),
318                                 tmpProfile_y(_tmpProfile_y)
319 {
320   kxmax=_kxmax;
321   kzmax=_kzmax;
322   Lx=_Lx;
323   Lz=_Lz;
324   Mx=_Mx;
325   My=_My;
```

```
326  Mz=_Mz;
327 }
```

7.26.4 Member Function Documentation

7.26.4.1 void skewsymmetricNL_task4::Run (int *tn*, int *nThreads*) [virtual]

Reimplemented from BaseTask.

References FlowField::cmplx(), diff(), f, i3j(), Mx, My, Mz, ComplexChebyCoeff::set(), tmpProfile, tmpProfile_y, and uu.

```
330 {
331    int stx = Mx*tn/nThreads;
332    int edx = Mx*(tn+1)/nThreads;
333
334  for (int i=0; i<3; ++i) {
335     int i1 = i3j(i,1);
336  // Add in du_i/dy, that is d/dx_j (u_i u_j) for j=1
337   for (int mx=stx; mx<edx; ++mx)
338      for (int mz=0; mz<Mz; ++mz) {
339        for (int my=0; my<My; ++my)
340          tmpProfile.set(my, uu.cmplx(mx,my,mz,i1));
341        diff(tmpProfile, tmpProfile_y);
342        for (int my=0; my<My; ++my)
343          f.cmplx(mx,my,mz,i) += 0.5*tmpProfile_y[my]; // j=1
344      }
345   }
346 }
```

Here is the call graph for this function:

7.26.5 Member Data Documentation

7.26.5.1 FlowField& skewsymmetricNL_task4::f [private]

Referenced by Run().

7.26.5.2 int skewsymmetricNL_task4::kxmax [private]

Referenced by skewsymmetricNL_task4().

7.26.5.3 int skewsymmetricNL_task4::kzmax [private]

Referenced by skewsymmetricNL_task4().

7.26.5.4 Real skewsymmetricNL_task4::Lx [private]

Referenced by skewsymmetricNL_task4().

7.26.5.5 Real skewsymmetricNL_task4::Lz [private]

Referenced by skewsymmetricNL_task4().

7.26.5.6 int skewsymmetricNL_task4::Mx [private]

Referenced by Run(), and skewsymmetricNL_task4().

7.26.5.7 int skewsymmetricNL_task4::My [private]

Referenced by Run(), and skewsymmetricNL_task4().

7.26.5.8 int skewsymmetricNL_task4::Mz [private]

Referenced by Run(), and skewsymmetricNL_task4().

7.26.5.9 ComplexChebyCoeff skewsymmetricNL_task4::tmpProfile [private]

Referenced by Run().

7.26.5.10 ComplexChebyCoeff skewsymmetricNL_task4::tmpProfile_y [private]

Referenced by Run().

7.26.5.11 const FlowField& skewsymmetricNL_task4::u [private]

7.26.5.12 FlowField& skewsymmetricNL_task4::uu [private]

Referenced by Run().

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

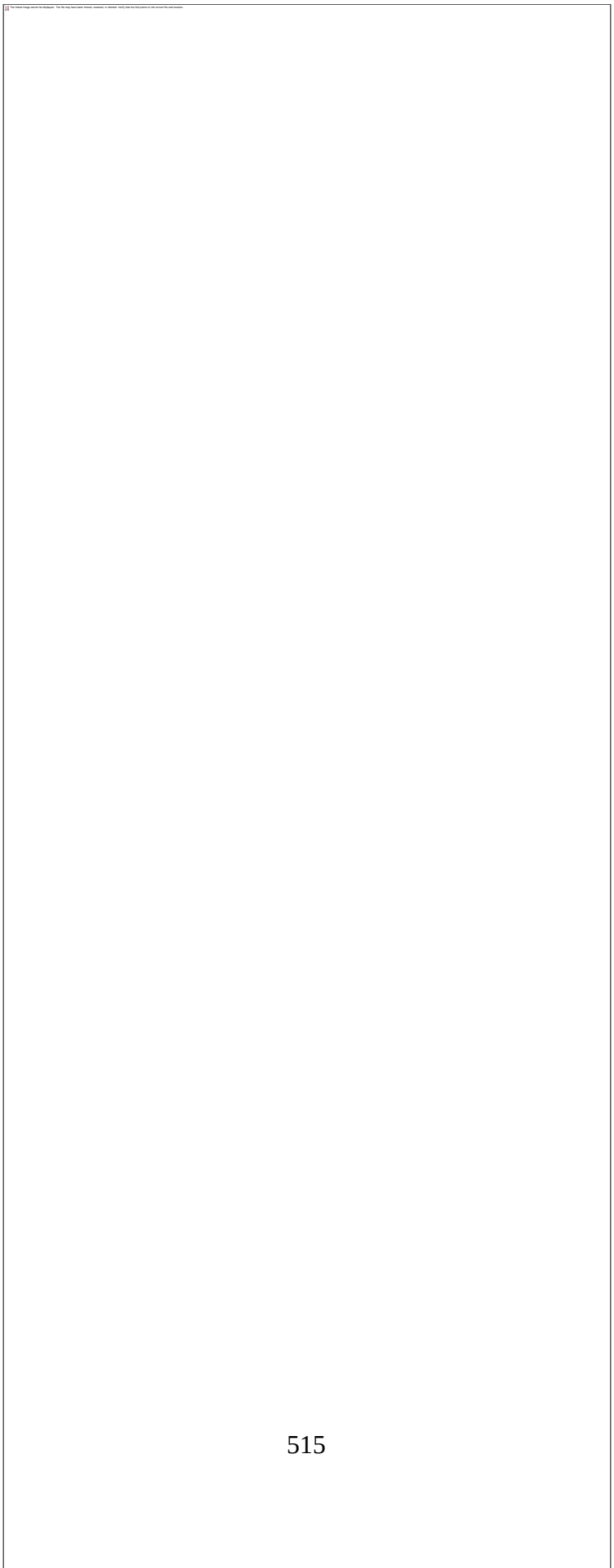
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.26.5.14

7.27 TauSolver Class Reference

#include <tausolver.h>

Collaboration diagram for TauSolver:


The inkwell image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.27.1 Public Member Functions

- TauSolver ()
- TauSolver (int kx, int kz, Real Lx, Real Lz, Real a, Real b, Real lambda, Real nu, int nChebyModes, bool tauCorrection=true)
- void solve (ComplexChebyCoeff &u, ComplexChebyCoeff &v, ComplexChebyCoeff &w, ComplexChebyCoeff &P, const ComplexChebyCoeff &Rx, const ComplexChebyCoeff &Ry, const ComplexChebyCoeff &Rz) const
- Real verify (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, const ComplexChebyCoeff &w, const ComplexChebyCoeff &P, const ComplexChebyCoeff &Rx, const ComplexChebyCoeff &Ry, const ComplexChebyCoeff &Rz, bool verbose=false) const
- void solve (ComplexChebyCoeff &u, ComplexChebyCoeff &v, ComplexChebyCoeff &w, ComplexChebyCoeff &P, Real &dPdx, const ComplexChebyCoeff &Rx, const ComplexChebyCoeff &Ry, const ComplexChebyCoeff &Rz, Real umean) const
- Real verify (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, const ComplexChebyCoeff &w, const ComplexChebyCoeff &P, Real dPdx, const ComplexChebyCoeff &Rx, const ComplexChebyCoeff &Ry, const ComplexChebyCoeff &Rz, Real umean, bool verbose=false) const
- void influenceCorrection (ChebyCoeff &P, ChebyCoeff &v) const
- void solve_P_and_v (ChebyCoeff &P, ChebyCoeff &v, const ChebyCoeff &r, const ChebyCoeff &Ry, Real &sigmaNb1, Real &sigmaNb) const
- Real verify_P_and_v (const ChebyCoeff &P, const ChebyCoeff &v, const ChebyCoeff &r, const ChebyCoeff &Ry, Real sigmaNb1, Real sigmaNb, bool verbose=false) const
- int kx () const
- int kz () const
- Real lambda () const
- Real nu () const

7.27.2 Private Member Functions

- Real tauNorm (const ChebyCoeff &u) const
- Real tauNorm (const ComplexChebyCoeff &u) const
- Real tauDist (const ChebyCoeff &u, const ChebyCoeff &v) const
- Real tauDist (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v) const

7.27.3 Private Attributes

- int N_
- int Nb_
- int kx_
- int kz_
- Real two_pi_kxLx_
- Real two_pi_kzLz_
- Real kappa2_

- Real a_
- Real b_
- Real lambda_
- Real nu_
- bool tauCorrection_
- HelmholtzSolver pressureHelmholtz_
- HelmholtzSolver velocityHelmholtz_
- ChebyCoeff P_0_
- ChebyCoeff v_0_
- ChebyCoeff P_plus_
- ChebyCoeff v_plus_
- ChebyCoeff P_minus_
- ChebyCoeff v_minus_
- Real i00_
- Real i01_
- Real i10_
- Real i11_
- Real sigma0_Nb1_
- Real sigma0_Nb_

7.27.4 Constructor & Destructor Documentation

7.27.4.1 TauSolver::TauSolver ()

```
78  :
79  N_(0),
80  Nb_(0),
81  kx_(0),
82  kz_(0),
83  two_pi_kxLx_(0),
84  two_pi_kzLz_(0),
85  kappa2_(0),
86  a_(0),
87  b_(0),
88  lambda_(0),
89  nu_(0),
90  tauCorrection_(true),
91  pressureHelmholtz_(),
92  velocityHelmholtz_(),
93  P_0_(),
94  v_0_(),
95  P_plus_(),
96  v_plus_(),
97  P_minus_(),
```

```
98   v_minus_(),
99   i00_(0),
100   i01_(0),
101   i10_(0),
102   i11_(0)
103 {}
```

7.27.4.2 TauSolver::TauSolver (int *kx*, int *kz*, Real *Lx*, Real *Lz*, Real *a*, Real *b*, Real *lambda*, Real *nu*, int *nChebyModes*, bool *tauCorrection* = true)

References a_, abs(), b_, diff(), diff2(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), Greater(), i00_, i01_, i10_, i11_, influenceCorrection(), kx_, kz_, lambda_, MINIMUM_DISCRIMINANT, N_, n_func(), Nb_, nu_, P_0_, P_minus_, P_plus_, pressureHelmholtz_, sigma0_Nb1_, sigma0_Nb_, HelmholtzSolver::solve(), Spectral, v_0_, v_minus_, v_plus_, and velocityHelmholtz_.

```
107   :
108   N_(nChebyModes),
109   Nb_(nChebyModes-1),
110   kx_(kx),
111   kz_(kz),
112   two_pi_kxLx_(2*pi*kx/Lx),
113   two_pi_kzLz_(2*pi*kz/Lz),
114   kappa2_(4*square(pi)*(square(kx/Lx)+square(kz/Lz))),
115   a_(a),
116   b_(b),
117   lambda_(lambda),
118   nu_(nu),
119   tauCorrection_(tauCorrection),
120   pressureHelmholtz_(N_, a_, b_, kappa2_),
121   velocityHelmholtz_(N_, a_, b_, lambda_, nu_),
122   P_0_(N_, a_, b_, Spectral),
123   v_0_(N_, a_, b_, Spectral),
124   P_plus_(N_, a_, b_, Spectral),
125   v_plus_(N_, a_, b_, Spectral),
126   P_minus_(N_, a_, b_, Spectral),
127   v_minus_(N_, a_, b_, Spectral),
128   i00_(0),
129   i01_(0),
130   i10_(0),
131   i11_(0)
132 {
133
```

```
134   //
================================================================
====
135   // Calculate the influence matrix.
136
137   // Make some local aliases to temp storage, for readability
138   // This is not an efficiency issue, since TauSolvers are constructed once.
139   ChebyCoeff dvplus_dy(N_, a_, b_, Spectral);
140   ChebyCoeff dvminus_dy(N_, a_, b_, Spectral);
141   ChebyCoeff zero(N_, a_, b_, Spectral);
142   ChebyCoeff dPdy(N_, a_, b_, Spectral);
143
144   // Solve homogeneous Helmholtz with P(-1) = 0, P(1) = 1.
145   pressureHelmholtz_.solve(P_plus_, zero, 0.0, 1.0);   // eqn 7.3.25 discrete
146   diff(P_plus_, dPdy);
147   velocityHelmholtz_.solve(v_plus_, dPdy, 0.0, 0.0);   // eqn 7.3.26discrete
148
149   // Solve homogeneous Helmholtz with P(-1) = 1, P(1) = 0.
150   pressureHelmholtz_.solve(P_minus_, zero, 1.0, 0.0); // eqn 7.3.25 discrete
151   diff(P_minus_, dPdy);
152   velocityHelmholtz_.solve(v_minus_, dPdy, 0.0, 0.0); // eqn 7.3.26 discrete
153
154   // Calculate influence matrix elements.
155   diff(v_plus_,  dvplus_dy);
156   diff(v_minus_, dvminus_dy);
157
158   Real A = dvplus_dy.eval_b();
159   Real B = dvminus_dy.eval_b();
160   Real C = dvplus_dy.eval_a();
161   Real D = dvminus_dy.eval_a();
162   Real discriminant = A*D - B*C;
163
164   // We know influence matrix is rank-deficient for kx==kz==0. It shouldn't
165   // be for other wave numbers.
166   if (kx_ !=0 || kz_!=0)
167     assert((abs(discriminant) / Greater(abs(A*D), abs(B*C)))
> MINIMUM_DISCRIMINANT);
168
169   // Go ahead and divide by zero for kx==kz==0 case. Influence matrix is
170   // unused in that case. Pollute the entries NaN's to make sure.
171   i00_ =  D/discriminant;
172   i01_ =  -B/discriminant;
173   i10_ =  -C/discriminant;
174   i11_ =  A/discriminant;
175
```

```
176   //
==========================================================================
==
177   // Solve the B0 problem for tau corrections in solve(P,v)
178   ChebyCoeff p0_rhs(N_, a_, b_, Spectral);
179   for (int i=0; i<=Nb_; ++i)
180     p0_rhs[i] = n_func(i, Nb_);
181
182   pressureHelmholtz_.solve(P_0_, p0_rhs, 0.0, 0.0); // eqn 7.3.41a
183
184   ChebyCoeff dP0dy = diff(P_0_);
185   //dP0dy_Nb1_ = dP0dy[Nb_-1];
186   //dP0dy_Nb_  = dP0dy[Nb_];
187   velocityHelmholtz_.solve(v_0_, dP0dy, 0.0, 0.0);  // eqn 7.3.41b
188
189   influenceCorrection(P_0_, v_0_);
190
191   ChebyCoeff v0yy = diff2(v_0_);
192   //sigma0_Nb1_ = nu_*v0yy[Nb_-1] -(lambda_*v_0_[Nb_-1] + dP0dy[Nb_-1]);
193   //sigma0_Nb_  = nu_*v0yy[Nb_]   -(lambda_*v_0_[Nb_]   + dP0dy[Nb_]);
194   sigma0_Nb1_  = lambda_*v_0_[Nb_-1] + dP0dy[Nb_-1] - nu_*v0yy[Nb_-1];
195   sigma0_Nb_   = lambda_*v_0_[Nb_]   + dP0dy[Nb_]   - nu_*v0yy[Nb_];
196
197 }
```

Here is the call graph for this function:

7.27.5 Member Function Documentation

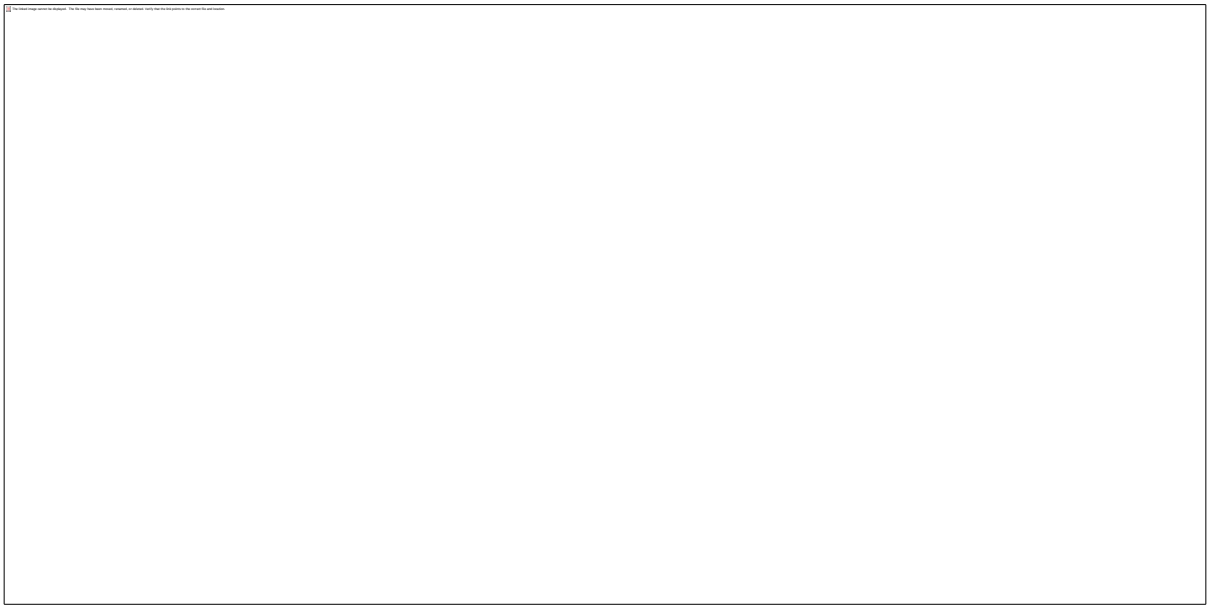
7.27.5.1 void TauSolver::influenceCorrection (ChebyCoeff & *P*, ChebyCoeff & *v*) const

References diff(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), i00_, i01_, i10_, i11_, N_, P_minus_, P_plus_, v_minus_, and v_plus_.

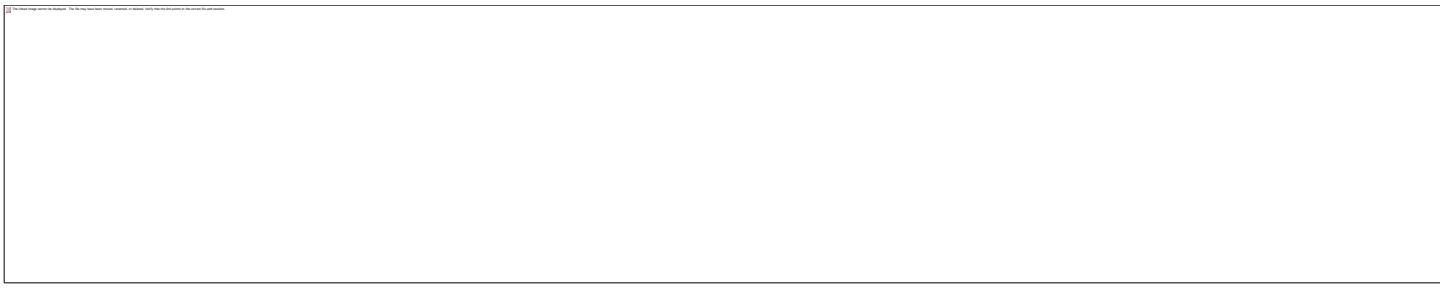
Referenced by solve_P_and_v(), and TauSolver().

```
199                                                 {
200
201   ChebyCoeff tmp = diff(v);
202   Real dvp_dy_plus  = tmp.eval_b();
203   Real dvp_dy_minus = tmp.eval_a();
204   Real delta_plus  = -i00_*dvp_dy_plus - i01_*dvp_dy_minus;
205   Real delta_minus = -i10_*dvp_dy_plus - i11_*dvp_dy_minus;
206
207   // Add the influence matrix corrections to the particular solutions
208   // to get a solution that statisfies both v(+-1)==0 and v'(+-1)==0.
209   for (int i=0; i<N_; ++i) {
210     P[i] += delta_plus*P_plus_[i] + delta_minus*P_minus_[i];
211     v[i] += delta_plus*v_plus_[i] + delta_minus*v_minus_[i];
212   }
213 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.27.5.2 int TauSolver::kx () const [inline]

References kx_.

```
94 {return kx_;}
```

7.27.5.3 int TauSolver::kz () const [inline]

References kz_.

```
95 {return kz_;}
```

7.27.5.4 Real TauSolver::lambda () const [inline]

References lambda_.

```
98 {return lambda_;}
```

7.27.5.5 Real TauSolver::nu () const [inline]

References nu_.

```
99 {return nu_;}
```

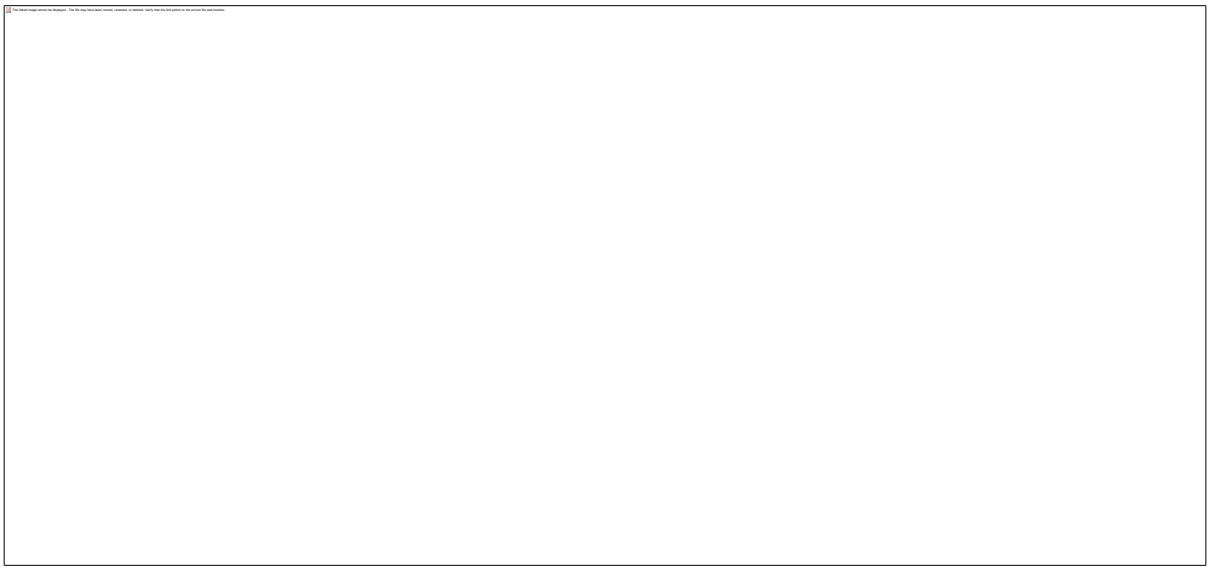
7.27.5.6 void TauSolver::solve (ComplexChebyCoeff & *u*, ComplexChebyCoeff & *v*, ComplexChebyCoeff & *w*, ComplexChebyCoeff & *P*, Real & *dPdx*, const ComplexChebyCoeff & *Rx*, const ComplexChebyCoeff & *Ry*, const ComplexChebyCoeff & *Rz*, Real *umean*) const

References a_, b_, diff(), I(), ComplexChebyCoeff::im, kx_, kz_, N_, pi, ComplexChebyCoeff::re, ComplexChebyCoeff::set(), HelmholtzSolver::solve(), solve_P_and_v(), Spectral, two_pi_kxLx_, two_pi_kzLz_, and velocityHelmholtz_.

```
447                                {
448
```

```
449  // This function should only be called for kx==kz==0, since the enforcing
450  // const velocity flux makes sense only for that case. Divergence is not a
451  // problem here, so solve everything via momentum.
452  assert(kx_==0 && kz_==0);
453
454  ComplexChebyCoeff r(N_,a_,b_,Spectral);
455  Real pi2 = 2*pi;
456  Complex pi2i = pi2*I;
457  Real sigmaNb1;          // tau correction term
458  Real sigmaNb;           // tau correction term
459
460  ChebyCoeff rr(N_,a_,b_,Spectral);
461
462  // Re and Im parts of v and P eqns decouple. Solve them seperately.
463  diff(Ry.re, rr);
464  int n; // MSVC++ FOR-SCOPE BUG
465  for (n=0; n<N_; ++n)
466    rr[n] -= two_pi_kxLx_*Rx.im[n] + two_pi_kzLz_*Rz.im[n];
467  solve_P_and_v(P.re, v.re, rr, Ry.re, sigmaNb1, sigmaNb);
468
469  diff(Ry.im, rr);
470  for (n=0; n<N_; ++n)
471    rr[n] += two_pi_kxLx_*Rx.re[n] + two_pi_kzLz_*Rz.re[n];
472  solve_P_and_v(P.im, v.im, rr, Ry.im, sigmaNb1, sigmaNb);
473
474  // Re and Im parts of u and w eqns seperate.
475  // Use r as temporary space to store RHS of eqns.
476  for (n=0; n<N_; ++n)
477    r.set(n, two_pi_kxLx_*I*P[n] - Rx[n]);
478
479  velocityHelmholtz_.solve(u.re, dPdx, r.re, umean, 0.0, 0.0);
480  velocityHelmholtz_.solve(u.im, r.im, 0.0, 0.0);
481
482  for (n=0; n<N_; ++n)
483    r.set(n, two_pi_kzLz_*I*P[n] - Rz[n]);
484
485  velocityHelmholtz_.solve(w.re, r.re, 0.0, 0.0);
486  velocityHelmholtz_.solve(w.im, r.im, 0.0, 0.0);
487
488 #ifdef DEBUG
489  //verify(u,v,w,P, dPdx, Rx,Ry,Rz, umean);
490 #endif
491
492
493  return;
494 }
```

Here is the call graph for this function:



7.27.5.7 void TauSolver::solve (ComplexChebyCoeff & *u*, ComplexChebyCoeff & *v*, ComplexChebyCoeff & *w*, ComplexChebyCoeff & *P*, const ComplexChebyCoeff & *Rx*, const ComplexChebyCoeff & *Ry*, const ComplexChebyCoeff & *Rz*) const

References a_, b_, diff(), I(), ComplexChebyCoeff::im, N_, ComplexChebyCoeff::re, ComplexChebyCoeff::set(), HelmholtzSolver::solve(), solve_P_and_v(), Spectral, two_pi_kxLx_, two_pi_kzLz_, and velocityHelmholtz_.

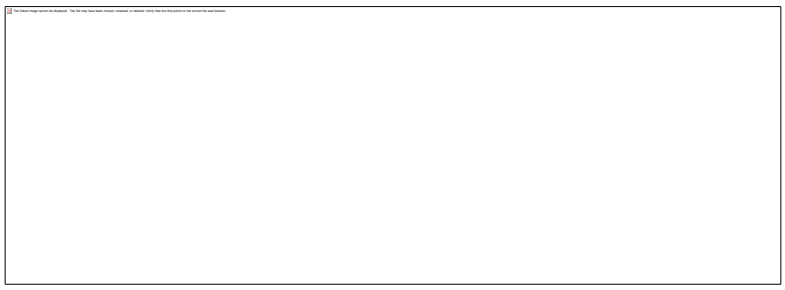
Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), and MultistepDNS::advance().

```
384 {
385
386   ComplexChebyCoeff r(N_,a_,b_,Spectral);
387   Real sigmaNb1;
388   Real sigmaNb;
389
390   ChebyCoeff rr(N_,a_,b_,Spectral);
391
392   // Always solve v(y) from momentum, and get P.
393   // Re and Im parts of v and P eqns decouple. Solve them seperately.
394   diff(Ry.re, rr);
395   int n; // MSVC++ FOR-SCOPE BUG
396   for (n=0; n<N_; ++n)
397     rr[n] -= two_pi_kxLx_*Rx.im[n] + two_pi_kzLz_*Rz.im[n];
398   solve_P_and_v(P.re, v.re, rr, Ry.re, sigmaNb1, sigmaNb);
399
```

```
400   diff(Ry.im, rr);
401   for (n=0; n<N_; ++n)
402     rr[n] += two_pi_kxLx_*Rx.re[n] + two_pi_kzLz_*Rz.re[n];
403   solve_P_and_v(P.im, v.im, rr, Ry.im, sigmaNb1, sigmaNb);
404
405   // Re and Im parts of u and w eqns seperate.
406   // Use r as temporary space to store RHS of eqns.
407   for (n=0; n<N_; ++n)
408     r.set(n, two_pi_kxLx_*I*P[n] - Rx[n]);
409   //Complex c = pi2i*kxLx_*P[n] - Rx[n];
410   //r.set(n,c);
411
412   velocityHelmholtz_.solve(u.re, r.re, 0.0, 0.0);
413   velocityHelmholtz_.solve(u.im, r.im, 0.0, 0.0);
414
415   for (n=0; n<N_; ++n)
416     r.set(n, two_pi_kzLz_*I*P[n] - Rz[n]);
417   //Complex c = pi2i*kzLz_*P[n] - Rz[n];
418   //r.set(n, c);
419
420   velocityHelmholtz_.solve(w.re, r.re, 0.0, 0.0);
421   velocityHelmholtz_.solve(w.im, r.im, 0.0, 0.0);
422
423   // This is for debugging ONLY
424   /*************************
425   if ((two_pi_kxLx_ == 0.0 & kx_ != 0) ||
426       (two_pi_kzLz_ == 0.0 & kz_ != 0)) {
427     verify(u,v,w,P,Rx,Ry,Rz,true);
428     u.setToZero();
429     v.setToZero();
430     w.setToZero();
431     P.setToZero();
432   }
433   *************************/
434
435 #ifdef DEBUG
436   //verify(u,v,w,P,Rx,Ry,Rz);
437 #endif
438   return;
439 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



7.27.5.8 void TauSolver::solve_P_and_v (ChebyCoeff & *P*, ChebyCoeff & *v*, const ChebyCoeff & *r*, const ChebyCoeff & *Ry*, Real & *sigmaNb1*, Real & *sigmaNb*) const

References diff(), diff2(), influenceCorrection(), kx_, kz_, lambda_, N_, Nb_, nu_, P_0_, pressureHelmholtz_, sigma0_Nb1_, sigma0_Nb_, HelmholtzSolver::solve(), tauCorrection_, v_0_, and velocityHelmholtz_.

Referenced by solve().

```
218       {
219
220   // P is Canuto & Hussaini's Ppart particular solution after this solve
221   pressureHelmholtz_.solve(P, r, 0.0, 0.0); // eqn 7.3.25 discrete HH1
222
223   // kx==kz==0 is a degenerate case for which the influence matrix is
224   // rank-deficient, the v solution is identically zero and P satisfies
225   // P' == Ry (which is equivalent to P'' == div(R) == r, found above).
226   if (kx_ == 0 && kz_ ==0) {
```

```
227     for (int i=0; i<N_; ++i)
228       v[i] = 0.0;
229     return;
230   }
231
232   // The rest of this method is for the case kx != 0 or kz != 0.
233   ChebyCoeff tmp = diff(P);
234   tmp -= Ry;
235
236   // v is Canuto & Hussaini's vpart particular solution after this solve
237   velocityHelmholtz_.solve(v, tmp, 0.0, 0.0); // eqn 7.3.25 discrete HH2
238   influenceCorrection(P,v);
239
240   // Jump ship if not doing tau correction
241   if (!tauCorrection_)
242     return;
243
244   // Add up tau correction terms. Quotes are from Canuto & Hussaini pg 219
245   // My Nb is Canuto's N.
246   // My sigma1_Nb  is Canuto's sigma_{1m} for m=N
247   // My sigma1_Nb1 is Canuto's sigma_{1m} for m=N-1
248
249   // Set tmp = vyy;
250   diff2(v,tmp);
251
252   // "define sigma_{1m} and sigma_{0m} for m = N-1, N as the tau
253   // terms that must be added to the v-momentum eqns for (P1,v1) and
254   // (P0,v0) for them to hold"
255   Real sigma1_Nb  = lambda_*v[Nb_]   - nu_*tmp[Nb_]   - Ry[Nb_];
256   Real sigma1_Nb1 = lambda_*v[Nb_-1] - nu_*tmp[Nb_-1] - Ry[Nb_-1];
257   diff(P, tmp);
258   sigma1_Nb  += tmp[Nb_];
259   sigma1_Nb1 += tmp[Nb_-1];
260
261   // "One can show that sigma_m = sigma_{1m}/(1-sigma_{0m}, m=N-1,N"
262   sigmaNb  = sigma1_Nb/(1.0 - sigma0_Nb_);
263   sigmaNb1 = sigma1_Nb1/(1.0 - sigma0_Nb1_);
264
265   // " and that ..."
266   for (int i=0; i<=Nb_; ++i) {
267     P[i]   += ((i%2 == 0) ? sigmaNb1 : sigmaNb) * P_0_[i];
268     v[i]   += ((i%2 == 0) ? sigmaNb  : sigmaNb1) * v_0_[i];
269   }
270
271   //#ifdef DEBUG
272   //verify_P_and_v(P,v,r,Ry, sigmaNb1, sigmaNb, true);
```

```
273   //#endif
274
275   return;
276 }
```

Here is the call graph for this function:

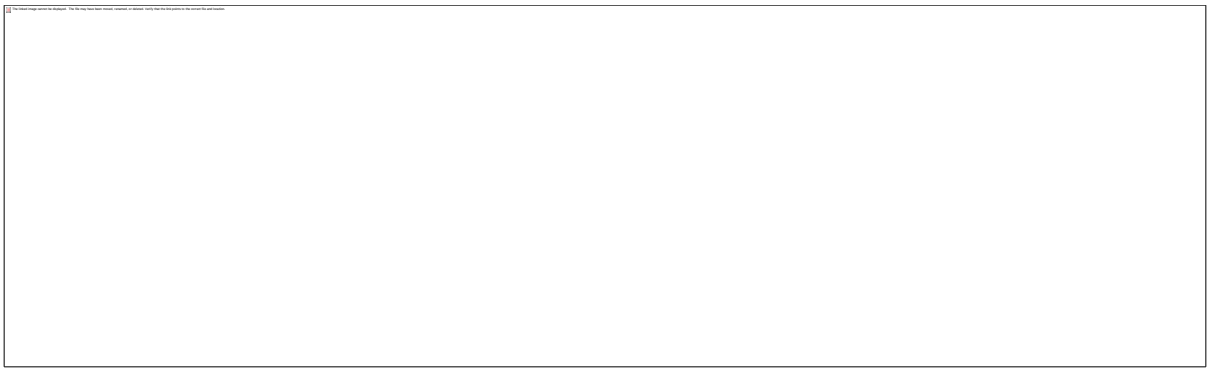
Here is the caller graph for this function:

7.27.5.9 Real TauSolver::tauDist (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*) const [private]

References L2Dist(), and ComplexChebyCoeff::numModes().

```
686                                                    {
687   ComplexChebyCoeff utmp(u.numModes()-2, u);
688   ComplexChebyCoeff vtmp(v.numModes()-2, v);
689   return L2Dist(utmp,vtmp);
690 }
```

Here is the call graph for this function:



7.27.5.10 Real TauSolver::tauDist (const ChebyCoeff & *u*, const ChebyCoeff & *v*) const [private]

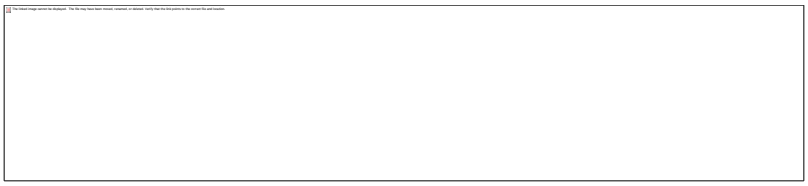
References L2Dist(), and ChebyCoeff::numModes().

Referenced by verify(), and verify_P_and_v().

```
681                                                 {
682   ChebyCoeff utmp(u.numModes()-2, u);
683   ChebyCoeff vtmp(v.numModes()-2, v);
684   return L2Dist(utmp,vtmp);
685 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

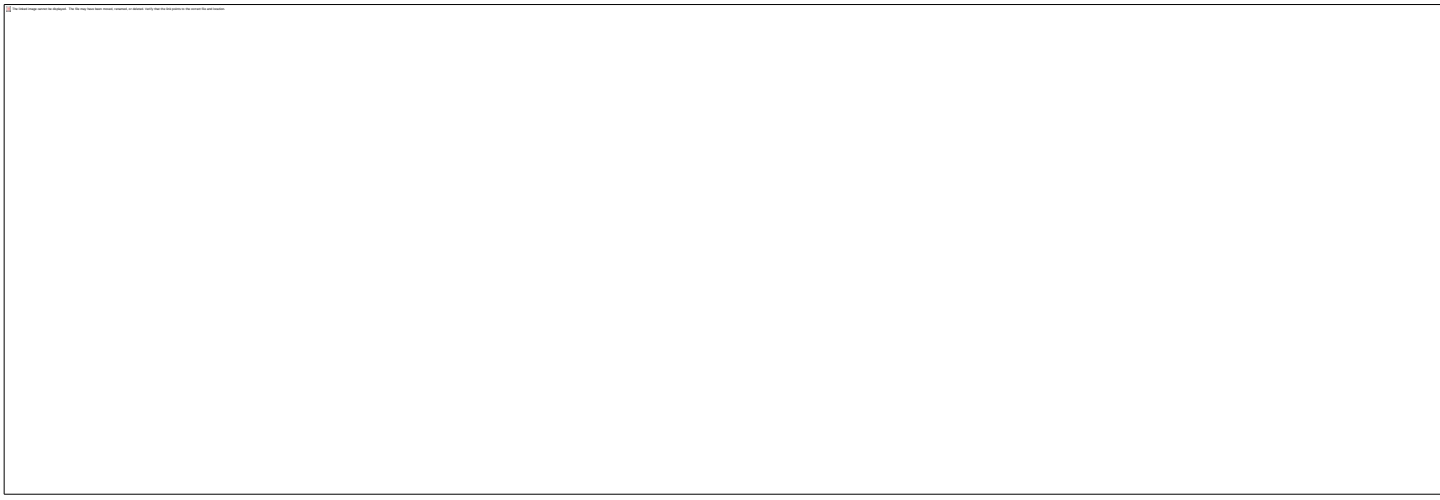


7.27.5.11 Real TauSolver::tauNorm (const ComplexChebyCoeff & *u*) const [private]

References L2Norm(), and ComplexChebyCoeff::numModes().

```
677                                                 {
678   ComplexChebyCoeff tmp(u.numModes()-2, u);
679   return L2Norm(u);
680 }
```

Here is the call graph for this function:



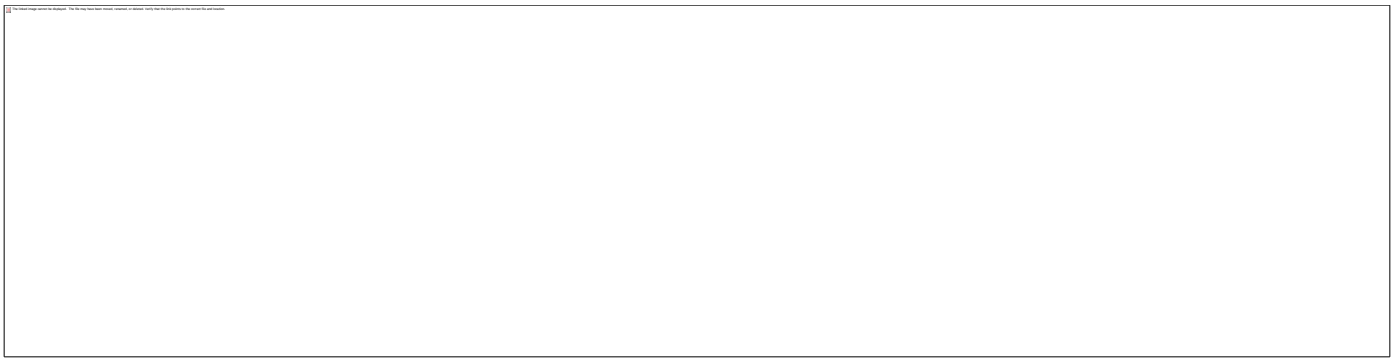
7.27.5.12 Real TauSolver::tauNorm (const ChebyCoeff & *u*) const [private]

References L2Norm(), and ChebyCoeff::numModes().

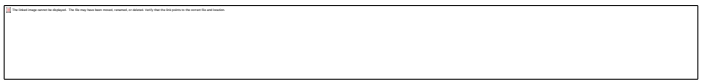
Referenced by verify().

```
673                                               {
674   ChebyCoeff tmp(u.numModes()-2, u);
675   return L2Norm(u);
676 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.27.5.13 Real TauSolver::verify (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, const ComplexChebyCoeff & *w*, const ComplexChebyCoeff & *P*, Real *dPdx*, const ComplexChebyCoeff & *Rx*, const ComplexChebyCoeff & *Ry*, const ComplexChebyCoeff & *Rz*, Real *umean*, bool *verbose* = false) const

References a_, abs(), abs2(), ComplexChebyCoeff::add(), b_, diff(), diff2(), EPSILON, ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), I(), ComplexChebyCoeff::im, kappa2_, kx_, kz_, L2Dist(), L2Norm(), lambda_, ComplexChebyCoeff::mean(), N_, nu_, Re(), ComplexChebyCoeff::re, Spectral, tauDist(), tauNorm(), two_pi_kxLx_, and two_pi_kzLz_.

```
511                                         {
512
513   // verify   nu u"(y) - lambda u(y) - grad P = -R,
514   //                                 div u = 0
515   //                                u(+-1) = 0
516
517   if (verbose) {
518     cout << "TauSolver::verify(u,v,w,P,dPdx,Rx,Ry,Rz,umean,verbose) {" << endl;
519     cout << " kx kz == " << kx_ << ' ' << kz_ << endl;
520   }
521   ComplexChebyCoeff lhs(N_,a_,b_,Spectral);
522   ComplexChebyCoeff tmp(N_,a_,b_,Spectral);
523   Real error = 0.0;
524   Real terr = 0.0;
525   Real lerr = 0.0;
526
527   // Verify u eqn, -nu u" + lambda u + dP/dx == Rx
```

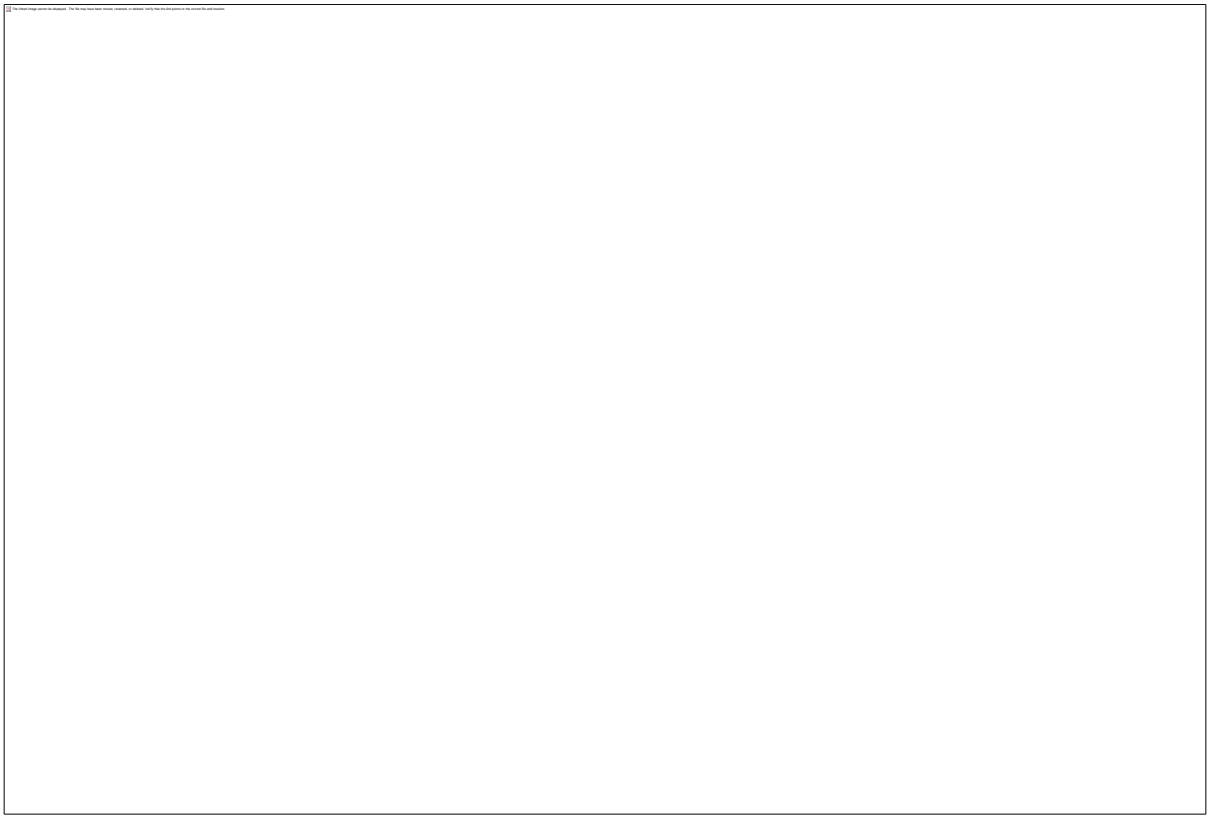
```
528  lhs = u;
529  lhs *= lambda_;
530  diff2(u, tmp);
531  tmp *= nu_;
532  lhs -= tmp;
533  tmp = P;
534  //tmp *= pi2*I*kxLx_; gcc-2.95 bug makes the following necessary
535  tmp *= Complex(0.0, two_pi_kxLx_);
536  lhs += tmp;
537  lhs.re[0] += dPdx;
538
539  terr = tauDist(lhs, Rx);
540  lerr = L2Dist(lhs, Rx);
541  error += lerr;
542  if (verbose) {
543    cout << "L2Norm(Rx) == " << L2Norm(Rx) << endl;
544    cout << "tauDist(nu u" - lambda u - dP/dx, -Rx) == " << terr << endl;
545    cout << " L2Dist(nu u" - lambda u - dP/dx, -Rx) == " << lerr << endl;
546  }
547
548  // Verify v eqn.
549  lhs = v;
550  lhs *= lambda_;
551  diff2(v, tmp);
552  tmp *= nu_;
553  lhs -= tmp;
554  diff(P, tmp);
555  lhs += tmp;
556  terr = tauDist(lhs, Ry);
557  lerr = L2Dist(lhs, Ry);
558  error += lerr;
559  if (verbose) {
560    cout << "L2Norm(Ry) == " << L2Norm(Ry) << endl;
561    cout << "tauDist(nu v" - lambda v - dP/dy, -Ry) == " << terr << endl;
562    cout << " L2Dist(nu v" - lambda v - dP/dy, -Ry) == " << lerr << endl;
563  }
564
565  // Verify w eqn
566  lhs = w;
567  lhs *= lambda_;
568  diff2(w, tmp);
569  tmp *= nu_;
570  lhs -= tmp;
571  tmp = P;
572  tmp *= Complex(0.0, two_pi_kzLz_);
573  lhs += tmp;
```

```
574  terr = tauDist(lhs, Rz);
575  lerr = L2Dist(lhs, Rz);
576  error += lerr;
577  if (verbose) {
578    cout << "L2Norm(Rz) == " << L2Norm(Rz) << endl;
579    cout << "tauDist(nu w" - lambda w - dP/dz, -Rz) == " << terr << endl;
580    cout << " L2Dist(nu w" - lambda w - dP/dz, -Rz) == " << lerr << endl;
581  }
582
583  // Verify P eqn. P" - kappa^2 P = div R
584  diff2(P, lhs);
585  tmp = P;
586  tmp *= -kappa2_;
587  lhs += tmp;
588
589  ComplexChebyCoeff r(N_,a_,b_,Spectral);
590  ChebyCoeff r_re(N_,a_,b_,Spectral);
591  ChebyCoeff r_im(N_,a_,b_,Spectral);
592
593  // Re and Im parts of v and P eqns decouple. Solve them seperately.
594  diff(Ry.re, r_re);
595  int n; // MSVC++ FOR-SCOPE BUG
596  for (n=0; n<N_; ++n)
597    r_re[n] -= two_pi_kxLx_*Rx.im[n] + two_pi_kzLz_*Rz.im[n];
598
599  diff(Ry.im, r_im);
600  for (n=0; n<N_; ++n)
601    r_im[n] += two_pi_kxLx_*Rx.re[n] + two_pi_kzLz_*Rz.re[n];
602
603  r.re = r_re;
604  r.im = r_im;
605  terr = tauDist(lhs, r);
606  lerr = L2Dist(lhs, r);
607  error += lerr;
608  if (verbose) {
609    cout << "L2Norm(div R) == " << L2Norm(r) << endl;
610    cout << "tauDist(P" - k^2 P, div R)   == " << terr << endl;
611    cout << " L2Dist(P" - k^2 P, div R)   == " << lerr << endl;
612  }
613  // Don't assert on pressure eqn, since it's modified for tau correction.
614  //else
615  //assert(error<EPSILON);
616
617
618  // Verify divergence
619  diff(v, tmp);
```

```
620  for (n=0; n<N_; ++n)
621    tmp.add(n, I*(two_pi_kxLx_*u[n] + two_pi_kzLz_*w[n]));
622    //tmp.add(n, I*pi2*(kxLx_*u[n] + kzLz_*w[n]));
623
624  terr = tauNorm(tmp);
625  lerr = L2Norm(tmp);
626  error += lerr;
627  if (verbose) {
628    cout << "tauNorm(div) == " << terr << endl;
629    cout << " L2Norm(div) == " << lerr << endl;
630  }
631
632  // BCs
633  Complex ua = u.eval_a();
634  Complex ub = u.eval_b();
635  error += abs(ua) + abs(ub);
636  if (verbose)
637    cout << "u(a),u(b) == " << ua << " " << ub << endl;
638
639
640  Complex va = v.eval_a();
641  Complex vb = v.eval_b();
642  error += abs(va) + abs(vb);
643  if (verbose)
644    cout << "v(a),v(b) == " << va << " " << vb << endl;
645
646  ComplexChebyCoeff vy = diff(v);
647  Complex vya = vy.eval_a();
648  Complex vyb = vy.eval_b();
649  error += abs(vya) + abs(vyb);
650  if (verbose)
651    cout << "v' at a,b == " << vya << " " << vyb << endl;
652
653  Complex wa = u.eval_a();
654  Complex wb = u.eval_b();
655  error += abs(wa) + abs(wb);
656  if (verbose)
657    cout << "w(a),w(b) == " << wa << " " << wb << endl;
658
659  Real mean_error = abs2(Re(u.mean()) - umean);
660  error += mean_error;
661  if (verbose)
662    cout << "abs2(u.mean() - umean) == " << mean_error << endl;
663  else
664    assert(mean_error < EPSILON);
665
```

```
666   if (verbose) {
667     cout << "total verification error == " << error << endl;
668     cout << "} TauSolver::verify(...)" << endl;
669   }
670   return error;
671 }
```

Here is the call graph for this function:



7.27.5.14 Real TauSolver::verify (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, const ComplexChebyCoeff & *w*, const ComplexChebyCoeff & *P*, const ComplexChebyCoeff & *Rx*, const ComplexChebyCoeff & *Ry*, const ComplexChebyCoeff & *Rz*, bool *verbose* = false) const

References ComplexChebyCoeff::mean(), and Re().

```
499                                                 {
500
501   Real umean = Re(u.mean());
502   Real dPdx = 0.0;
503   return verify(u,v,w,P,dPdx,Rx,Ry,Rz,umean,verbose);
```

```
504 }
```

Here is the call graph for this function:

7.27.5.15 Real TauSolver::verify_P_and_v (const ChebyCoeff & *P*, const ChebyCoeff & *v*, const ChebyCoeff & *r*, const ChebyCoeff & *Ry*, Real *sigmaNb1*, Real *sigmaNb*, bool *verbose* = false) const

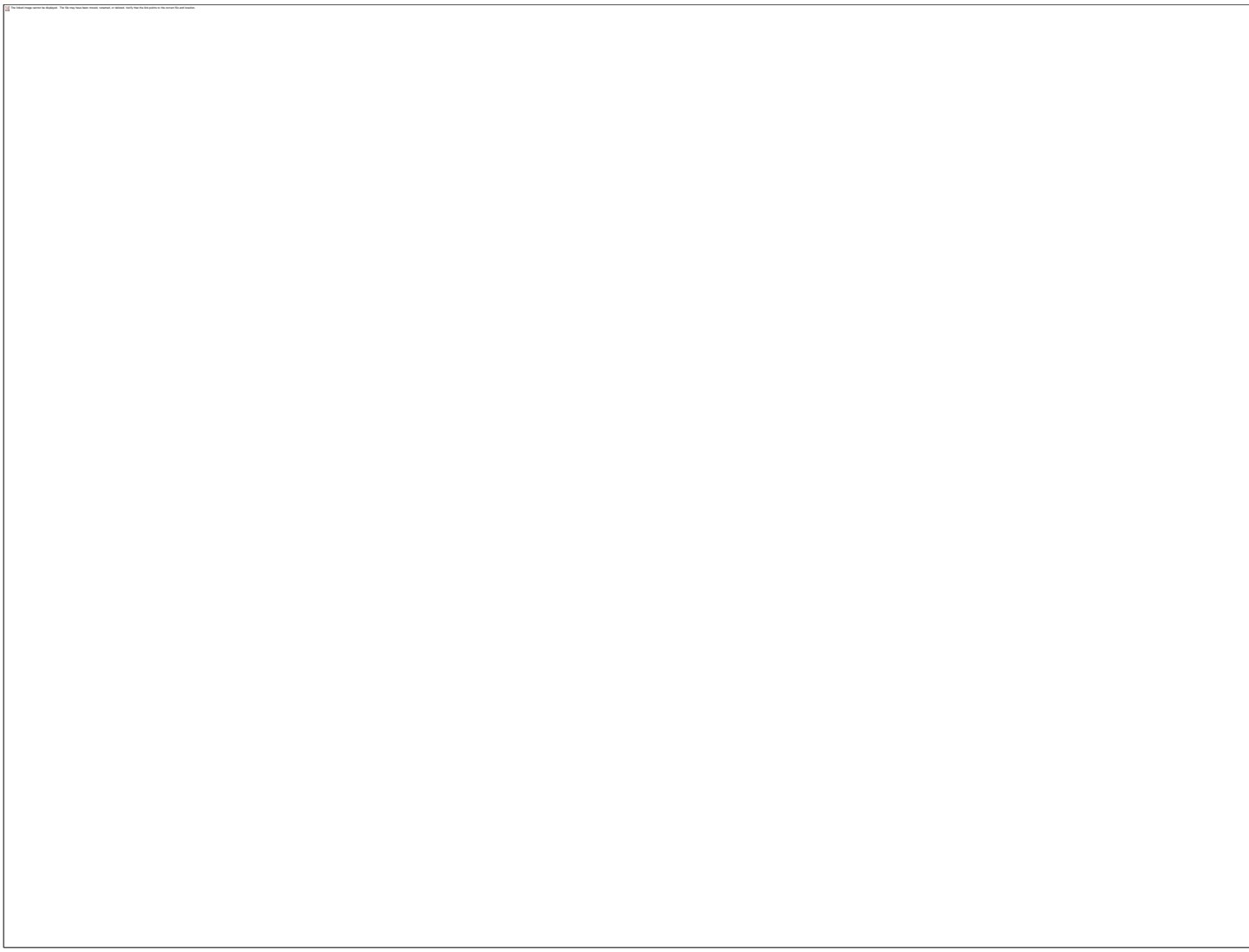
References a_, abs(), b_, diff(), EPSILON, ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), kappa2_, L2Dist(), L2Norm(), lambda_, N_, Nb_, nu_, Spectral, and tauDist().

```
281                                                 {
282
283   Real error = 0.0;
284
285   if (verbose) {
286     cout << "--------TauSolver::verify_P_and_v(P,v,r,Ry,sigmaNb1,sigmaNb) {" << endl;
287     cout << "nu = " << nu_ << "; lambda = " << lambda_ << endl;
288   }
289
290   ChebyCoeff Py =  diff(P);
291   ChebyCoeff Pyy = diff(Py);
292   ChebyCoeff p_lhs(P);
293   p_lhs *= -kappa2_;
294   p_lhs += Pyy;
295
296   Real l2err = L2Dist(p_lhs, r);
297   Real t2err = tauDist(p_lhs, r);
298   error += l2err;
299   if (verbose) {
300     cout << "Homog tauDist(P" - k^2 P, r)   == " << t2err << endl;
301     cout << "Homog  L2Dist(P" - k^2 P, r)   == " << l2err << endl;
302   }
303
304   ChebyCoeff vy = diff(v);
305   ChebyCoeff vyy = diff(vy);
306   ChebyCoeff v_lhs = nu_*vyy - lambda_*v - Py;
307   ChebyCoeff minusRy = Ry;
308   minusRy *= -1.0;
309
```

```
310  t2err = tauDist(v_lhs, minusRy);
311  l2err = L2Dist(v_lhs, minusRy);
312  error += l2err;
313  if (verbose) {
314    cout << "tauDist(v_lhs, -Ry) == " << t2err << endl;
315    cout << " L2Dist(v_lhs, -Ry) == " << l2err << endl;
316  }
317
318  // Now compare P and v eqns with tau correction terms.
319  ChebyCoeff sigma(N_,a_,b_,Spectral);
320  sigma[Nb_-1] = sigmaNb1;
321  sigma[Nb_]   = sigmaNb;
322  ChebyCoeff p_rhs(N_,a_,b_,Spectral);
323  diff(sigma, p_rhs);
324  p_rhs += r;
325
326  t2err = tauDist(p_lhs, p_rhs);
327  l2err = L2Dist(p_lhs, p_rhs);
328  error += l2err;
329  if (verbose) {
330    cout << "Homog tauDist(P" - k^2 P, r + sigma')   == " << t2err << endl;
331    cout << "Homog  L2Dist(P" - k^2 P, r + sigma')   == " << l2err << endl;
332  }
333
334  ChebyCoeff v_rhs(minusRy);
335  v_rhs -= sigma;
336
337  t2err = tauDist(v_lhs, v_rhs);
338  l2err = L2Dist(v_lhs, v_rhs);
339  error += l2err;
340  if (verbose) {
341    cout << "tauDist(v_lhs, -Ry - sigma) == " << tauDist(v_lhs, v_rhs) << endl;
342    cout << " L2Dist(v_lhs, -Ry - sigma) == " << L2Dist(v_lhs, v_rhs)  << endl;
343  }
344
345  Real vp = v.eval_b();
346  Real vm = v.eval_a();
347  Real vyp = vy.eval_b();
348  Real vym = vy.eval_a();
349  Real v_norm =
350    (abs(nu_*L2Norm(vyy)) + abs(lambda_*L2Norm(v))) + L2Norm(Py);
351  v_norm = (v_norm > EPSILON) ? v_norm : 1.0;
352
353  Real p_norm = L2Norm(Pyy) + kappa2_*L2Norm(P);
354  p_norm = (p_norm > EPSILON) ? p_norm : 1.0;
355
```

```
356
357   error += abs(vp)/v_norm;
358   error += abs(vm)/v_norm;
359   error += abs(vyp)/v_norm;
360   error += abs(vym)/v_norm;
361
362   if (verbose) {
363     cout << " v(+/- 1) == " << vp << ' ' << vm << endl;
364     cout << "v'(+/- 1) == " << vyp << ' ' << vym << endl;
365   }
366
367   //Real v_err = tauDist(v_lhs,v_rhs)/v_norm;
368   //Real p_err = tauDist(p_lhs,p_rhs)/p_norm;
369   //assert(v_err < EPSILON);
370   //assert(p_err < EPSILON);
371   if (verbose)
372     cout << "} TauSolver::verify_P_and_v" << endl;
373
374   return error;
375
376 }
```

Here is the call graph for this function:



7.27.6 Member Data Documentation

7.27.6.1 Real TauSolver::a_ [private]

Referenced by solve(), TauSolver(), verify(), and verify_P_and_v().

7.27.6.2 Real TauSolver::b_ [private]

Referenced by solve(), TauSolver(), verify(), and verify_P_and_v().

7.27.6.3 Real TauSolver::i00_ [private]

Referenced by influenceCorrection(), and TauSolver().

7.27.6.4 Real TauSolver::i01_ [private]

Referenced by influenceCorrection(), and TauSolver().

7.27.6.5 Real TauSolver::i10_ [private]

Referenced by influenceCorrection(), and TauSolver().

7.27.6.6 Real TauSolver::i11_ [private]

Referenced by influenceCorrection(), and TauSolver().

7.27.6.7 Real TauSolver::kappa2_ [private]

Referenced by verify(), and verify_P_and_v().

7.27.6.8 int TauSolver::kx_ [private]

Referenced by kx(), solve(), solve_P_and_v(), TauSolver(), and verify().

7.27.6.9 int TauSolver::kz_ [private]

Referenced by kz(), solve(), solve_P_and_v(), TauSolver(), and verify().

7.27.6.10 Real TauSolver::lambda_ [private]

Referenced by lambda(), solve_P_and_v(), TauSolver(), verify(), and verify_P_and_v().

7.27.6.11 int TauSolver::N_ [private]

Referenced by influenceCorrection(), solve(), solve_P_and_v(), TauSolver(), verify(), and verify_P_and_v().

7.27.6.12 int TauSolver::Nb_ [private]

Referenced by solve_P_and_v(), TauSolver(), and verify_P_and_v().

7.27.6.13 Real TauSolver::nu_ [private]

Referenced by nu(), solve_P_and_v(), TauSolver(), verify(), and verify_P_and_v().

7.27.6.14 ChebyCoeff TauSolver::P_0_ [private]

Referenced by solve_P_and_v(), and TauSolver().

7.27.6.15 ChebyCoeff TauSolver::P_minus_ [private]

Referenced by influenceCorrection(), and TauSolver().

7.27.6.16 ChebyCoeff TauSolver::P_plus_ [private]

Referenced by influenceCorrection(), and TauSolver().

7.27.6.17 HelmholtzSolver TauSolver::pressureHelmholtz_ [private]

Referenced by solve_P_and_v(), and TauSolver().

7.27.6.18 Real TauSolver::sigma0_Nb1_ [private]

Referenced by solve_P_and_v(), and TauSolver().

7.27.6.19 Real TauSolver::sigma0_Nb_ [private]

Referenced by solve_P_and_v(), and TauSolver().

7.27.6.20 bool TauSolver::tauCorrection_ [private]

Referenced by solve_P_and_v().

7.27.6.21 Real TauSolver::two_pi_kxLx_ [private]

Referenced by solve(), and verify().

7.27.6.22 Real TauSolver::two_pi_kzLz_ [private]

Referenced by solve(), and verify().

7.27.6.23 ChebyCoeff TauSolver::v_0_ [private]

Referenced by solve_P_and_v(), and TauSolver().

7.27.6.24 ChebyCoeff TauSolver::v_minus_ [private]

Referenced by influenceCorrection(), and TauSolver().

7.27.6.25 ChebyCoeff TauSolver::v_plus_ [private]

Referenced by influenceCorrection(), and TauSolver().

7.27.6.26 HelmholtzSolver TauSolver::velocityHelmholtz_ [private]

Referenced by solve(), solve_P_and_v(), and TauSolver().

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

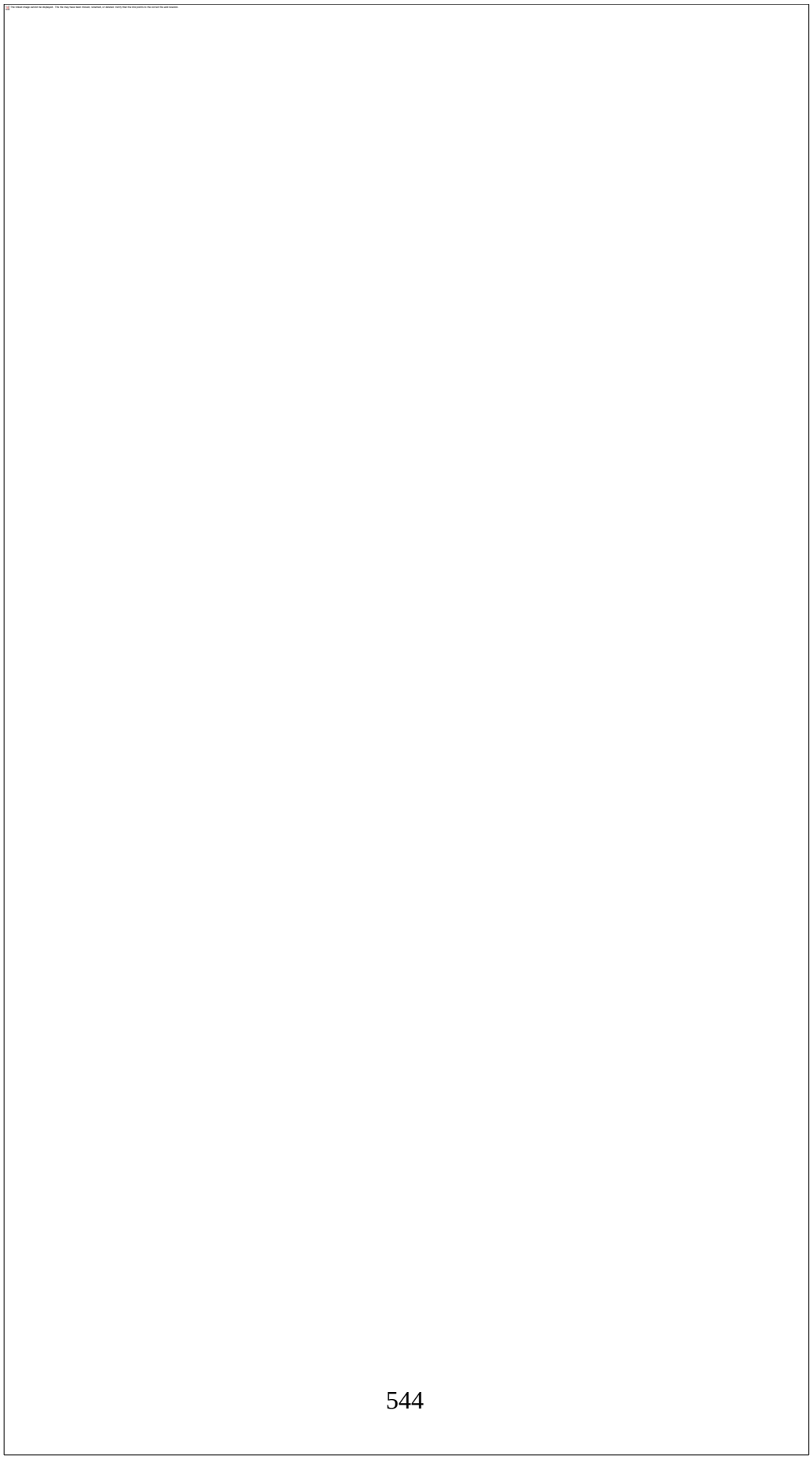
- /_cuda/channelflow-0.9.22/channelflow/tausolver.h
- /_cuda/channelflow-0.9.22/channelflow/tausolver.cpp

7.27.6.28

7.28 ThreadPool Class Reference

#include <dns.h>

Collaboration diagram for ThreadPool:



7.28.1 Public Member Functions

- void CreateThreads (int numOfThreads)
- void * Run ()
- void setLimits (Limits &)
- void DestroyThreadPool ()
- void executeTask (BaseTask *)

7.28.2 Static Public Member Functions

- static ThreadPool * GetInstance ()

7.28.3 Private Member Functions

- ThreadPool ()
- void WaitToComplete ()
- void WakeUp ()
- void setTask (BaseTask *)

7.28.4 Private Attributes

- int m_nmbThreads
- int m_total_running_threads
- bool m_stop_threads
- BaseTask * m_task
- bool * m_work
- map< pthread_t, int > m_map
- Limits m_lim
- pthread_mutex_t m_mutex
- pthread_cond_t m_cond
- pthread_mutex_t m_mutex_done
- pthread_cond_t m_cond_done

7.28.5 Static Private Attributes

- static ThreadPool * pInstance = NULL

7.28.6 Constructor & Destructor Documentation

7.28.6.1 ThreadPool::ThreadPool () [private]

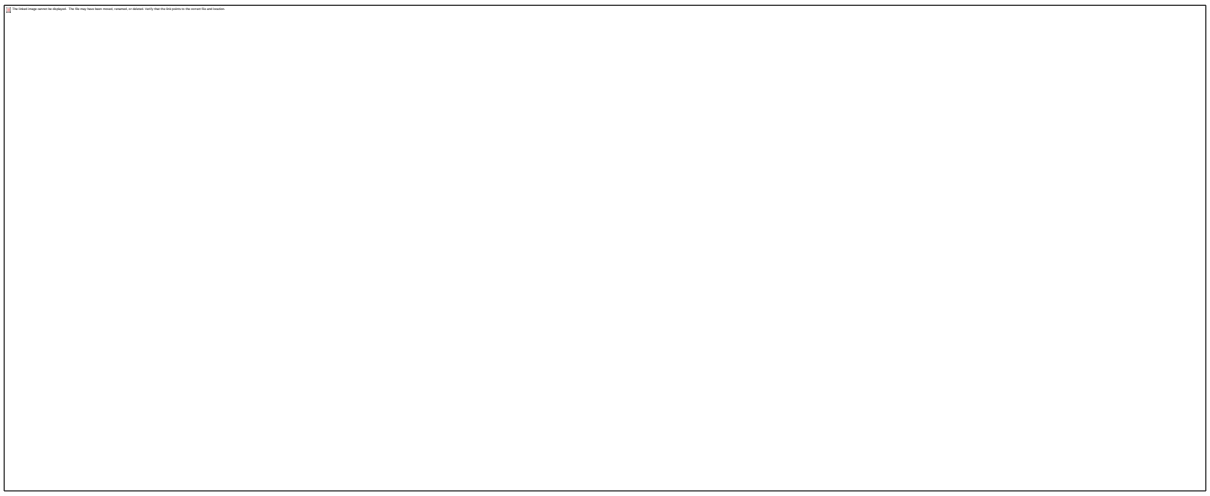
References m_cond, m_cond_done, m_mutex, m_mutex_done, and m_task.

Referenced by GetInstance().

```
413 {
414         m_task = NULL;
```

```
415         m_mutex=PTHREAD_MUTEX_INITIALIZER;
416         m_cond=PTHREAD_COND_INITIALIZER;
417
418         m_mutex_done = PTHREAD_MUTEX_INITIALIZER;
419         m_cond_done = PTHREAD_COND_INITIALIZER;
420
421 }
```

Here is the caller graph for this function:



7.28.7 Member Function Documentation

7.28.7.1 void ThreadPool::CreateThreads (int *numOfThreads*)

References m_map, m_nmbThreads, m_stop_threads, m_work, and start_thread().

```
433 {
434       pthread_t handel;
435       m_work = new bool[numOfThreads];
436       m_nmbThreads = numOfThreads;
437       m_stop_threads = false;
438
439       for(int i = 0; i < numOfThreads; ++i)
440       {
441         m_work[i] = false;
442         pthread_create(&handel, NULL, &start_thread ,(void *) this );
443         m_map[handel] = i;
444         //m_handel.push_back(handel);
445
446       }
```

```
447 }
```

Here is the call graph for this function:

7.28.7.2 void ThreadPool::DestroyThreadPool ()

References m_cond, m_mutex, and m_stop_threads.

```
466 {
467   m_stop_threads = true;
468   cout << "DestroyThreadPool called" << endl;
469 
470    pthread_mutex_lock(&m_mutex);
471       pthread_cond_broadcast(&m_cond);
472    pthread_mutex_unlock(&m_mutex);
473 
474 /*
475    for(int k = 0; k < m_handel.size(); k++)
476    {
477       pthread_join(m_handel[k], NULL);
478    }
479 */
480    //sleep(5);
481 
482 }
```

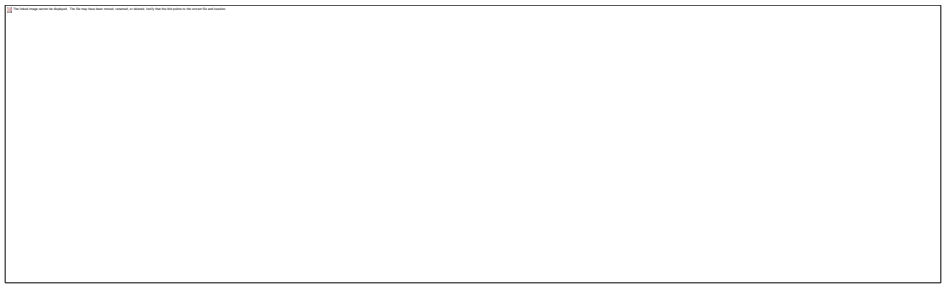
7.28.7.3 void ThreadPool::executeTask (BaseTask * *tsk*)

References setTask(), WaitToComplete(), and WakeUp().

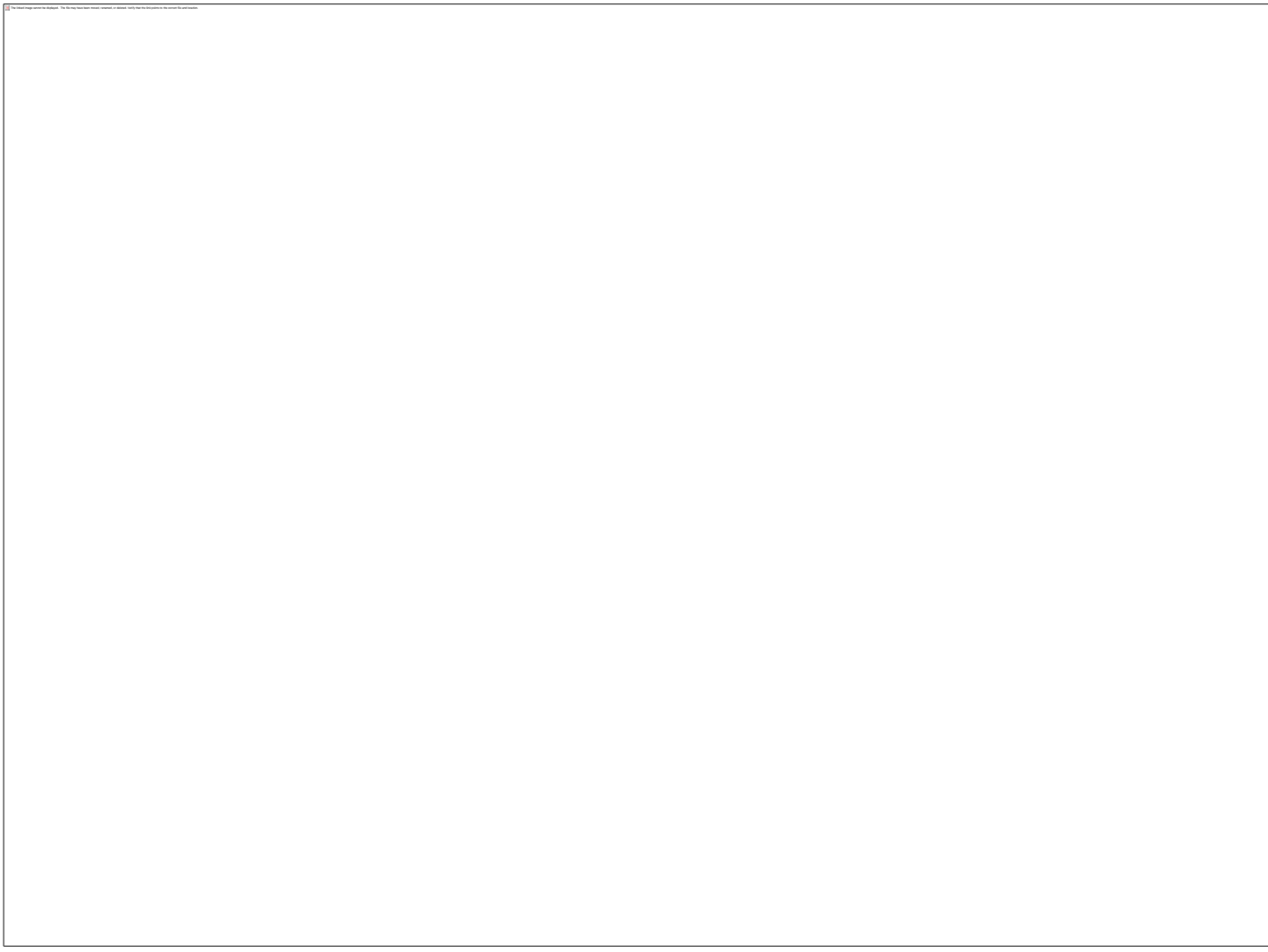
Referenced by skewsymmetricNL_THREAD().

```
491 {
492   setTask(tsk);
493   WakeUp();
494   WaitToComplete();
495 
496   delete tsk;
497 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.28.7.4 ThreadPool * ThreadPool::GetInstance () [static]

References pInstance, and ThreadPool().
Referenced by skewsymmetricNL_THREAD().

```
423 {
424         if (pInstance== NULL)
425         {
426                 pInstance = new ThreadPool();
427         }
428                 return pInstance;
429 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.28.7.5 void * ThreadPool::Run ()

References m_cond, m_cond_done, m_map, m_mutex, m_mutex_done, m_nmbThreads, m_stop_threads, m_task, m_total_running_threads, m_work, and BaseTask::Run().

Referenced by start_thread().

```
529 {
530     for (;;)
531     {
532       pthread_mutex_lock(&m_mutex);
533        int tn= m_map[pthread_self()];
534        while ( !m_work[tn] && m_stop_threads == false)
535        {
```

```
536           pthread_cond_wait(&m_cond, &m_mutex);
537
538         }
539         pthread_mutex_unlock(&m_mutex);
540
541         //cout <<"IG: worker thread id=" << tn << " running " <<endl;
542
543          if(m_stop_threads) break;
544
545          if(m_task) m_task->Run(tn,m_nmbThreads);
546          m_work[tn]= false;
547
548        pthread_mutex_lock(&m_mutex_done);
549           m_total_running_threads--;
550           pthread_cond_signal(&m_cond_done);
551            //cout << "IG: tn=" << tn << "finished m_total_running_threads="
<<m_total_running_threads <<endl;
552        pthread_mutex_unlock(&m_mutex_done);
553      }
554
555
556   return NULL;
557 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.28.7.6 void ThreadPool::setLimits (Limits & *lim*)

References m_lim.

```
520 {
521   m_lim = lim;
522
523 }
```

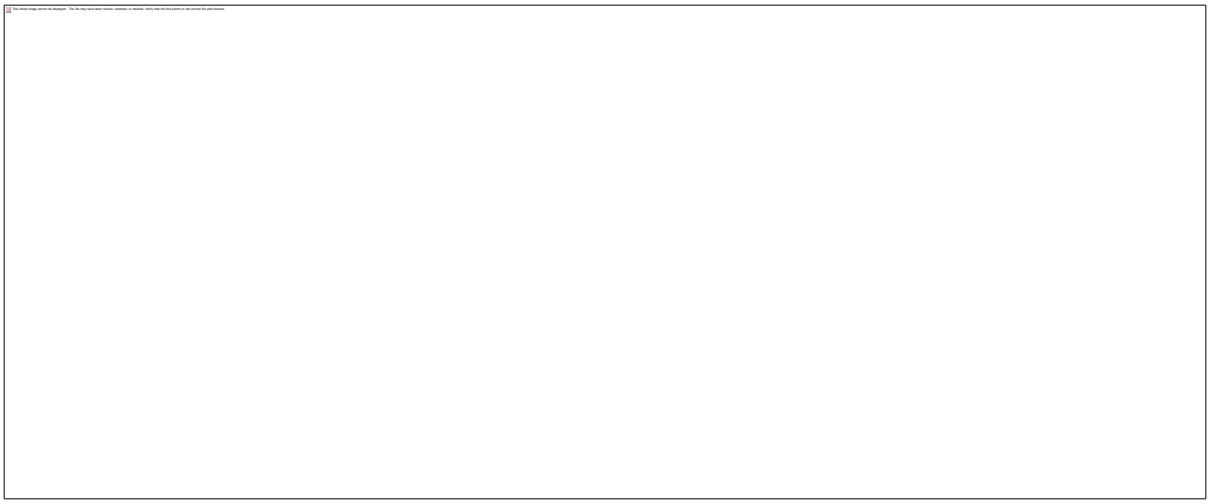
7.28.7.7 void ThreadPool::setTask (BaseTask * *tk*) [private]

References m_task.

Referenced by executeTask().

```
486 {
487  m_task = tk;
488 }
```

Here is the caller graph for this function:



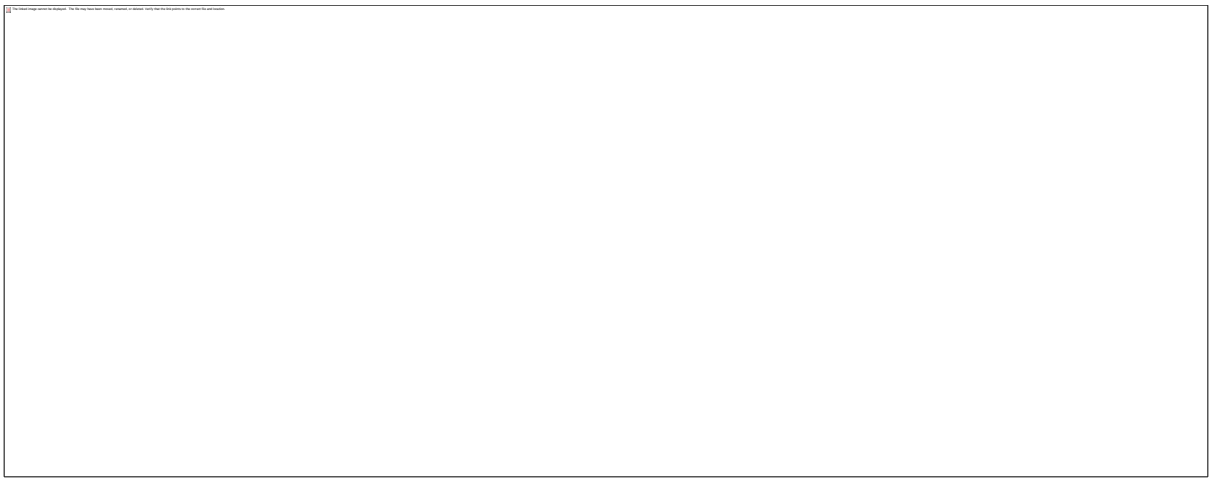
7.28.7.8 void ThreadPool::WaitToComplete () [private]

References m_cond_done, m_mutex_done, and m_total_running_threads.

Referenced by executeTask().

```
503 {
504     for(;;)
505     {
506       pthread_mutex_lock(&m_mutex_done);
507           while(m_total_running_threads > 0)
508           {
509               pthread_cond_wait(&m_cond_done, &m_mutex_done);
510               //cout << "IG wake up as thread finised " <<endl;
511           }
512        pthread_mutex_unlock(&m_mutex_done);
513        if(m_total_running_threads <= 0) break;
514     }
515 }
```

Here is the caller graph for this function:



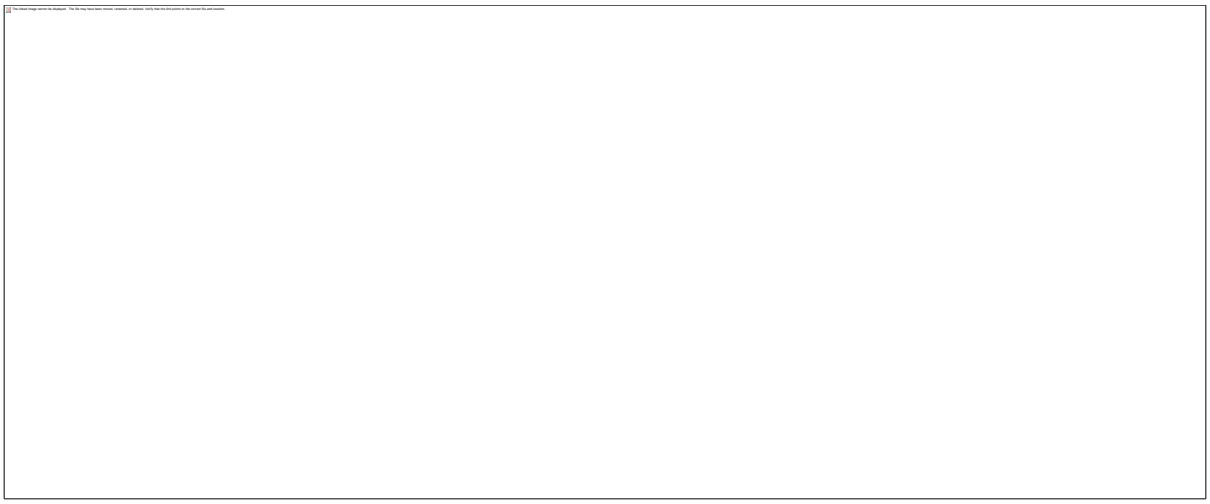
7.28.7.9 void ThreadPool::WakeUp () [private]

References m_cond, m_mutex, m_nmbThreads, m_stop_threads, m_total_running_threads, and m_work.

Referenced by executeTask().

```
450 {
451
452     m_total_running_threads = m_nmbThreads;
453     m_stop_threads = false;
454
455     for(int k = 0; k < m_nmbThreads; k++)
456     {
457       m_work[k] = true;
458     }
459
460     pthread_mutex_lock(&m_mutex);
461       pthread_cond_broadcast(&m_cond);
462     pthread_mutex_unlock(&m_mutex);
463 }
```

Here is the caller graph for this function:



7.28.8 Member Data Documentation

7.28.8.1 pthread_cond_t ThreadPool::m_cond [private]

Referenced by DestroyThreadPool(), Run(), ThreadPool(), and WakeUp().

7.28.8.2 pthread_cond_t ThreadPool::m_cond_done [private]

Referenced by Run(), ThreadPool(), and WaitToComplete().

7.28.8.3 Limits ThreadPool::m_lim [private]

Referenced by setLimits().

7.28.8.4 map<pthread_t,int> ThreadPool::m_map [private]

Referenced by CreateThreads(), and Run().

7.28.8.5 pthread_mutex_t ThreadPool::m_mutex [private]

Referenced by DestroyThreadPool(), Run(), ThreadPool(), and WakeUp().

7.28.8.6 pthread_mutex_t ThreadPool::m_mutex_done [private]

Referenced by Run(), ThreadPool(), and WaitToComplete().

7.28.8.7 int ThreadPool::m_nmbThreads [private]

Referenced by CreateThreads(), Run(), and WakeUp().

7.28.8.8 bool ThreadPool::m_stop_threads [private]

Referenced by CreateThreads(), DestroyThreadPool(), Run(), and WakeUp().

7.28.8.9 BaseTask* ThreadPool::m_task [private]

Referenced by Run(), setTask(), and ThreadPool().

7.28.8.10 int ThreadPool::m_total_running_threads [private]

Referenced by Run(), WaitToComplete(), and WakeUp().

7.28.8.11 bool* ThreadPool::m_work [private]

Referenced by CreateThreads(), Run(), and WakeUp().

7.28.8.12 ThreadPool * ThreadPool::pInstance = NULL [static, private]

Referenced by GetInstance().

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

- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.28.8.14

7.29 TimeStep Class Reference

#include <dns.h>

7.29.1 Public Member Functions

- TimeStep (Real dt, Real dtmin, Real dtmax, Real dT, Real CFLmin, Real CFLmax)
- bool adjust (Real CFL, bool verbose=true)
- Real CFL () const
- int n () const
- Real dt () const
- Real dT () const
- operator Real () const

7.29.2 Private Attributes

- int n_
- int nmin_
- int nmax_
- Real dT_
- Real CFLmin_
- Real CFL_
- Real CFLmax_

7.29.3 Constructor & Destructor Documentation

7.29.3.1 TimeStep::TimeStep (Real *dt*, Real *dtmin*, Real *dtmax*, Real *dT*, Real *CFLmin*, Real *CFLmax*)

References n_, nmax_, and nmin_.

```
566   :
567   n_(int(dT/dt + EPSILON)),
568   nmin_(int(dT/dtmax + EPSILON)),
569   nmax_(int(dT/dtmin + EPSILON)),
570   dT_(dT),
571   CFLmin_(CFLmin),
572   CFL_((CFLmax+CFLmin)/2), // will take on meaninful value after first adjust
573   CFLmax_(CFLmax)
574 {
575   assert(dt>0 && dt<=dT);
576   assert(dt>=dtmin && dt<=dtmax);
577   assert(n_>=nmin_ &&  n_<=nmax_);
578   assert(nmin_ > 0);
579 }
```

7.29.4 Member Function Documentation

7.29.4.1 bool TimeStep::adjust (Real *CFL*, bool *verbose* = true)

References CFL_, CFLmax_, CFLmin_, dT_, n(), n_, nmax_, and nmin_.

```
585                                 {
586   CFL_ = CFL;
587   if (CFLmin_ <= CFL && CFL <= CFLmax_)
588     return false;
589
590   // Potential new value of n. Potential new value of CFL is kept in CFL.
591   int n = n_;
592
593   // Decrease CFL by increasing n, decreasing dt = T/n
594   // CFL' == CFL * dt'/dt == CFL * n/n' == CFL * n/(n+1);
595   while (CFL > CFLmax_ && n < nmax_) {
596     CFL *= Real(n)/Real(n+1);
597     ++n;
598   }
599
600   // Increase CFL by decreasing n, increasing dt = T/n
601   // CFL' == CFL * dt'/dt == CFL * n/n' == CFL * n/(n-1);
602   while (CFL < CFLmin_ && n > nmin_) {
603     CFL *= Real(n)/Real(n-1);
604     --n;
605   }
606
607   // Check to see if above loops exited before getting CFL in proper range
608   // or if somehow n got bigger than nmax (latter should never happen).
609   if (CFL > CFLmax_ || n > nmax_) {
610     CFL *= Real(n)/Real(nmax_);
611     n = nmax_;
612     cerr << "TimeStep::adjust(CFL) : dt bottomed out at\n"
613          << " dt  == " << dT_/n << endl
614          << " CFL == " << CFL  << endl
615          << " n   == " << n <<endl;
616   }
617   else if (CFL < CFLmin_ || n < nmin_) {
618     CFL *= Real(n)/Real(nmin_);
619     n = nmin_;
620     cerr << "TimeStep::adjust(CFL) : dt topped out at\n"
621          << " dt  == " << dT_/n << endl
```

```
622         << " CFL == " << CFL  << endl
623         << " n   == " << n << endl;
624   }
625
626   // If final choice for n differs from original n_, reset internal values
627   bool adjustment = (n == n_) ? false : true;
628   if (adjustment && verbose) {
629     cerr << "TimeStep::adjust(CFL) { " << endl;
630     cerr << "   n : " << n_ << " -> " << n << endl;
631     cerr << "  dt : " << dT_/n_ << " -> " << dT_/n << endl;
632     cerr << " CFL : " << CFL_ << " -> " << (n_*CFL_)/n << endl;
633     cerr << "}" << endl;
634     CFL_ *= Real(n_)/Real(n);
635     n_ = n;
636   }
637   return adjustment;
638 }
```

Here is the call graph for this function:

7.29.4.2 Real TimeStep::CFL () const

References CFL_.

```
640 {return CFL_;}
```

7.29.4.3 Real TimeStep::dT () const

References dT_.

```
642 {return dT_;}
```

7.29.4.4 Real TimeStep::dt () const

References dT_, and n_.

```
641 {return dT_/n_;}
```

7.29.4.5 int TimeStep::n () const

References n_.

Referenced by adjust().

```
644 {return n_;}
```

Here is the caller graph for this function:

7.29.4.6 TimeStep::operator Real () const

References dT_, and n_.

```
643 {return dT_/n_;}
```

7.29.5 Member Data Documentation

7.29.5.1 Real TimeStep::CFL_ [private]

Referenced by adjust(), and CFL().

7.29.5.2 Real TimeStep::CFLmax_ [private]

Referenced by adjust().

7.29.5.3 Real TimeStep::CFLmin_ [private]

Referenced by adjust().

7.29.5.4 Real TimeStep::dT_ [private]

Referenced by adjust(), dT(), dt(), and operator Real().

7.29.5.5 int TimeStep::n_ [private]

Referenced by adjust(), dt(), n(), operator Real(), and TimeStep().

7.29.5.6 int TimeStep::nmax_ [private]

Referenced by adjust(), and TimeStep().

7.29.5.7 int TimeStep::nmin_ [private]

Referenced by adjust(), and TimeStep().

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

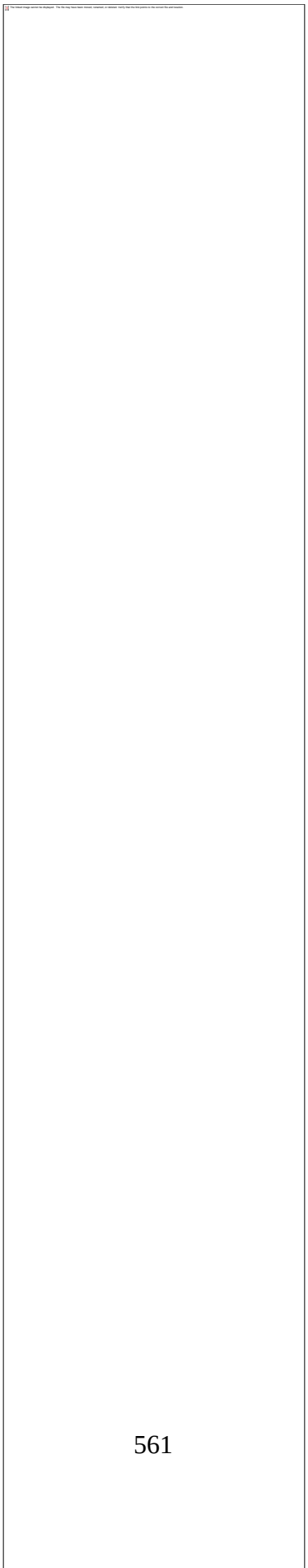
- /_cuda/channelflow-0.9.22/channelflow/dns.h
- /_cuda/channelflow-0.9.22/channelflow/dns.cpp

7.29.5.9

7.30 TurbStats Class Reference

#include <turbstats.h>

Collaboration diagram for TurbStats:



7.30.1 Public Member Functions

- TurbStats ()
- TurbStats (const std::string &filebase)
- TurbStats (const ChebyCoeff &Ubase, Real nu)
- void reset ()
- void addData (FlowField &ubase, FlowField &tmp)
- void msave (const std::string &filebase, bool wallunits=false) const
- ChebyCoeff U () const
- ChebyCoeff dUdy () const
- ChebyCoeff Ubase () const
- ChebyCoeff ubase () const
- ChebyCoeff uu () const
- ChebyCoeff uv () const
- ChebyCoeff uw () const
- ChebyCoeff vv () const
- ChebyCoeff vw () const
- ChebyCoeff ww () const
- Real ustar () const
- Real parabolicReynolds () const
- Real bulkReynolds () const
- Real centerlineReynolds () const
- Vector yplus () const

7.30.2 Private Attributes

- int count_
- Real nu_
- ChebyCoeff Ubase_
- ChebyCoeff ubase_
- ChebyCoeff U_
- ChebyCoeff uu_
- ChebyCoeff uv_
- ChebyCoeff uw_
- ChebyCoeff vv_
- ChebyCoeff vw_
- ChebyCoeff ww_

7.30.3 Constructor & Destructor Documentation

7.30.3.1 TurbStats::TurbStats ()

```
36  :
```

```
37   count_(0),
38   nu_(0)
39 {}
```

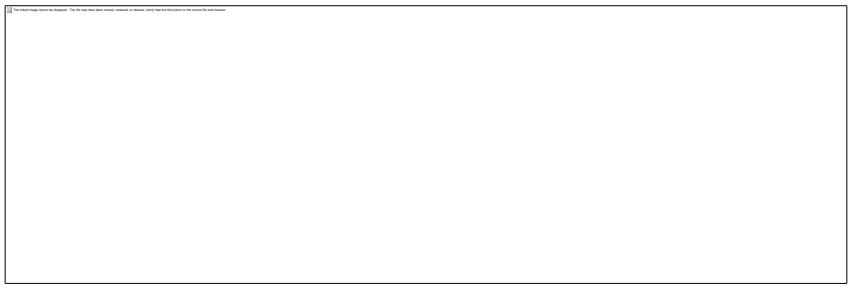
7.30.3.2 TurbStats::TurbStats (const std::string & *filebase*)

7.30.3.3 TurbStats::TurbStats (const ChebyCoeff & *Ubase*, Real *nu*)

References ChebyCoeff::makePhysical(), ChebyCoeff::numModes(), Spectral, ChebyCoeff::state(), and Ubase_.

```
92   :
93   count_(0),
94   nu_(nu),
95   Ubase_(Ubase),
96   ubase_(Ubase.numModes(),Ubase.a(),Ubase.b(),Physical),
97   U_(Ubase.numModes(),Ubase.a(),Ubase.b(),Physical),
98   uu_(Ubase.numModes(),Ubase.a(),Ubase.b(),Physical),
99   uv_(Ubase.numModes(),Ubase.a(),Ubase.b(),Physical),
100   uw_(Ubase.numModes(),Ubase.a(),Ubase.b(),Physical),
101   vv_(Ubase.numModes(),Ubase.a(),Ubase.b(),Physical),
102   vw_(Ubase.numModes(),Ubase.a(),Ubase.b(),Physical),
103   ww_(Ubase.numModes(),Ubase.a(),Ubase.b(),Physical)
104 {
105   if (Ubase_.state() == Spectral) {
106     ChebyTransform t(Ubase_.numModes());
107     Ubase_.makePhysical(t);
108   }
109
110 }
```

Here is the call graph for this function:



7.30.4 Member Function Documentation

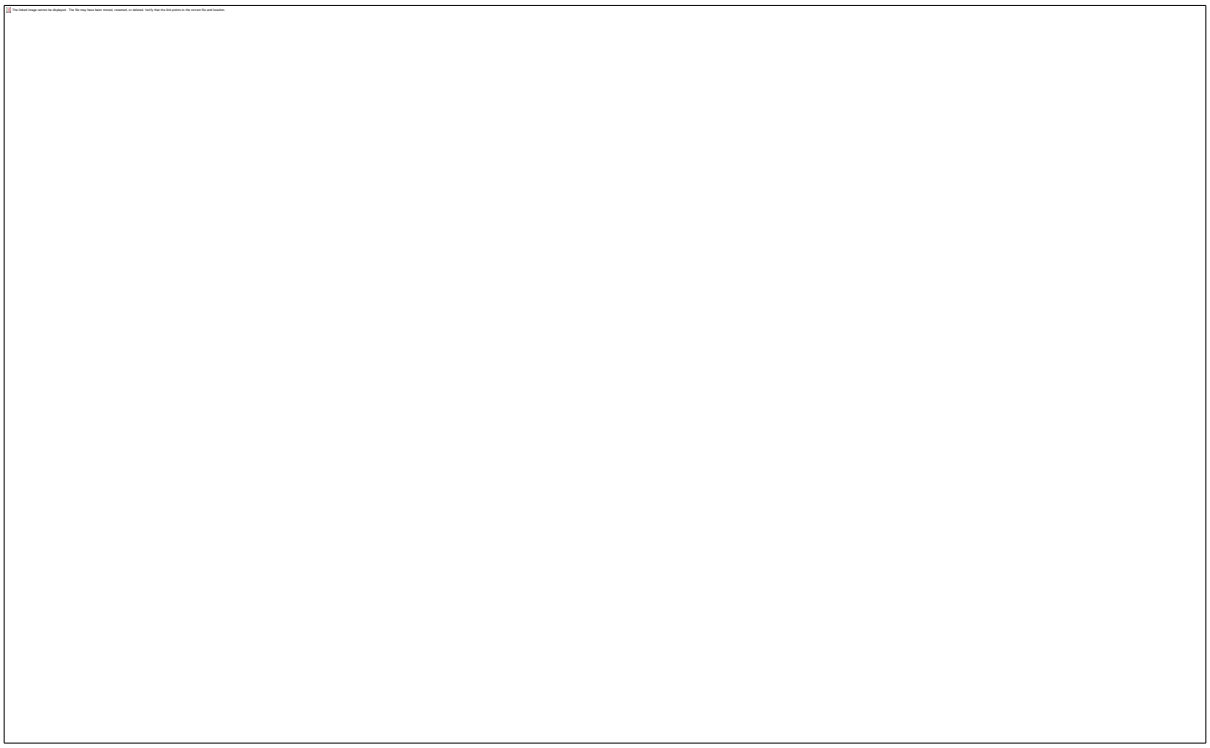
7.30.4.1 void TurbStats::addData (FlowField & *ubase*, FlowField & *tmp*)

References FlowField::cmplx(), count_, FlowField::makePhysical_xz(), FlowField::makePhysical_y(), FlowField::makeSpectral_xz(), FlowField::makeState(), FlowField::numXgridpts(), FlowField::numYgridpts(), FlowField::numZgridpts(), Physical, Re(), FlowField::setState(), FlowField::setToZero(), U_, Ubase_, ubase_, uu_, uv_, uw_, vv_, vw_, ww_, FlowField::xzstate(), and FlowField::ystate().

```
125                                     {
126   fieldstate xzstate = un.xzstate();
127   fieldstate ystate = un.ystate();
128   int Nx = un.numXgridpts();
129   int Ny = un.numYgridpts();
130   int Nz = un.numZgridpts();
131
132   // Add 0,0 profile to mean velocity
133   un.makeSpectral_xz();
134   un.makePhysical_y();
135
136   for (int ny=0; ny<Ny; ++ny) {
137     ubase_[ny] += Re(un.cmplx(0,ny,0,0));
138     U_[ny] += Re(un.cmplx(0,ny,0,0)) + Ubase_[ny];
139   }
140
141   // Compute uu(y).
142   un.makePhysical_xz();
143   tmp.setToZero();
144   tmp.setState(Physical, Physical);
145
146   // tmp[0]=uu, tmp[1]=vv, tmp[2]=ww
147   for (int nx=0; nx<Nx; ++nx)
148     for (int nz=0; nz<Nz; ++nz)
149       for (int ny=0; ny<Ny; ++ny) {
150         // cache thrash
151         Real u = un(nx,ny,nz,0) + Ubase_[ny];
152         Real v = un(nx,ny,nz,1);
153         Real w = un(nx,ny,nz,2);
154         tmp(nx,ny,nz,0) = u*u;
155         tmp(nx,ny,nz,1) = u*v;
156         tmp(nx,ny,nz,2) = u*w;
157         tmp(nx,ny,nz,3) = v*v;
158         tmp(nx,ny,nz,4) = v*w;
159         tmp(nx,ny,nz,5) = w*w;
160       }
161   tmp.makeSpectral_xz();
```

```
162   for (int ny=0; ny<Ny; ++ny) {
163     uu_[ny] += Re(tmp.cmplx(0,ny,0,0));
164     uv_[ny] += Re(tmp.cmplx(0,ny,0,1));
165     uw_[ny] += Re(tmp.cmplx(0,ny,0,2));
166     vv_[ny] += Re(tmp.cmplx(0,ny,0,3));
167     vw_[ny] += Re(tmp.cmplx(0,ny,0,4));
168     ww_[ny] += Re(tmp.cmplx(0,ny,0,5));
169   }
170   ++count_;
171   un.makeState(xzstate,ystate);
172 }
```

Here is the call graph for this function:



7.30.4.2 Real TurbStats::bulkReynolds () const

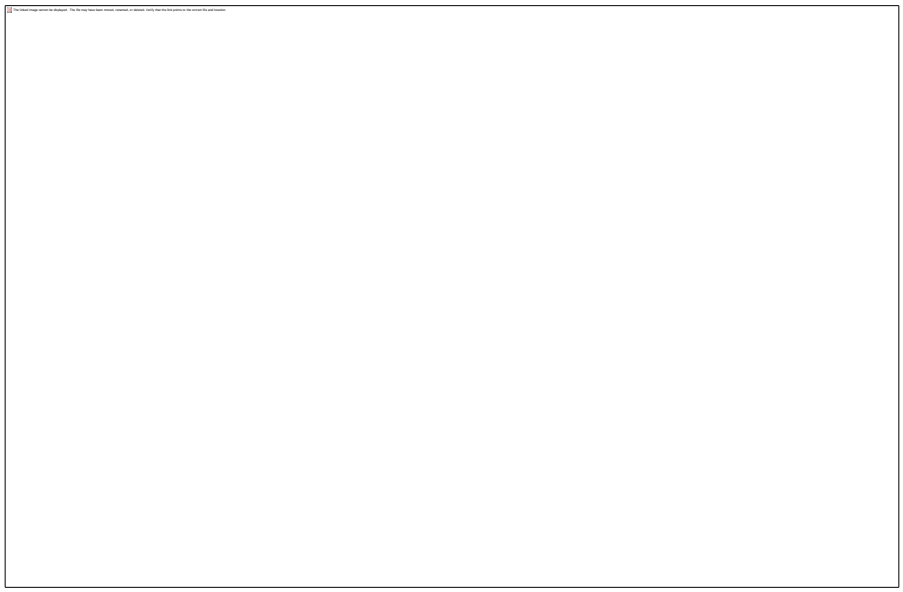
References ChebyCoeff::a(), ChebyCoeff::b(), count_, ChebyCoeff::makeSpectral(), ChebyCoeff::mean(), ChebyCoeff::N(), nu_, U(), and U_.

Referenced by parabolicReynolds().

```
185                                 {
186   // calculate bulk velocity
187   ChebyCoeff U = U_;
188   U *= 1.0/count_;
```

```
189   ChebyTransform trans(U.N());
190   U.makeSpectral(trans);
191   Real Ubulk = U.mean();
192   return 0.5*(U.b() - U.a())*Ubulk/nu_;
193 }
```

Here is the call graph for this function:



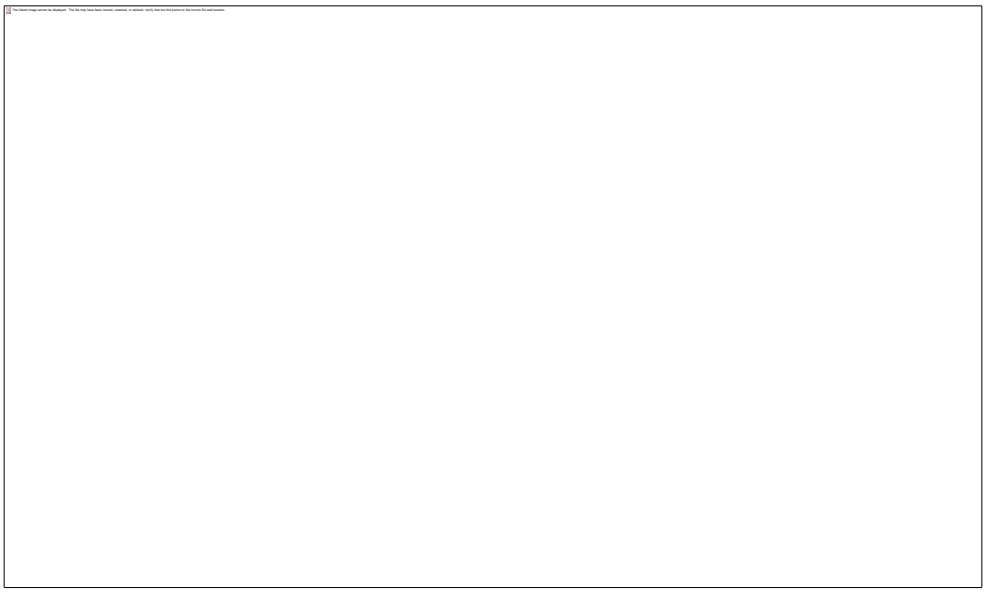
Here is the caller graph for this function:

7.30.4.3 Real TurbStats::centerlineReynolds () const

References ChebyCoeff::a(), ChebyCoeff::b(), count_, ChebyCoeff::eval(), ChebyCoeff::makeSpectral(), ChebyCoeff::N(), nu_, U(), and U_.

```
199                                 {
200   ChebyCoeff U = U_;
201   U *= 1.0/count_;
202   ChebyTransform trans(U.N());
203   U.makeSpectral(trans);
204   Real center = 0.5*(U.a() + U.b());
205   Real Ucenter = U.eval(center);
206   return 0.5*(U.b() - U.a())*Ucenter/nu_;
207 }
```

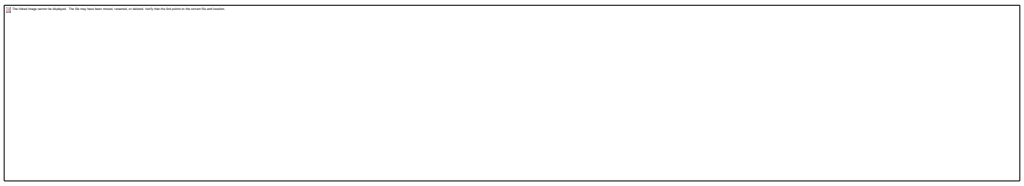
Here is the call graph for this function:



7.30.4.4 ChebyCoeff TurbStats::dUdy () const

Referenced by ustar().

Here is the caller graph for this function:



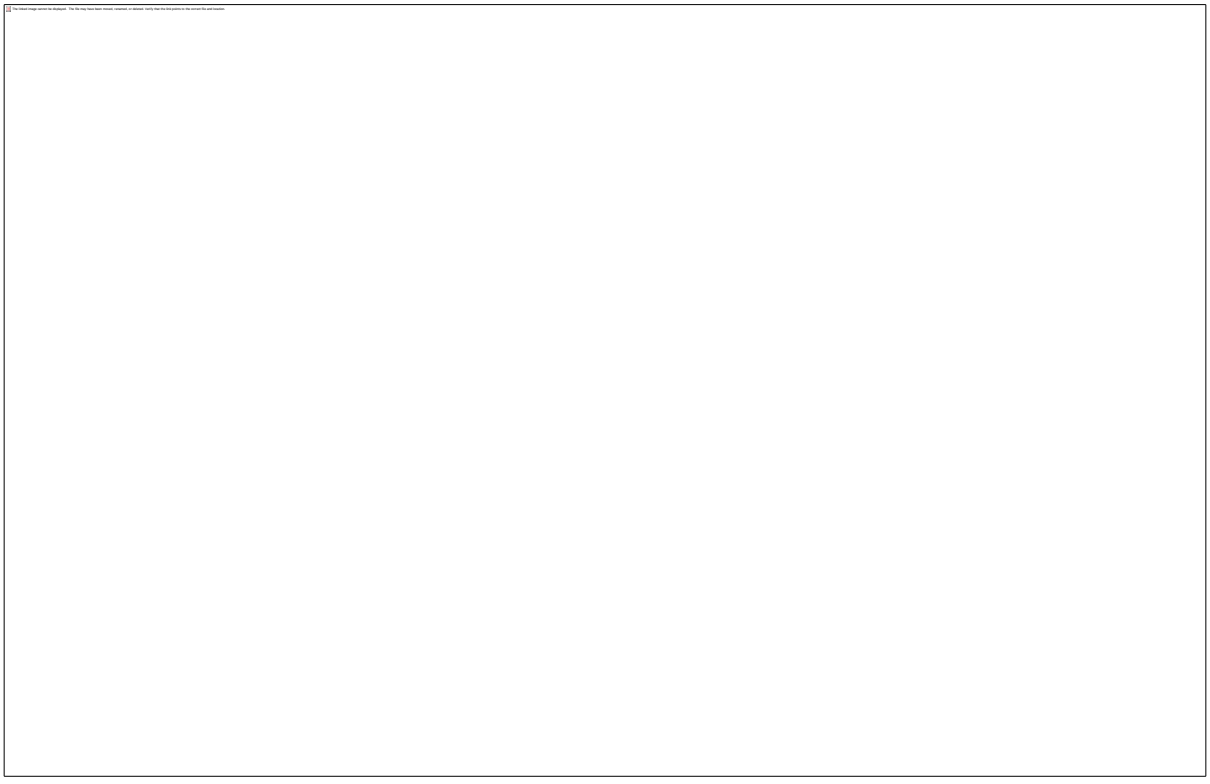
7.30.4.5 void TurbStats::msave (const std::string & *filebase*, bool *wallunits* = false) const

References ChebyCoeff::a(), ChebyCoeff::b(), count_, nu_, ChebyCoeff::numModes(), REAL_DIGITS, square(), U_, Ubase_, ubase_, ustar(), uu_, uv_, uw_, vv_, vw_, and ww_.

```
271                                                 {
272   string filename(filebase);
273   filename += string(".asc");
274   ofstream os(filename.c_str());
275   os << setprecision(REAL_DIGITS);
276   char s = ' ';
277   int Ny = uu_.numModes();
278   Real u_star = ustar();
279 
280   os << "% Ny a b nu datacount wallunits? ustar\n";
```

```
281
282   os << '%' << Ny <<s<< uu_.a() <<s<< uu_.b() <<s<< nu_ <<s<< count_
283      <<s<< wallunits <<s<< u_star << '\n';
284
285   Real c = (wallunits) ? 1.0/square(u_star) : 1.0;
286   Real d = (wallunits) ? 1.0/u_star : 1.0;
287   c /= count_;
288   d /= count_;
289   for (int ny=0; ny<Ny; ++ny) {
290     os << c*(uu_[ny] - square(U_[ny])/count_) << s
291        << c*uv_[ny] << s
292        << c*uw_[ny] << s
293        << c*vv_[ny] << s
294        << c*vw_[ny] << s
295        << c*ww_[ny] << s
296        << d*U_[ny] << s
297        << d*ubase_[ny] << s
298        << d*Ubase_[ny] << '\n';
299   }
300 }
```

Here is the call graph for this function:

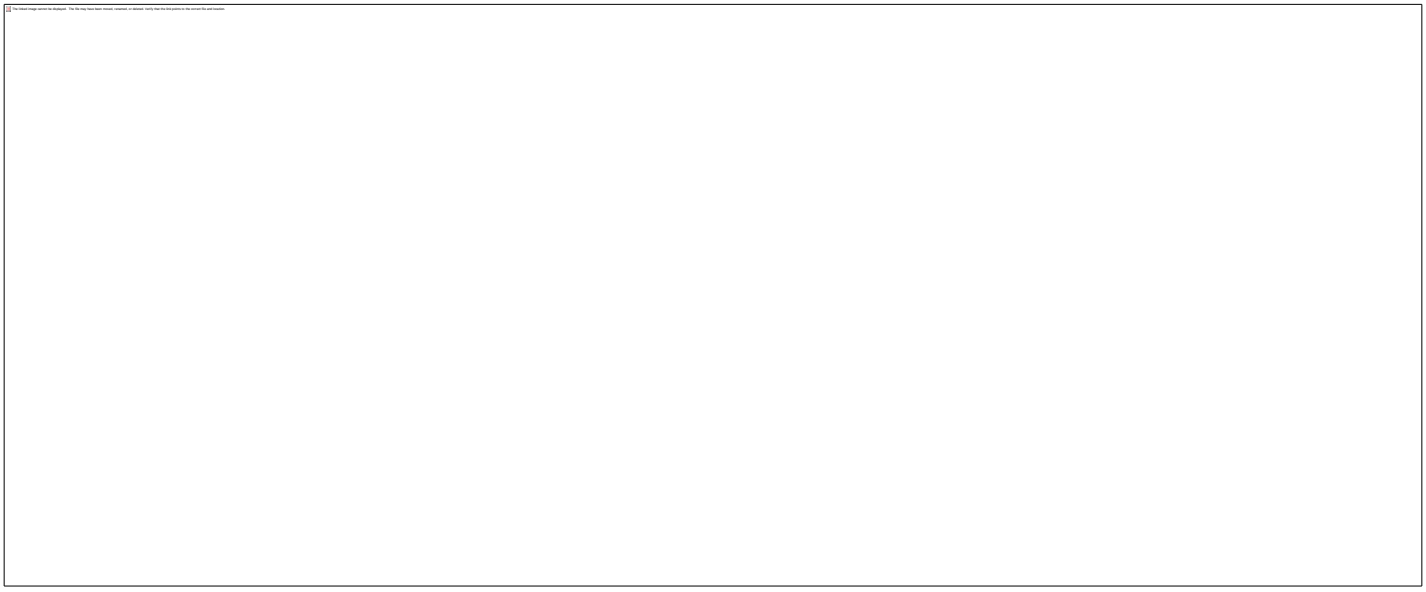


7.30.4.6 Real TurbStats::parabolicReynolds () const

References bulkReynolds().

```
195                                 {
196   return 1.5*bulkReynolds();
197 }
```

Here is the call graph for this function:



7.30.4.7 void TurbStats::reset ()

References count_, ChebyCoeff::setToZero(), U_, ubase_, Ubase_, uu_, uv_, uw_, vv_, vw_, and ww_.

```
112                     {
113   count_=0;
114   Ubase_.setToZero();
115   ubase_.setToZero();
116   U_.setToZero();
117   uu_.setToZero();
118   uv_.setToZero();
119   uw_.setToZero();
120   vv_.setToZero();
121   vw_.setToZero();
122   ww_.setToZero();
123 }
```

Here is the call graph for this function:

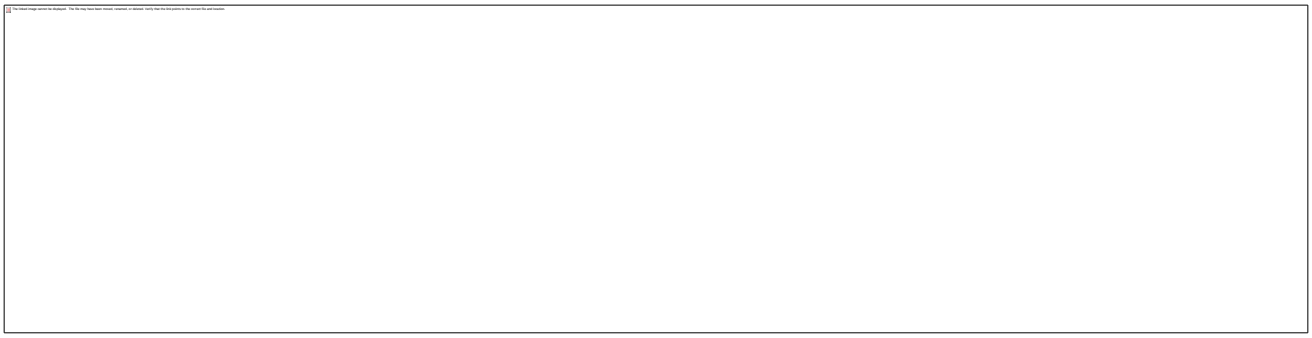
7.30.4.8 ChebyCoeff TurbStats::U () const

References count_, and U_.

Referenced by bulkReynolds(), centerlineReynolds(), and ustar().

```
222                         {
223   ChebyCoeff rtn(U_);
224   rtn *= 1.0/count_;
225   return rtn;
226 }
```

Here is the caller graph for this function:



7.30.4.9 ChebyCoeff TurbStats::ubase () const

References count_, and ubase_.

```
228                           {
229   ChebyCoeff rtn(ubase_);
230   rtn *= 1.0/count_;
231   return rtn;
232 }
```

7.30.4.10 ChebyCoeff TurbStats::Ubase () const

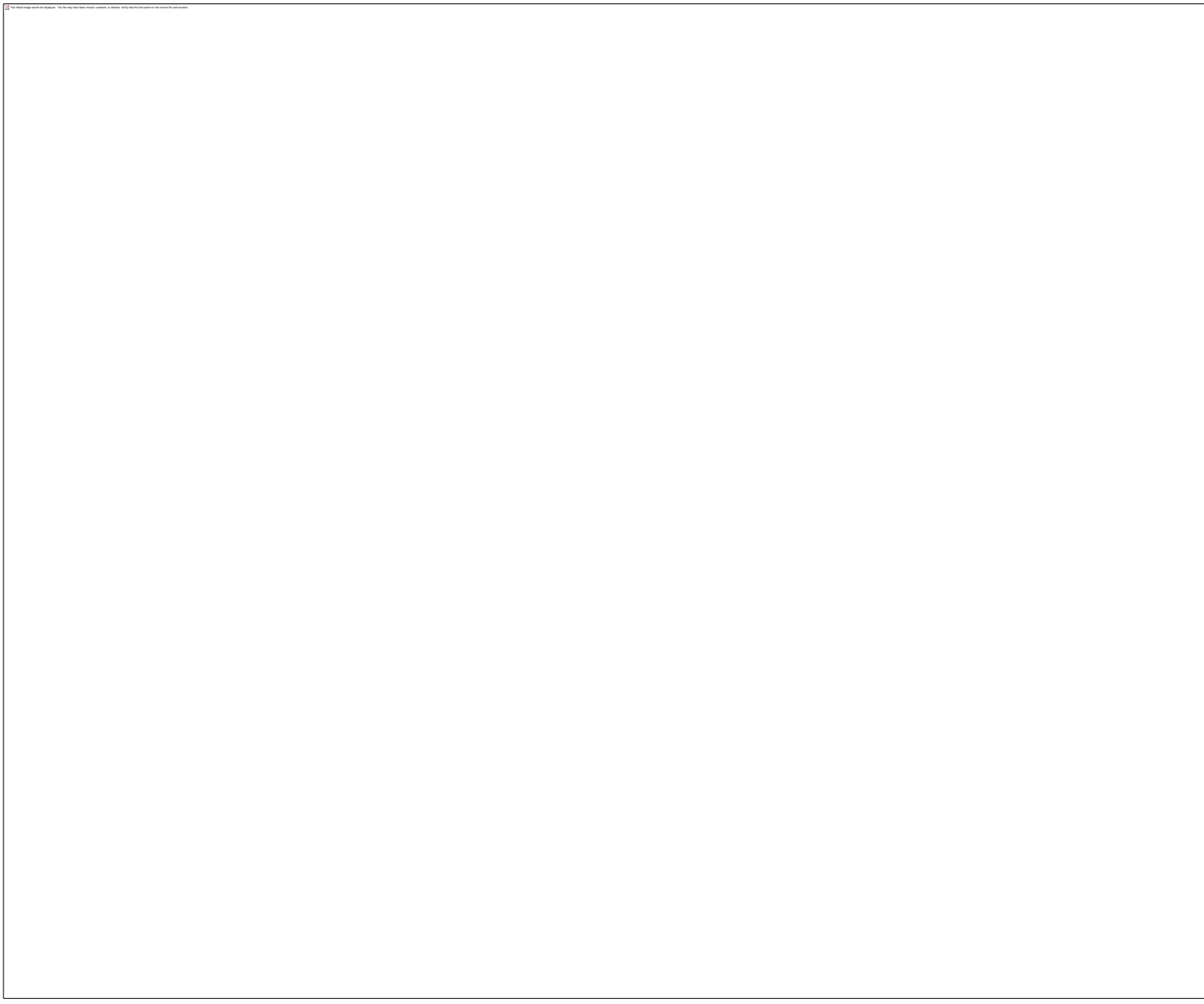
7.30.4.11 Real TurbStats::ustar () const

References abs(), count_, diff(), dUdy(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), ChebyCoeff::makePhysical(), ChebyCoeff::makeSpectral(), ChebyCoeff::N(), nu_, U(), and U_.

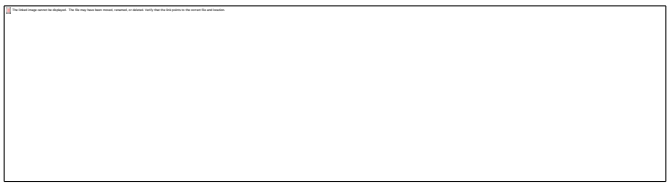
Referenced by msave(), and yplus().

```
174                         {
175   ChebyCoeff U = U_;
176   U *= 1.0/count_;
177   ChebyTransform trans(U.N());
178   U.makeSpectral(trans);
179   ChebyCoeff dUdy = diff(U);
180   dUdy.makePhysical(trans);
181   return sqrt(nu_/2*(abs(dUdy.eval_a() + abs(dUdy.eval_b()))));
182 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

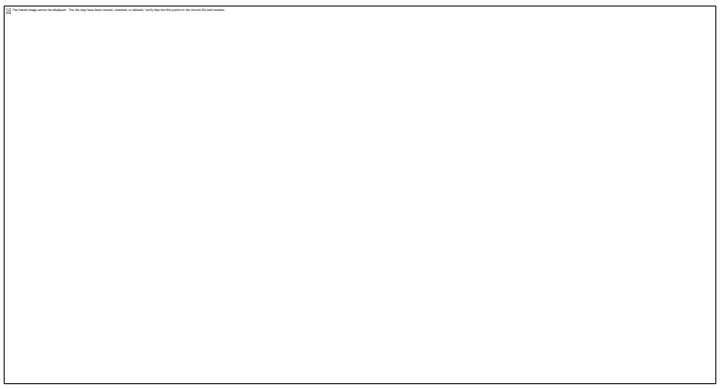


7.30.4.12 ChebyCoeff TurbStats::uu () const

References ChebyCoeff::a(), ChebyCoeff::b(), count_, ChebyCoeff::numModes(), Physical, square(), U_, and uu_.

```
234                             {
235   int Ny = uu_.numModes();
236   ChebyCoeff rtn(Ny, uu_.a(), uu_.b(), Physical);
237   Real c = 1.0/count_;
238   for (int ny=0; ny<Ny; ++ny)
239     rtn[ny] = c*uu_[ny] - square(c*U_[ny]);
240 
241   return rtn;
242 }
```

Here is the call graph for this function:



7.30.4.13 ChebyCoeff TurbStats::uv () const

References count_, and uv_.

```
244                             {
245   ChebyCoeff rtn = uv_;
246   rtn *= 1.0/count_;
247   return rtn;
248 }
```

7.30.4.14 ChebyCoeff TurbStats::uw () const

References count_, and uw_.

```
250                         {
251   ChebyCoeff rtn = uw_;
252   rtn *= 1.0/count_;
253   return rtn;
254 }
```

7.30.4.15 ChebyCoeff TurbStats::vv () const

References count_, and vv_.

```
255                         {
256   ChebyCoeff rtn = vv_;
257   rtn *= 1.0/count_;
258   return rtn;
259 }
```

7.30.4.16 ChebyCoeff TurbStats::vw () const

References count_, and vw_.

```
260                         {
261   ChebyCoeff rtn = vw_;
262   rtn *= 1.0/count_;
263   return rtn;
264 }
```

7.30.4.17 ChebyCoeff TurbStats::ww () const

References count_, and ww_.

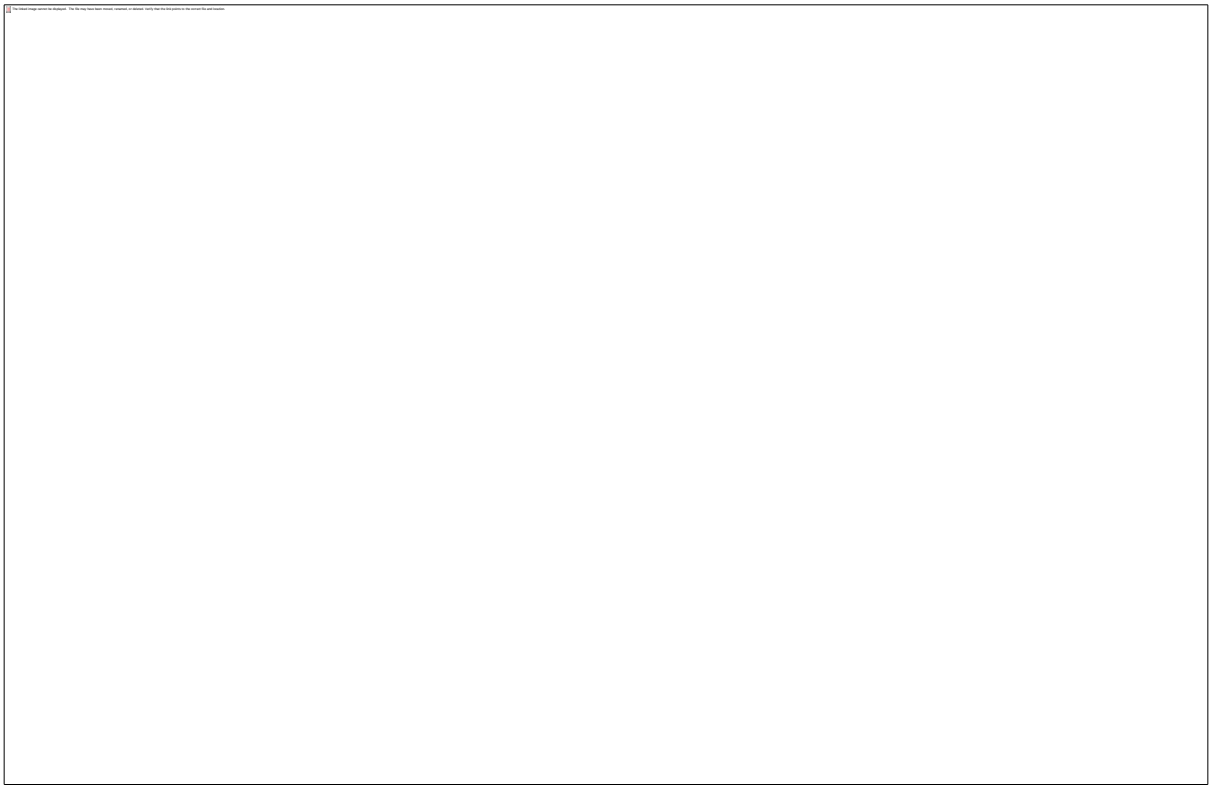
```
265                         {
266   ChebyCoeff rtn = ww_;
267   rtn *= 1.0/count_;
268   return rtn;
269 }
```

7.30.4.18 Vector TurbStats::yplus () const

References ChebyCoeff::a(), ChebyCoeff::b(), chebypoints(), nu_, ChebyCoeff::numModes(), U_, and ustar().

```
209                         {
210   int Ny = U_.numModes();
211   Real a = U_.a();
212   Real b = U_.b();
213   Real c = ustar()/nu_;
214 
215   Vector y = chebypoints(Ny, a, b);
216   Vector yp(Ny);
217   for (int ny=0; ny<Ny; ++ny)
218     yp[ny] = c*(y[ny] - a);
219   return yp;
220 }
```

Here is the call graph for this function:



7.30.5 Member Data Documentation

7.30.5.1 int TurbStats::count_ [private]

Referenced by addData(), bulkReynolds(), centerlineReynolds(), msave(), reset(), U(), ubase(), ustar(), uu(), uv(), uw(), vv(), vw(), and ww().

7.30.5.2 Real TurbStats::nu_ [private]

Referenced by bulkReynolds(), centerlineReynolds(), msave(), ustar(), and yplus().

7.30.5.3 ChebyCoeff TurbStats::U_ [private]

Referenced by addData(), bulkReynolds(), centerlineReynolds(), msave(), reset(), U(), ustar(), uu(), and yplus().

7.30.5.4 ChebyCoeff TurbStats::ubase_ [private]

Referenced by addData(), msave(), reset(), and ubase().

7.30.5.5 ChebyCoeff TurbStats::Ubase_ [private]

Referenced by addData(), msave(), reset(), and TurbStats().

7.30.5.6 ChebyCoeff TurbStats::uu_ [private]

Referenced by addData(), msave(), reset(), and uu().

7.30.5.7 ChebyCoeff TurbStats::uv_ [private]

Referenced by addData(), msave(), reset(), and uv().

7.30.5.8 ChebyCoeff TurbStats::uw_ [private]

Referenced by addData(), msave(), reset(), and uw().

7.30.5.9 ChebyCoeff TurbStats::vv_ [private]

Referenced by addData(), msave(), reset(), and vv().

7.30.5.10 ChebyCoeff TurbStats::vw_ [private]

Referenced by addData(), msave(), reset(), and vw().

7.30.5.11 ChebyCoeff TurbStats::ww_ [private]

Referenced by addData(), msave(), reset(), and ww().

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

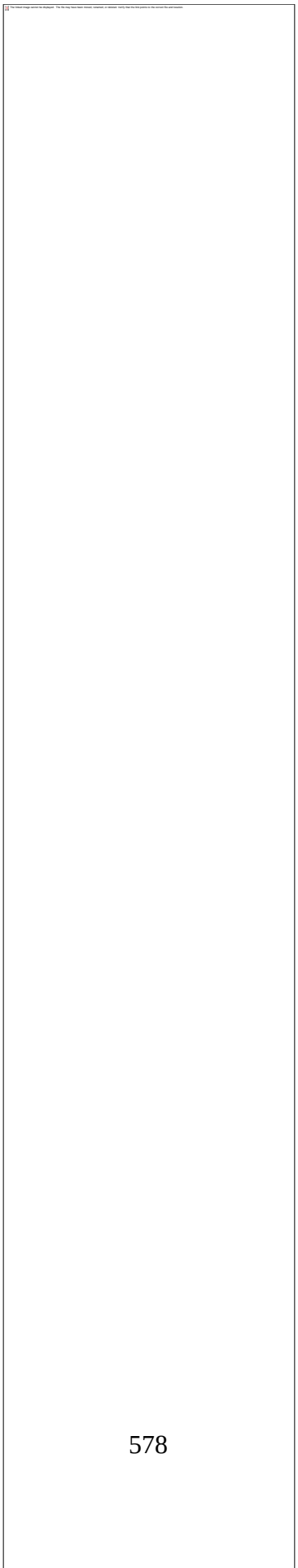
- /_cuda/channelflow-0.9.22/channelflow/turbstats.h
- /_cuda/channelflow-0.9.22/channelflow/turbstats.cpp

7.30.5.13

7.31 Vector Class Reference

#include <vector.h>

Inheritance diagram for Vector:


The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

7.31.1 Public Member Functions

- Vector (int N=0)
- Vector (const Vector &a)
- Vector (const std::string &filename)
- virtual ~Vector ()
- void resize (int N)
- void setToZero ()
- virtual void randomize ()
- Vector & operator= (const Vector &a)
- Real & operator[] (int i)
- Real operator[] (int i) const
- Real & operator() (int i)
- Real operator() (int i) const
- Vector & operator*= (Real c)
- Vector & operator/= (Real c)
- Vector & operator+= (Real c)
- Vector & operator-= (Real c)
- Vector & operator+= (const Vector &c)
- Vector & operator-= (const Vector &c)
- Vector & dottimes (const Vector &c)
- Vector & dotdivide (const Vector &c)
- Vector & abs ()
- Vector subvector (int offset, int N) const
- Vector modularSubvector (int offset, int N) const
- int N () const
- int length () const
- const Real * pointer () const
- Real * pointer ()
- void save (const std::string &filebase) const

7.31.2 Protected Attributes

- Real * data_
- int N_

7.31.3 Friends

- void swap (Vector &f, Vector &g)

7.31.4 Constructor & Destructor Documentation

7.31.4.1 Vector::Vector (int *N* = 0)

References data_, and N_.

```
38                  :
39   data_((Real*) fftw_malloc(N*sizeof(Real))),
40   N_(N)
41 {
42   //cerr << "Vector ctor " << long(data_) << endl;
43   assert(data_ != 0);
44   Real* dptr = data_;
45   for (int i=0; i<N_; ++i)
46     *dptr++ = 0.0;
47 }
```

7.31.4.2 Vector::Vector (const Vector & *a*)

References data_, and N_.

```
49                          :
50   data_((Real*) fftw_malloc(a.N_*sizeof(Real))),
51   N_(a.N_)
52 {
53   //cerr << "Vector ctor " << long(data_) << endl;
54   assert(data_ != 0);
55
56   Real* dptr = data_;
57   Real* aptr = a.data_;
58   for (int i=0; i<N_; ++i)
59     *dptr++ = *aptr++;
60 }
```

7.31.4.3 Vector::Vector (const std::string & *filename*)

References data_, N(), and N_.

```
63                                  :
64   data_(0),
65   N_(0)
66 {
67   //cerr << "Vector ctor " << long(data_) << endl;
68   string filename(filebase);
69   filename += string(".asc");
70
71   //ifstream sis(sizefile.c_str());
72   ifstream is(filename.c_str());
```

```
73
74  // Read in header. Form is "%5 4" for a 5x4 matrix
75  char c;
76  int M,N;
77  is >> c;
78  if (c != '%') {
79    string message("Vector(filebase): bad header in file ");
80    message += filename;
81    cerr << message << endl;
82    assert(false);
83  }
84  is >> M >> N;
85
86  assert(M == 1 || N == 1);
87  N_ = (M > N) ? M : N;
88
89  data_ = (Real*) fftw_malloc(N_*sizeof(Real));
90
91  //char BUFFER[256];
92  //is.getline(BUFFER, 256); // Finsh off the first line
93  //is.getline(BUFFER, 256); // Get the comment string.
94
95  Real* dptr = data_;
96  Real* end = data_ + N_;
97  while (dptr < end)
98    is >> *dptr++;
99  is.close();
100 }
```

Here is the call graph for this function:

7.31.4.4 Vector::~Vector () [virtual]

References data_.

```
130             {
131  //cerr << "Vector dtor " << long(data_) << flush;
132  fftw_free(data_);
133  //cerr << "done" << endl;
134  data_=0;
135 }
```

7.31.5 Member Function Documentation

7.31.5.1 Vector & Vector::abs ()

References data_, and N_.

Referenced by abs().

```
197                     {
198   for (int i=0; i<N_; ++i)
199     data_[i] = fabs(data_[i]);
200   return *this;
201 }
```

Here is the caller graph for this function:

7.31.5.2 Vector & Vector::dotdivide (const Vector & *c*)

References data_, and N_.

```
190                                 {
191   assert(a.N_ == N_);
192   for (int i=0; i<N_; ++i)
193     data_[i] /= a.data_[i];
194   return *this;
195 }
```

7.31.5.3 Vector & Vector::dottimes (const Vector & *c*)

References data_, and N_.

```
183                                 {
184   assert(a.N_ == N_);
185   for (int i=0; i<N_; ++i)
186     data_[i] *= a.data_[i];
187   return *this;
188 }
```

7.31.5.4 int Vector::length () const [inline]

References N_.

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), diff(), diff2(), DNSAlgorithm::DNSAlgorithm(), dotdivide(), dottimes(), ChebyCoeff::eval(), integrate(), L1Dist(), L1Norm(), L2Dist(), L2Dist2(), L2Norm(), L2Norm2(), ComplexChebyCoeff::length(), LinfDist(), LinfNorm(), maxElemIndex(), mean(), operator*(), operator+(), operator-(), operator<<(), operator==(), HelmholtzSolver::residual(), ComplexChebyCoeff::save(), and HelmholtzSolver::verify().

```
140 {return N_;}
```

Here is the caller graph for this function:

7.31.5.5 Vector Vector::modularSubvector (int *offset*, int *N*) const

References data_, and N_.

```
211                                         {
212   Vector subvec(N);
213   for (int i=0; i<N; ++i)
214     subvec[i] = data_[(i+offset) % N_];
215   return subvec;
216 }
```

7.31.5.6 int Vector::N () const [inline]

Reimplemented in ChebyCoeff.

References N_.

Referenced by Vector().

```
139 {return N_;}
```

Here is the caller graph for this function:

7.31.5.7 Real Vector::operator() (int *i*) const [inline]

References data_, and N_.

```
134                                       {
135   assert(i>=0 && i<N_);
136   return data_[i];
137 }
```

7.31.5.8 Real & Vector::operator() (int *i*) [inline]

References data_, and N_.

```
129                                     {
130   assert(i>=0 && i<N_);
131   return data_[i];
132 }
```

7.31.5.9 Vector & Vector::operator*= (Real *c*)

Reimplemented in ChebyCoeff.

References data_, and N_.

```
144                                {
145   for (int i=0; i<N_; ++i)
146     data_[i] *= c;
147   return *this;
148 }
```

7.31.5.10 Vector & Vector::operator+= (const Vector & *c*)

Reimplemented in ChebyCoeff.

References data_, and N_.

```
169                                       {
170   assert(a.N_ == N_);
171   for (int i=0; i<N_; ++i)
172     data_[i] += a.data_[i];
173   return *this;
174 }
```

7.31.5.11 Vector & Vector::operator+= (Real *c*)

References data_, and N_.

```
157                                {
158   for (int i=0; i<N_; ++i)
159     data_[i] += c;
160   return *this;
161 }
```

7.31.5.12 Vector & Vector::operator-= (const Vector & *c*)

Reimplemented in ChebyCoeff.

References data_, and N_.

```
176                                       {
177   assert(a.N_ == N_);
178   for (int i=0; i<N_; ++i)
179     data_[i] -= a.data_[i];
180   return *this;
181 }
```

7.31.5.13 Vector & Vector::operator-= (Real *c*)

References data_, and N_.

```
163                             {
164   for (int i=0; i<N_; ++i)
165     data_[i] -= c;
166   return *this;
167 }
```

7.31.5.14 Vector & Vector::operator/= (Real *c*)

References data_, and N_.

```
150                             {
151   Real cinv = 1.0/c;
152   for (int i=0; i<N_; ++i)
153     data_[i] *= cinv;
154   return *this;
155 }
```

7.31.5.15 Vector & Vector::operator= (const Vector & *a*)

References data_, and N_.

```
218                                 {
219   if (data_ != a.data_) {
220     if (N_ != a.N_) {
221       //cout << "delete in operator= on " << long(data_) << endl;
222       fftw_free(data_);
223       data_ = (Real*) fftw_malloc(a.N_*sizeof(Real));
224       N_ = a.N_;
225       assert(data_ != 0);
226     }
227     Real* dptr = data_;
228     Real* aptr = a.data_;
229     for (int i=0; i<N_; ++i)
230       *dptr++ = *aptr++;
231   }
232   return *this;
233 }
```

7.31.5.16 Real Vector::operator[] (int *i*) const [inline]

References data_, and N_.

```
125                                 {
126   assert(i>=0 && i<N_);
127   return data_[i];
128 }
```

7.31.5.17 Real & Vector::operator[] (int *i*) [inline]

References data_, and N_.

```
120                                 {
121   assert(i>=0 && i<N_);
122   return data_[i];
123 }
```

7.31.5.18 Real * Vector::pointer () [inline]

References data_.

```
141 {return data_;}
```

7.31.5.19 const Real * Vector::pointer () const [inline]

References data_.

```
142 {return data_;}
```

7.31.5.20 void Vector::randomize () [virtual]

References data_, N_, and randomReal().

```
120                       {
121   for (int i=0; i<N_; ++i)
122     data_[i] = randomReal();
123 }
```

Here is the call graph for this function:

7.31.5.21 void Vector::resize (int *N*)

References data_, lesser(), and N_.

Referenced by ChebyCoeff::binaryLoad(), ChebyCoeff::ChebyCoeff(), ComplexChebyCoeff::ComplexChebyCoeff(), diff(), ChebyCoeff::eval(), integrate(), and ComplexChebyCoeff::resize().

```
102                         {
103   if (newN != N_) {
104     Real* newdata = (Real*) fftw_malloc(newN*sizeof(Real));
105     //cerr << "Vector resize " << long(data_) << " -> " << long(newdata) << endl;
106     assert(newdata != 0);
107     int shortN = lesser(N_, newN);
108     int i; // FOR-SCOPE BUG
109     for (i=0; i<shortN; ++i)
110       newdata[i] = data_[i];
111     for (i=shortN; i<newN; ++i)
112       newdata[i] = 0.0;
113     fftw_free(data_);
114     data_ = newdata;
115     N_ = newN;
116   }
117 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

7.31.5.22 void Vector::save (const std::string & *filebase*) const

References data_, N_, REAL_DIGITS, and REAL_IOWIDTH.

```
235                                     {
236   string filename(filebase);
237   filename += string(".asc");
238   ofstream os(filename.c_str());
239
240   os << scientific << setprecision(REAL_DIGITS);
241   os << "% " << N_ << " 1\n";
242   for (int i=0; i<N_; ++i)
243     os << setw(REAL_IOWIDTH) << data_[i] << '\n';
244   os.close();
245 }
```

7.31.5.23 void Vector::setToZero ()

Reimplemented in ChebyCoeff.

References data_, and N_.

```
124                     {
125   for (int i=0; i<N_; ++i)
126     data_[i] = 0.0;
127 }
```

7.31.5.24 Vector Vector::subvector (int *offset*, int *N*) const

References data_, and N_.

```
203                                     {
204   Vector subvec(N);
205   assert(N+offset <= N_);
206   for (int i=0; i<N; ++i)
207     subvec[i] = data_[i+offset];
208   return subvec;
209 }
```

7.31.6 Friends And Related Function Documentation

7.31.6.1 void swap (Vector & *f*, Vector & *g*) [friend]

Reimplemented in ChebyCoeff.

7.31.7 Member Data Documentation

7.31.7.1 Real* Vector::data_ [protected]

Referenced by abs(), ChebyCoeff::binaryDump(), ChebyCoeff::binaryLoad(), ChebyCoeff::ChebyCoeff(), ChebyCoeff::chebyfft(), dotdivide(), dottimes(), ChebyCoeff::eval(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), ChebyCoeff::fill(), ChebyCoeff::ichebyfft(), ChebyCoeff::interpolate(), ChebyCoeff::mean(), modularSubvector(), operator()(), operator*=(), ChebyCoeff::operator*=(), operator+=(), ChebyCoeff::operator+=(), operator-=(), ChebyCoeff::operator-=(), operator/=(), operator=(), operator[](), pointer(), randomize(), ChebyCoeff::randomize(), ChebyCoeff::reflect(), resize(), save(), ChebyCoeff::save(), setToZero(), ChebyCoeff::setToZero(), subvector(), swap(), Vector(), and ~Vector().

7.31.7.2 int Vector::N_ [protected]

Referenced by abs(), ChebyCoeff::binaryDump(), ChebyCoeff::binaryLoad(), ChebyCoeff::ChebyCoeff(), ChebyCoeff::chebyfft(), ChebyCoeff::congruent(), dotdivide(), dottimes(), ChebyCoeff::eval(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), ChebyCoeff::fill(), ChebyCoeff::ichebyfft(), ChebyCoeff::interpolate(), length(), ChebyCoeff::makePhysical(), ChebyCoeff::makeSpectral(), ChebyCoeff::makeState(), ChebyCoeff::mean(), modularSubvector(), N(), ChebyCoeff::N(), ChebyCoeff::numModes(), operator()(), operator*=(), ChebyCoeff::operator*=(), operator+=(), ChebyCoeff::operator+=(), operator-=(), ChebyCoeff::operator-=(), operator/=(), operator=(), operator[](), randomize(), ChebyCoeff::randomize(), ChebyCoeff::reflect(), resize(), save(), ChebyCoeff::save(), setToZero(), ChebyCoeff::setToZero(), subvector(), swap(), and Vector().

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

- /_cuda/channelflow-0.9.22/channelflow/vector.h
- /_cuda/channelflow-0.9.22/channelflow/vector.cpp