THREAD INJECTION TO MAKE DNS CHANNELFLOW RUN IN PARALLEL

Part 2

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 2

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

ABSTRACT

In Part 1 of this report it was noted that the Navier-Stokes equation will be solved for flow systems that are practically important by a combination of improvements in both computer hardware and software. In this work we demonstrate how thread injection can be used to parallelize solutions of the Navier-Stokes equation. The method is a modification of Channelflow which is open source software developed by channelflow.org. 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. Readers requiring more details and assistance are most welcome to contact Mr Igor Grossman at igor.grossman@live.vu.edu.au.

Because the report is voluminous it has been produced as two consecutive fascicles, namely Parts 1 and 2.

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

Table of Contents

THREAD INJECTION TO MAKE DNS CHANNELFLOW RUN IN PARALLEL i
ABSTRACT i
1 File Documentation 6
1.1 /_cuda/channelflow-0.9.22/channelflow/array.h File Reference 6
1.1.1 Classes 6
1.1.2 Typedefs 7
1.1.3 Functions 7
1.1.4 Variables 7
1.1.5 Typedef Documentation 7
1.1.6 Function Documentation 7
1.1.7 Variable Documentation 7
1.2 /_cuda/channelflow-0.9.22/channelflow/bandedtridiag.cpp File Reference 8
1.2.1 Variables 8
1.2.2 Variable Documentation 8
1.3 /_cuda/channelflow-0.9.22/channelflow/bandedtridiag.h File Reference 9
1.3.1 Classes 9
1.3.2 Typedefs 9
1.3.3 Typedef Documentation 9
1.4 /_cuda/channelflow-0.9.22/channelflow/basisfunc.cpp File Reference 10
1.4.1 Functions 10
1.4.2 Function Documentation 11
1.5 /_cuda/channelflow-0.9.22/channelflow/basisfunc.h File Reference 70
1.5.1 Classes 70
1.5.2 Functions 70
1.5.3 Function Documentation 71
1.6 /_cuda/channelflow-0.9.22/channelflow/chebyshev.cpp File Reference 131
1.6.1 Functions 131
1.6.2 Variables 133
1.6.3 Function Documentation 133

1.6.4 Variable Documentation 192
1.7 /_cuda/channelflow-0.9.22/channelflow/chebyshev.h File Reference 193
1.7.1 Classes 193
1.7.2 Enumerations 193
1.7.3 Functions 193
1.7.4 Enumeration Type Documentation 196
1.7.5 Function Documentation 196
1.8 /_cuda/channelflow-0.9.22/channelflow/complexdefs.h File Reference 258
1.8.1 Functions 258
1.8.2 Function Documentation 258
1.9 /_cuda/channelflow-0.9.22/channelflow/diffops.cpp File Reference 260
1.9.1 Functions 261
1.9.2 Function Documentation 262
1.10 /_cuda/channelflow-0.9.22/channelflow/diffops.h File Reference 364
1.10.1 Enumerations 365
1.10.2 Functions 365
1.10.3 Enumeration Type Documentation 366
1.10.4 Function Documentation 366
1.11 /_cuda/channelflow-0.9.22/channelflow/dns.cpp File Reference 467
1.11.1 Functions 467
1.11.2 Variables 467
1.11.3 Function Documentation 467
1.11.4 Variable Documentation 473
1.12 /_cuda/channelflow-0.9.22/channelflow/dns.h File Reference 474
1.12.1 Classes 476
1.12.2 Enumerations 476
1.12.3 Functions 476
1.12.4 Enumeration Type Documentation 477
1.12.5 Function Documentation 478
1.13 /_cuda/channelflow-0.9.22/channelflow/flowfield.cpp File Reference 485
1.13.1 Functions 485
1.13.2 Variables 486

1.13.3 Function Documentation 486
1.13.4 Variable Documentation 542
1.14 /_cuda/channelflow-0.9.22/channelflow/flowfield.h File Reference 543
1.14.1 Classes 543
1.14.2 Functions 543
1.14.3 Function Documentation 544
1.15 /_cuda/channelflow-0.9.22/channelflow/helmholtz.cpp File Reference 606
1.15.1 Variables 606
1.15.2 Variable Documentation 606
1.16 /_cuda/channelflow-0.9.22/channelflow/helmholtz.h File Reference 607
1.16.1 Classes 607
1.17 /_cuda/channelflow-0.9.22/channelflow/mathdefs.cpp File Reference 608
1.17.1 Defines 608
1.17.2 Functions 608
1.17.3 Define Documentation 609
1.17.4 Function Documentation 609
1.18 /_cuda/channelflow-0.9.22/channelflow/mathdefs.h File Reference 624
1.18.1 Defines 624
1.18.2 Typedefs 624
1.18.3 Enumerations 624
1.18.4 Functions 624
1.18.5 Variables 625
1.18.6 Define Documentation 626
1.18.7 Typedef Documentation 626
1.18.8 Enumeration Type Documentation 626
1.18.9 Function Documentation 626
1.18.10 Variable Documentation 650
1.19 /_cuda/channelflow-0.9.22/channelflow/poissonsolver.cpp File Reference 651
1.20 /_cuda/channelflow-0.9.22/channelflow/poissonsolver.h File Reference 652
1.20.1 Classes 653
1.21 /_cuda/channelflow-0.9.22/channelflow/tausolver.cpp File Reference 654
1.21.1 Functions 654

1.21.2 Variables 654
1.21.3 Function Documentation 655
1.21.4 Variable Documentation 657
1.22 /_cuda/channelflow-0.9.22/channelflow/tausolver.h File Reference 658
1.22.1 Classes 659
1.22.2 Functions 659
1.22.3 Function Documentation 659
1.23 /_cuda/channelflow-0.9.22/channelflow/turbstats.cpp File Reference 661
1.24 /_cuda/channelflow-0.9.22/channelflow/turbstats.h File Reference 662
1.24.1 Classes 662
1.25 /_cuda/channelflow-0.9.22/channelflow/utilfuncs.cpp File Reference 663
1.25.1 Functions 663
1.25.2 Function Documentation 663
1.26 /_cuda/channelflow-0.9.22/channelflow/utilfuncs.h File Reference 676
1.26.1 Classes 676
1.26.2 Functions 676
1.26.3 Function Documentation 677
1.27 /_cuda/channelflow-0.9.22/channelflow/vector.cpp File Reference 690
1.27.1 Functions 690
1.27.2 Function Documentation 691
1.28 /_cuda/channelflow-0.9.22/channelflow/vector.h File Reference 702
1.28.1 Classes 703
1.28.2 Functions 703
1.28.3 Function Documentation 704
2 Index 714

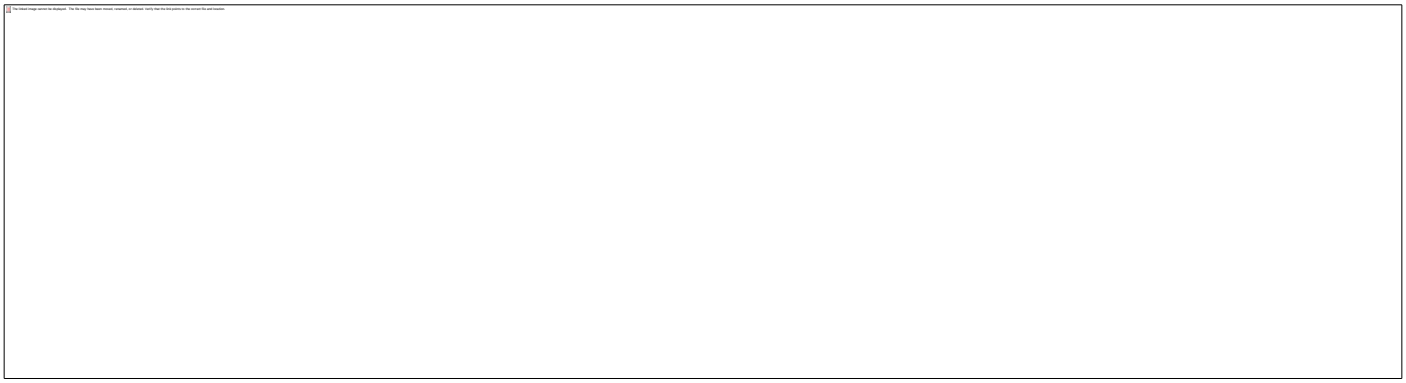
THREAD INJECTION TO MAKE DNS CHANNELFLOW RUN IN PARALLEL

Part 2

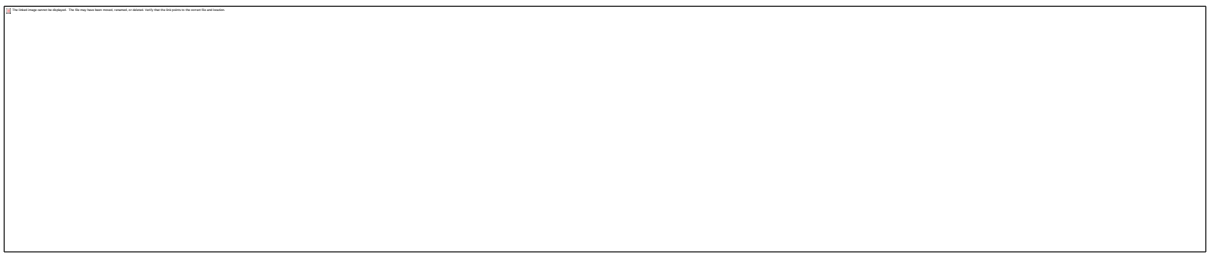
1 File Documentation

1.1 /_cuda/channelflow-0.9.22/channelflow/array.h File Reference

#include <assert.h>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include "channelflow/mathdefs.h"
Include dependency graph for array.h:



This graph shows which files directly or indirectly include this file:



1.1.1 Classes

- class array< T >

1.1.2 Typedefs

- typedef double Real

1.1.3 Functions

- template<class T > std::ostream & operator<< (std::ostream &, const array< T > &a)

1.1.4 Variables

- const int REAL_OUTPUT_DIGITS = 17

1.1.5 Typedef Documentation

1.1.5.1 typedef double Real

1.1.6 Function Documentation

1.1.6.1 template<class T > std::ostream & operator<< (std::ostream & *os*, const array< T > & *a*) [inline]

```
228                                                 {
229   int N = a.length();
230   char seperator=(N<10) ? ' ' : '\n';
231   os << std::setprecision(17);
232   for (int i=0; i<N; ++i)
233     os << a[i] << seperator;
234   return os;
235 }
```

1.1.7 Variable Documentation

1.1.7.1 const int REAL_OUTPUT_DIGITS = 17

1.1.7.2

1.2 /_cuda/channelflow-0.9.22/channelflow/bandedtridiag.cpp File Reference

#include <fstream>
#include <iomanip>
#include "channelflow/bandedtridiag.h"
Include dependency graph for bandedtridiag.cpp:

1.2.1 Variables

- const Real EPSILON = 1e-13

1.2.2 Variable Documentation

1.2.2.1 const Real EPSILON = 1e-13

Referenced by chebyProfile(), chebyUprofile(), BandedTridiag::operator==(), randomProfile(), randomUprofile(), TauSolver::verify(), HelmholtzSolver::verify(), and TauSolver::verify_P_and_v().

1.2.2.2

1.3 /_cuda/channelflow-0.9.22/channelflow/bandedtridiag.h File Reference

#include "channelflow/mathdefs.h"
#include "channelflow/vector.h"
Include dependency graph for bandedtridiag.h:

This graph shows which files directly or indirectly include this file:

1.3.1 Classes

- class BandedTridiag

1.3.2 Typedefs

- typedef double Real

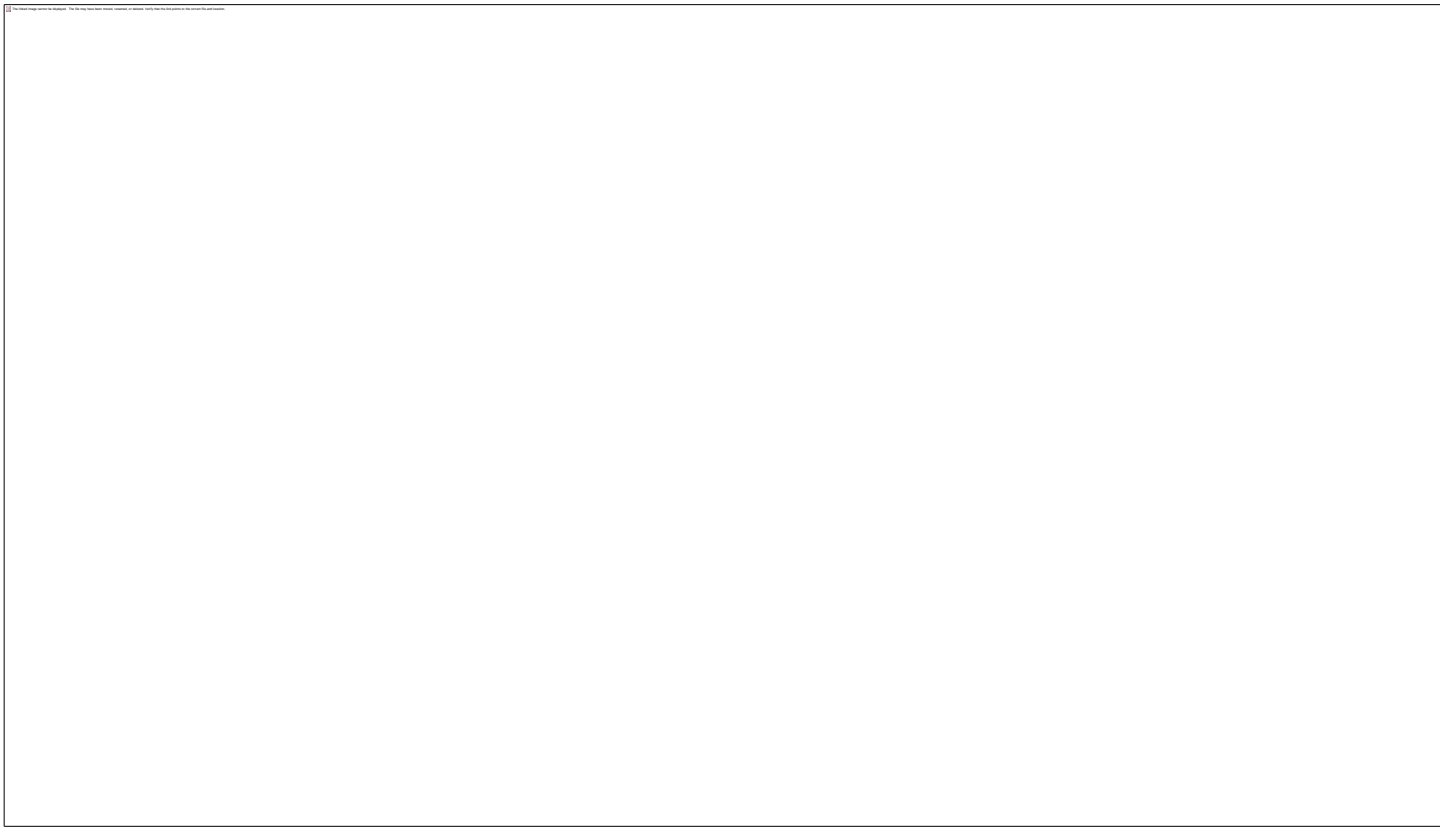
1.3.3 Typedef Documentation

1.3.3.1 typedef double Real

1.3.3.2

1.4 /_cuda/channelflow-0.9.22/channelflow/basisfunc.cpp File Reference

#include <fstream>
#include <iomanip>
#include "channelflow/basisfunc.h"
Include dependency graph for basisfunc.cpp:



1.4.1 Functions

- bool operator!= (const BasisFunc &f, const BasisFunc &g)
- bool operator== (const BasisFunc &f, const BasisFunc &g)
- Real L2Norm (const BasisFunc &phi, bool normalize)
- Real L2Norm2 (const BasisFunc &phi, bool normalize)
- Real L2Dist2 (const BasisFunc &f, const BasisFunc &g, bool normalize)
- Real L2Dist (const BasisFunc &f, const BasisFunc &g, bool normalize)
- Complex L2InnerProduct (const BasisFunc &f, const BasisFunc &g, bool normalize)
- Real chebyNorm (const BasisFunc &phi, bool normalize)
- Real chebyNorm2 (const BasisFunc &phi, bool normalize)
- Real chebyDist2 (const BasisFunc &f, const BasisFunc &g, bool normalize)
- Real chebyDist (const BasisFunc &f, const BasisFunc &g, bool normalize)
- Complex chebyInnerProduct (const BasisFunc &f, const BasisFunc &g, bool normalize)
- Real norm (const BasisFunc &phi, NormType n, bool normalize)

- Real norm2 (const BasisFunc &phi, NormType n, bool normalize)
- Real dist2 (const BasisFunc &f, const BasisFunc &g, NormType n, bool nrmlz)
- Real dist (const BasisFunc &f, const BasisFunc &g, NormType n, bool nrmlz)
- Complex innerProduct (const BasisFunc &f, const BasisFunc &g, NormType n, bool nrmlz)
- Real divNorm2 (const BasisFunc &f, bool normalize)
- Real divNorm (const BasisFunc &f, bool normalize)
- Real divDist2 (const BasisFunc &f, const BasisFunc &g, bool normalize)
- Real divDist (const BasisFunc &f, const BasisFunc &g, bool normalize)
- Real bcNorm2 (const BasisFunc &f, bool normalize)
- Real bcNorm (const BasisFunc &f, bool normalize)
- Real bcDist2 (const BasisFunc &f, const BasisFunc &g, bool normalize)
- Real bcNorm (const BasisFunc &f, const BasisFunc &g, bool normalize)
- BasisFunc xdiff (const BasisFunc &phi)
- BasisFunc ydiff (const BasisFunc &phi)
- BasisFunc zdiff (const BasisFunc &phi)
- ComplexChebyCoeff divergence (const BasisFunc &phi)
- BasisFunc laplacian (const BasisFunc &phi)
- BasisFunc gradient (const ComplexChebyCoeff &f, int kx, int kz, Real Lx, Real Lz)
- BasisFunc gradient (const BasisFunc &phi, int i)
- ComplexChebyCoeff dot (const BasisFunc &f_, const BasisFunc &g_)
- BasisFunc cross (const BasisFunc &f_, const BasisFunc &g_)
- BasisFunc curl (const BasisFunc &f_)
- void dotgrad (const BasisFunc &phi_n, const BasisFunc &grad_phi_pu, const BasisFunc &grad_phi_pv, const BasisFunc &grad_phi_pw, ComplexChebyCoeff &tmp, BasisFunc &rtn)
- BasisFunc dotgrad (const BasisFunc &phi_n_arg, const BasisFunc &phi_p_arg)
- void ubcFix (ChebyCoeff &u, bool a_dirichlet, bool b_dirichlet)
- void ubcFix (ComplexChebyCoeff &u, bool a_dirichlet, bool b_dirichlet)
- void vbcFix (ChebyCoeff &v, bool a_dirichlet, bool b_dirichlet)
- void vbcFix (ComplexChebyCoeff &v, bool a_dirichlet, bool b_dirichlet)

1.4.2 Function Documentation

1.4.2.1 Real bcDist2 (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize*)

References abs2(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

Referenced by bcDist().

```
716                                                 {
717   Real bc2 =
718     abs2(f.u().eval_a() - g.u().eval_a()) +
719     abs2(f.u().eval_b() - g.u().eval_b()) +
720     abs2(f.v().eval_a() - g.v().eval_a()) +
```

```
721     abs2(f.v().eval_b() - g.v().eval_b()) +
722     abs2(f.w().eval_a() - g.w().eval_a()) +
723     abs2(f.w().eval_b() - g.w().eval_b());
724   if (!normalize)
725     bc2 *= f.Lx()*f.Lz();
726   return bc2;
727 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.4.2.2 Real bcNorm (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize*)

References bcNorm2().

```
729                                                 {
730   return sqrt(bcNorm2(f, normalize));
731 }
```

Here is the call graph for this function:

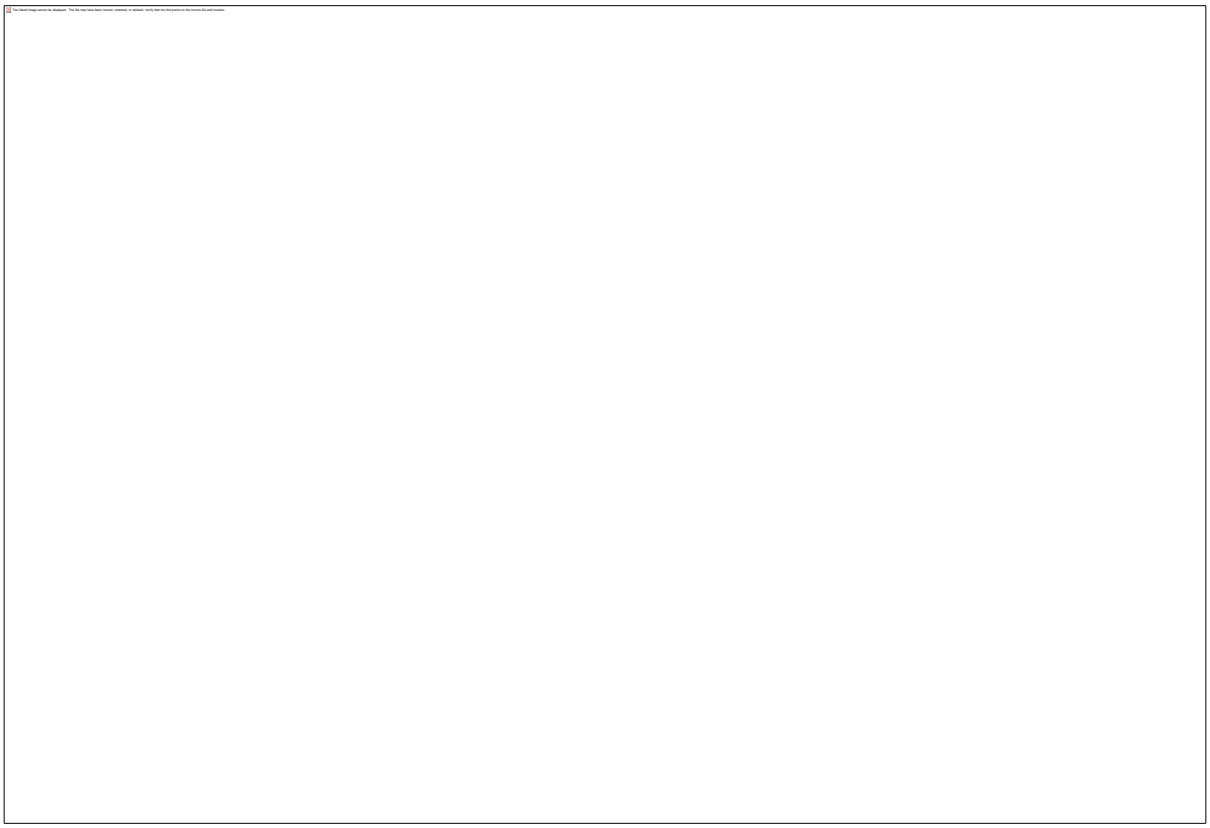
1.4.2.3 Real bcNorm (const BasisFunc & *f*, bool *normalize*)

References bcNorm2().

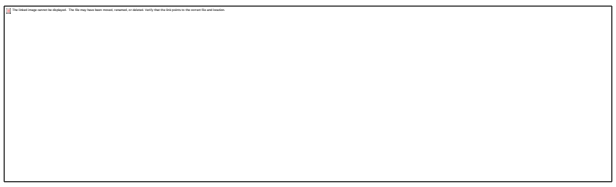
Referenced by PressureSolver::verify(), and PoissonSolver::verify().

```
712                                         {
713   return sqrt(bcNorm2(f, normalize));
714 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.4.2.4 Real bcNorm2 (const BasisFunc & *f*, bool *normalize*)

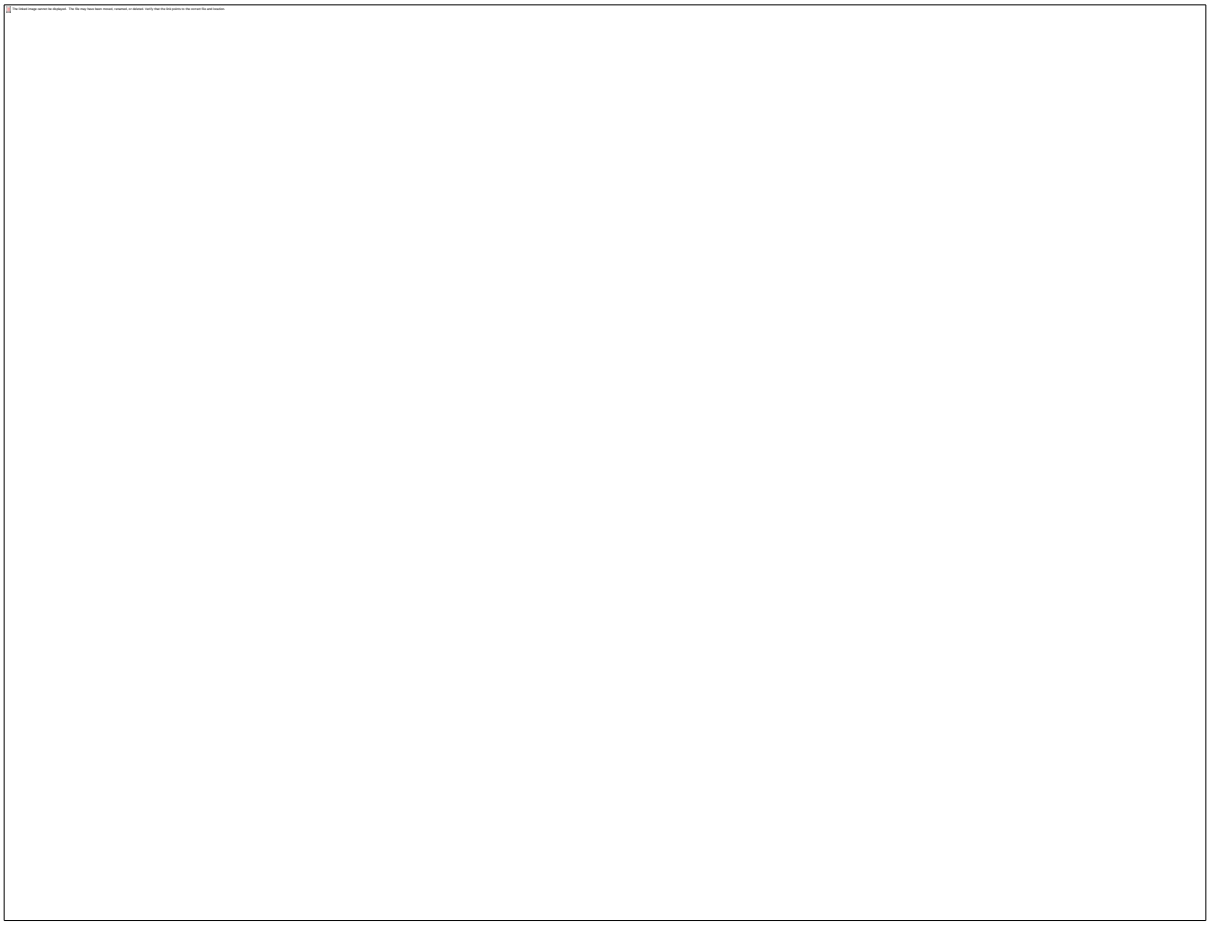
References abs2(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

Referenced by bcNorm().

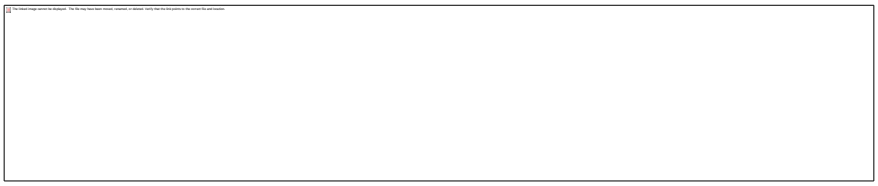
```
702                                     {
703   Real bc2 =
704     abs2(f.u().eval_a()) + abs2(f.u().eval_b()) +
705     abs2(f.v().eval_a()) + abs2(f.v().eval_b()) +
706     abs2(f.w().eval_a()) + abs2(f.w().eval_b());
```

```
707   if (!normalize)
708     bc2 *= f.Lx()*f.Lz();
709   return bc2;
710 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.4.2.5 Real chebyDist (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize*)

References chebyDist2().

Referenced by dist().

```
623                                                 {
624   return sqrt(chebyDist2(f,g,normalize));
625 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

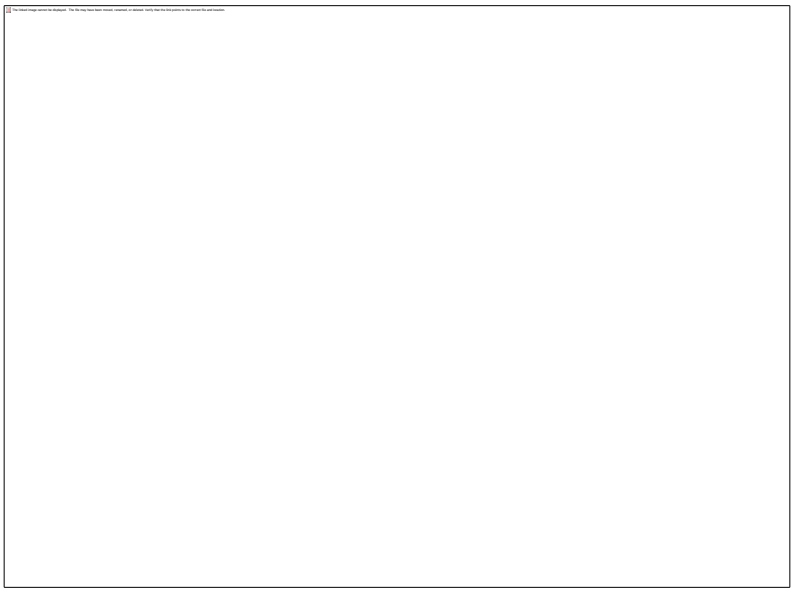
1.4.2.6 Real chebyDist2 (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize*)

References chebyDist2(), BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

Referenced by chebyDist(), chebyDist2(), and dist2().

```
611                                                 {
612   assert(f.geometryCongruent(g));
613   Real rtn=0.0;
614   if (f.kx()==g.kx() && f.kz()==g.kz())
615     for (int n=0; n<f.Nd(); ++n)
616       rtn += chebyDist2(f[n], g[n], normalize);
617   if (!normalize)
618     rtn *= f.Lx()*f.Lz();
619   return rtn;
620
621 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.4.2.7 Complex chebyInnerProduct (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize*)

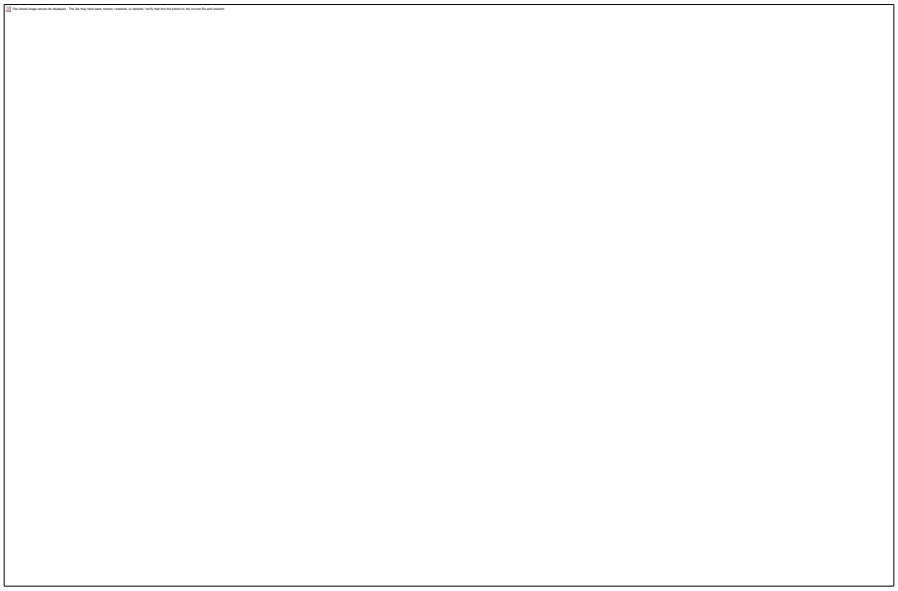
References chebyInnerProduct(), BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

Referenced by chebyInnerProduct(), and innerProduct().

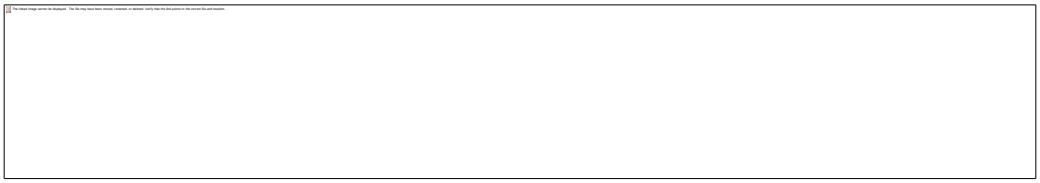
```
628                                     {
629   assert(f.geometryCongruent(g));
630   Complex rtn(0.0, 0.0);
631   if (f.kx()==g.kx() && f.kz()==g.kz())
632     for (int n=0; n<f.Nd(); ++n)
633       rtn += chebyInnerProduct(f[n], g[n], normalize);
634   if (!normalize)
635     rtn *= f.Lx() * f.Lz();
636   return rtn;
```

```
637 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



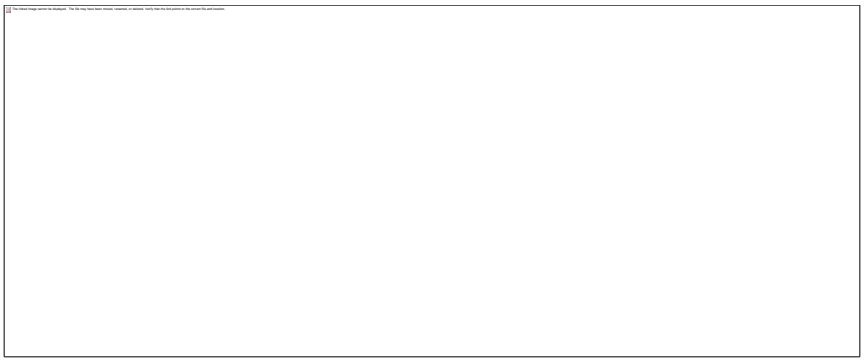
1.4.2.8 Real chebyNorm (const BasisFunc & *phi*, bool *normalize*)

References chebyNorm2(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

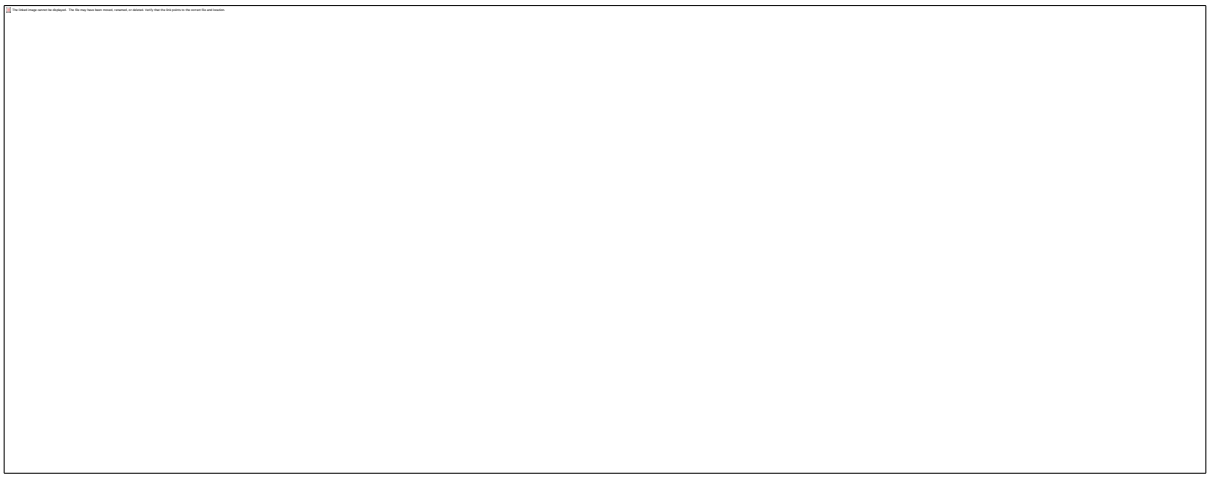
Referenced by norm().

```
593                                         {
594   Real rtn2 = 0.0;
595   for (int n=0; n<phi.Nd(); ++n)
596     rtn2 += chebyNorm2(phi[n], normalize);
597   if (!normalize)
598     rtn2 *= phi.Lx()*phi.Lz();
599   return sqrt(rtn2);
600 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

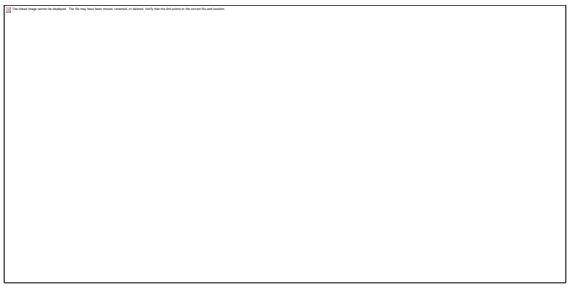


1.4.2.9 Real chebyNorm2 (const BasisFunc & *phi*, bool *normalize*)

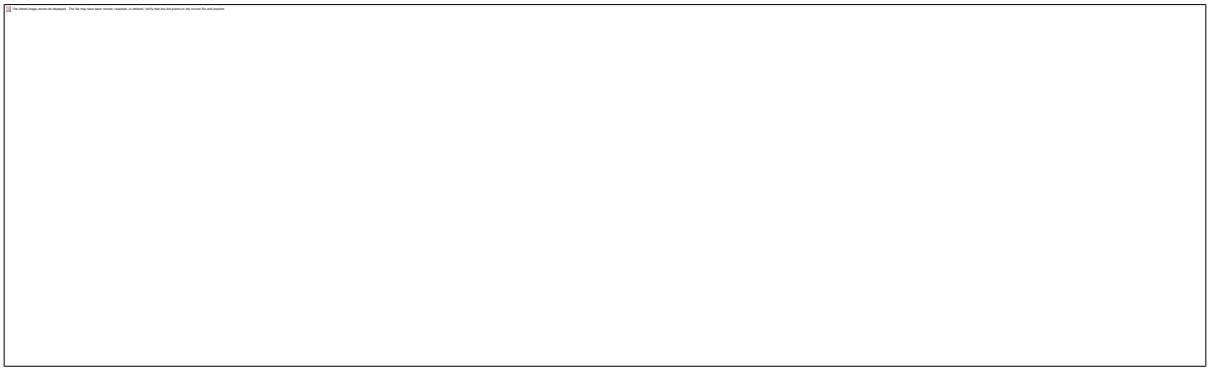
References chebyNorm2(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().
Referenced by chebyNorm(), chebyNorm2(), and norm2().

```
602                                         {
603   Real rtn = 0.0;
604   for (int n=0; n<phi.Nd(); ++n)
605     rtn += chebyNorm2(phi[n], normalize);
606   if (!normalize)
607     rtn *= phi.Lx()*phi.Lz();
608   return rtn;
609 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.4.2.10 BasisFunc cross (const BasisFunc & *f_*, const BasisFunc & *g_*)

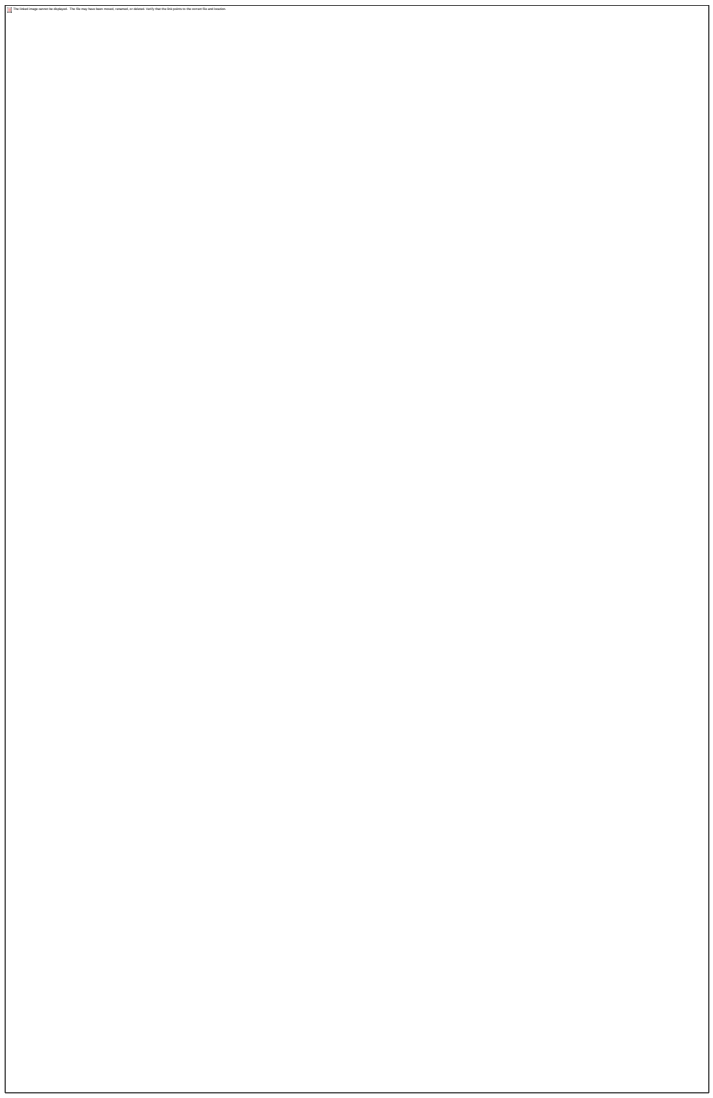
References BasisFunc::a(), BasisFunc::b(), BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::makePhysical(), BasisFunc::makeSpectral(), BasisFunc::Ny(), Physical, Spectral, and BasisFunc::state().

Referenced by cross().

```
842                                                {
843   BasisFunc& f = (BasisFunc&) f_;
844   BasisFunc& g = (BasisFunc&) g_;
845   assert(f.geometryCongruent(g));
846   int Ny= f.Ny();
847
848   ChebyTransform t(Ny);
849   fieldstate entrystate = f.state();
850   f.makePhysical(t);
851   g.makePhysical(t);
852
853
854   BasisFunc rtn(Ny, f.kx()+g.kx(), f.kz()+g.kz(), f.Lx(), f.Lz(), f.a(), f.b(),
855                 Physical);
856   for (int ny=0; ny<Ny; ++ny) {
857     Complex fu = f[0][ny];
```

```
858     Complex fv = f[1][ny];
859     Complex fw = f[2][ny];
860     Complex gu = g[0][ny];
861     Complex gv = g[1][ny];
862     Complex gw = g[2][ny];
863 
864     rtn[0].set(ny, fv*gw - fw*gv);
865     rtn[1].set(ny, fw*gu - fu*gw);
866     rtn[2].set(ny, fu*gv - fv*gu);
867   }
868 
869   if (entrystate==Spectral) {
870     f.makeSpectral(t);
871     g.makeSpectral(t);
872     rtn.makeSpectral(t);
873   }
874   return rtn;
875 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.4.2.11 BasisFunc curl (const BasisFunc & *f_*)

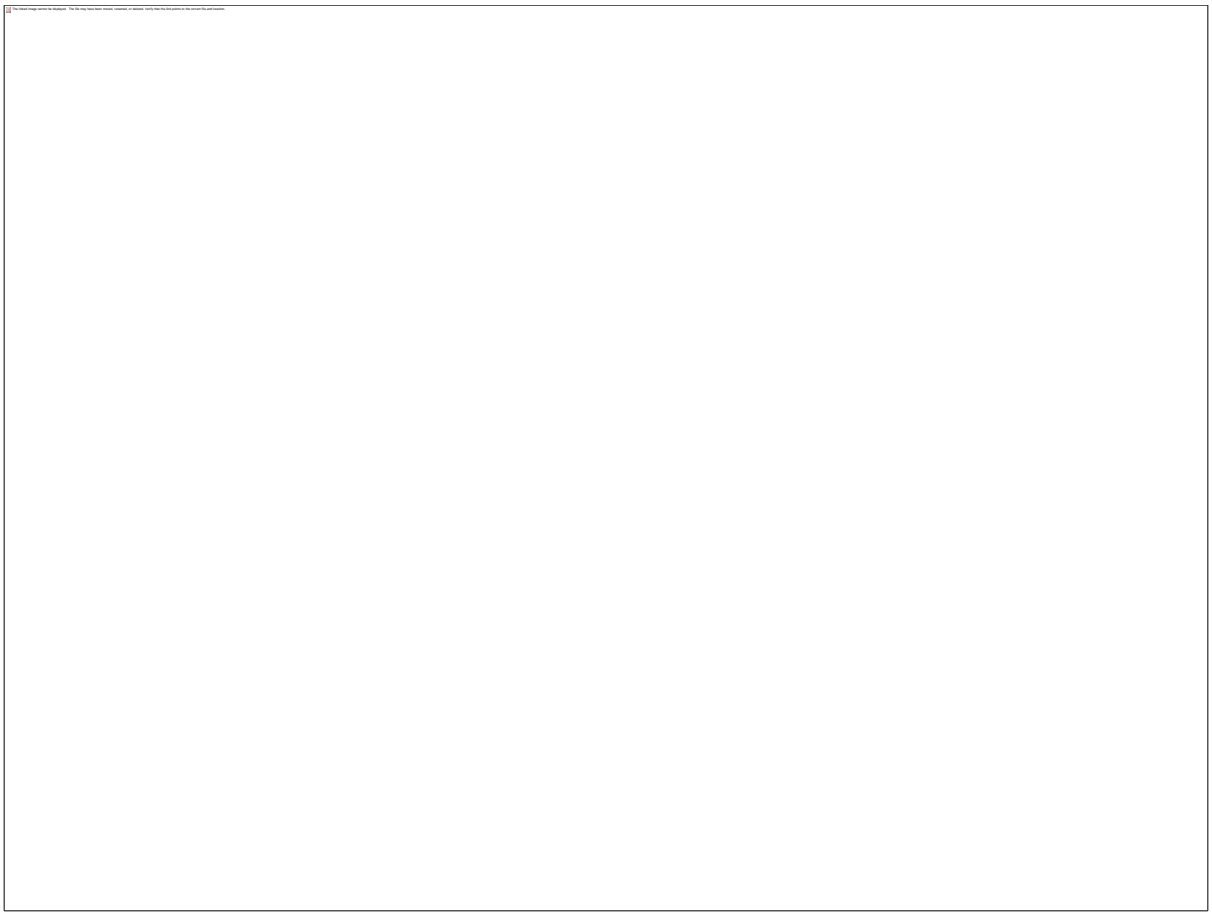
References I(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::makeSpectral(), BasisFunc::Ny(), pi, BasisFunc::setToZero(), Spectral, BasisFunc::state(), and ydiff().

Referenced by curl(), and rotationalNL().

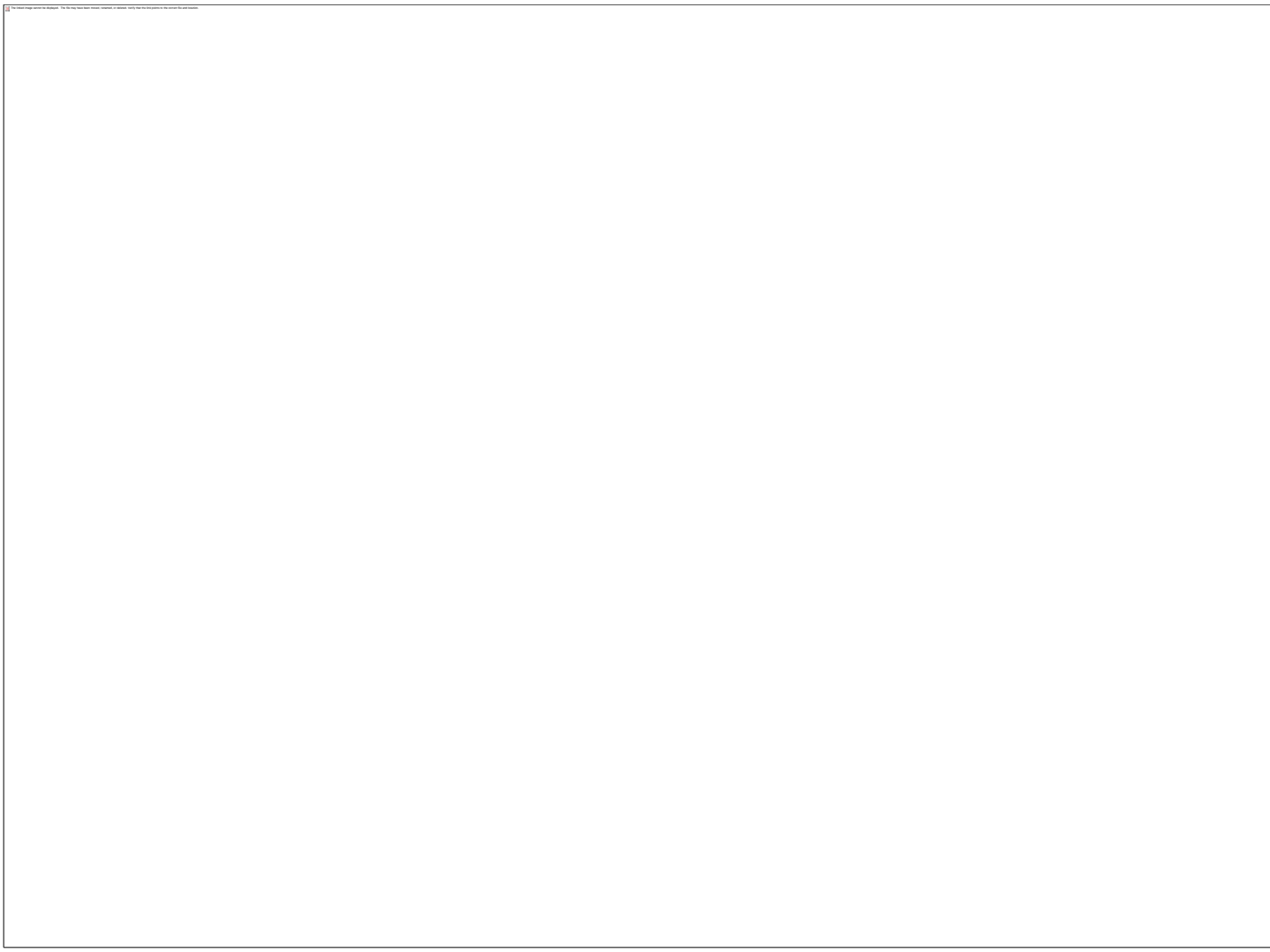
```
877                             {
878
879   BasisFunc& f = (BasisFunc&) f_;
```

```
880   fieldstate entrystate = f.state();
881   int Ny= f.Ny();
882   ChebyTransform t(Ny);
883   f.makeSpectral(t);
884
885   BasisFunc f_y = ydiff(f);
886   Complex d_dx = (2*pi*f.kx()/f.Lx())*I;
887   Complex d_dz = (2*pi*f.kz()/f.Lz())*I;
888
889   BasisFunc rtn(f);
890
891   rtn.setToZero();
892   for (int ny=0; ny<Ny; ++ny) {
893     Complex u  = f[0][ny];
894     Complex v  = f[1][ny];
895     Complex w  = f[2][ny];
896     Complex u_y = f_y[0][ny];
897     Complex w_y = f_y[2][ny];
898
899     rtn[0].set(ny, w_y - d_dz*v);
900     rtn[1].set(ny, d_dz*u - d_dx*w);
901     rtn[2].set(ny, d_dx*v - u_y);
902   }
903
904   if (entrystate==Spectral) {
905     f.makeSpectral(t);
906     rtn.makeSpectral(t);
907   }
908   return rtn;
909 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.4.2.12 Real dist (const BasisFunc & *f*, const BasisFunc & *g*, NormType *n*, bool *nrmlz*)

References chebyDist(), L2Dist(), and L2NORM.

```
653                                                 {
654   return (n==L2NORM) ? L2Dist(f,g,nrmlz) : chebyDist(f,g,nrmlz);
655 }
```

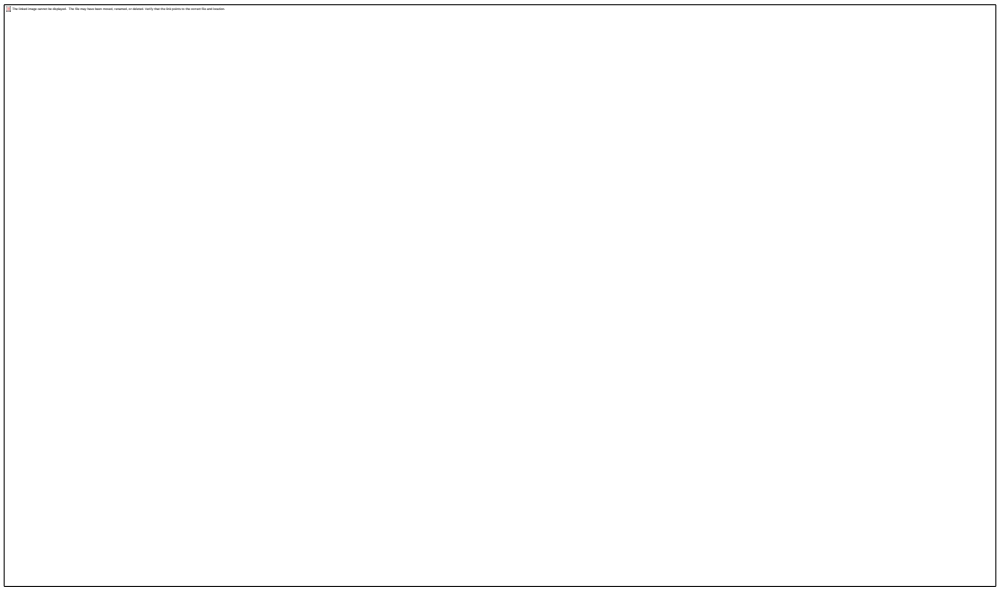
Here is the call graph for this function:

1.4.2.13 Real dist2 (const BasisFunc & *f*, const BasisFunc & *g*, NormType *n*, bool *nrmlz*)

References chebyDist2(), L2Dist2(), and L2NORM.

```
649                                                     {
650   return (n==L2NORM) ? L2Dist2(f,g,nrmlz) : chebyDist2(f,g,nrmlz);
651 }
```

Here is the call graph for this function:

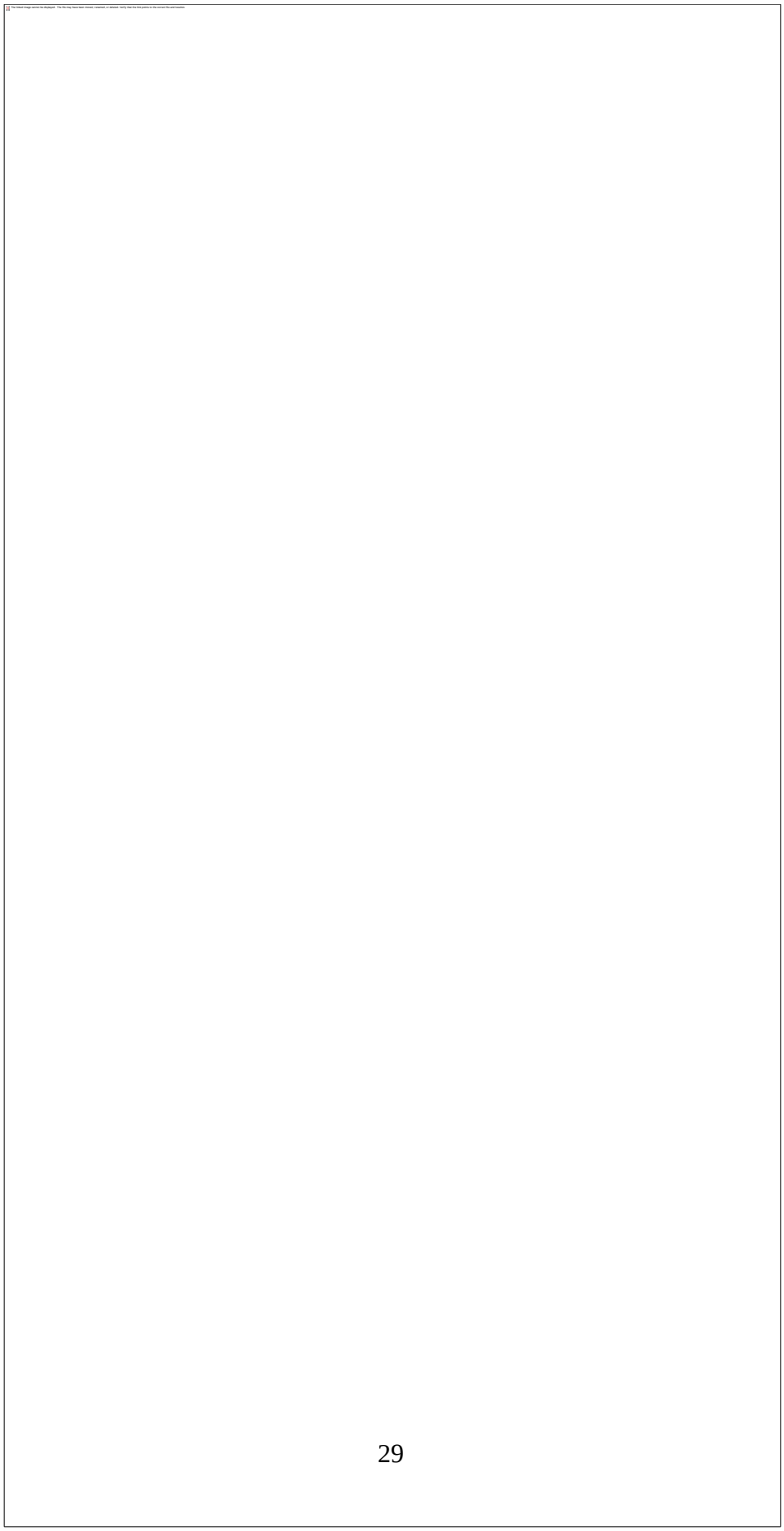


1.4.2.14 Real divDist (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize*)

References divDist2().

```
698                                                 {
699   return sqrt(divDist2(f,g, normalize));
700 }
```

Here is the call graph for this function:



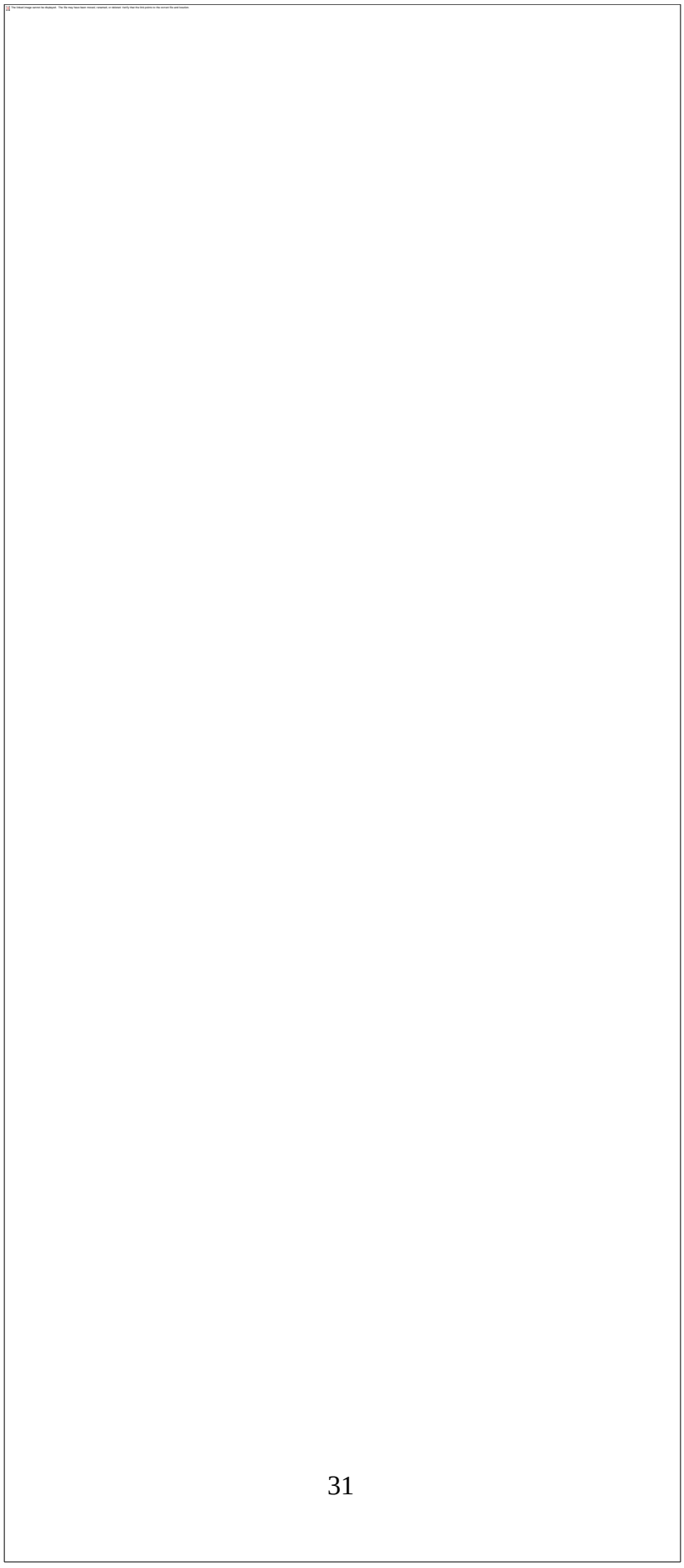
1.4.2.15 Real divDist2 (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize*)

References BasisFunc::congruent(), diff(), div(), I(), BasisFunc::kx(), BasisFunc::kz(), L2Norm2(), BasisFunc::Lx(), BasisFunc::Lz(), pi, Spectral, BasisFunc::state(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

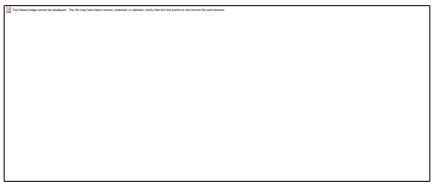
Referenced by divDist(), and divDist2().

```
681                                                 {
682   assert(f.state() == Spectral && g.state() == Spectral);
683   assert(f.congruent(g));
684   ComplexChebyCoeff div = f.u();
685   div -= g.u();
686   div *= (2*pi*f.kx()/f.Lx())*I;
687   ComplexChebyCoeff tmp = f.w();
688   tmp -= g.w();
689   tmp *= (2*pi*f.kz()/f.Lz())*I;
690   div += tmp;
691   diff(f.v(), tmp);
692   div += tmp;
693   diff(g.v(), tmp);
694   div -= tmp;
695   return L2Norm2(div, normalize);
696 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

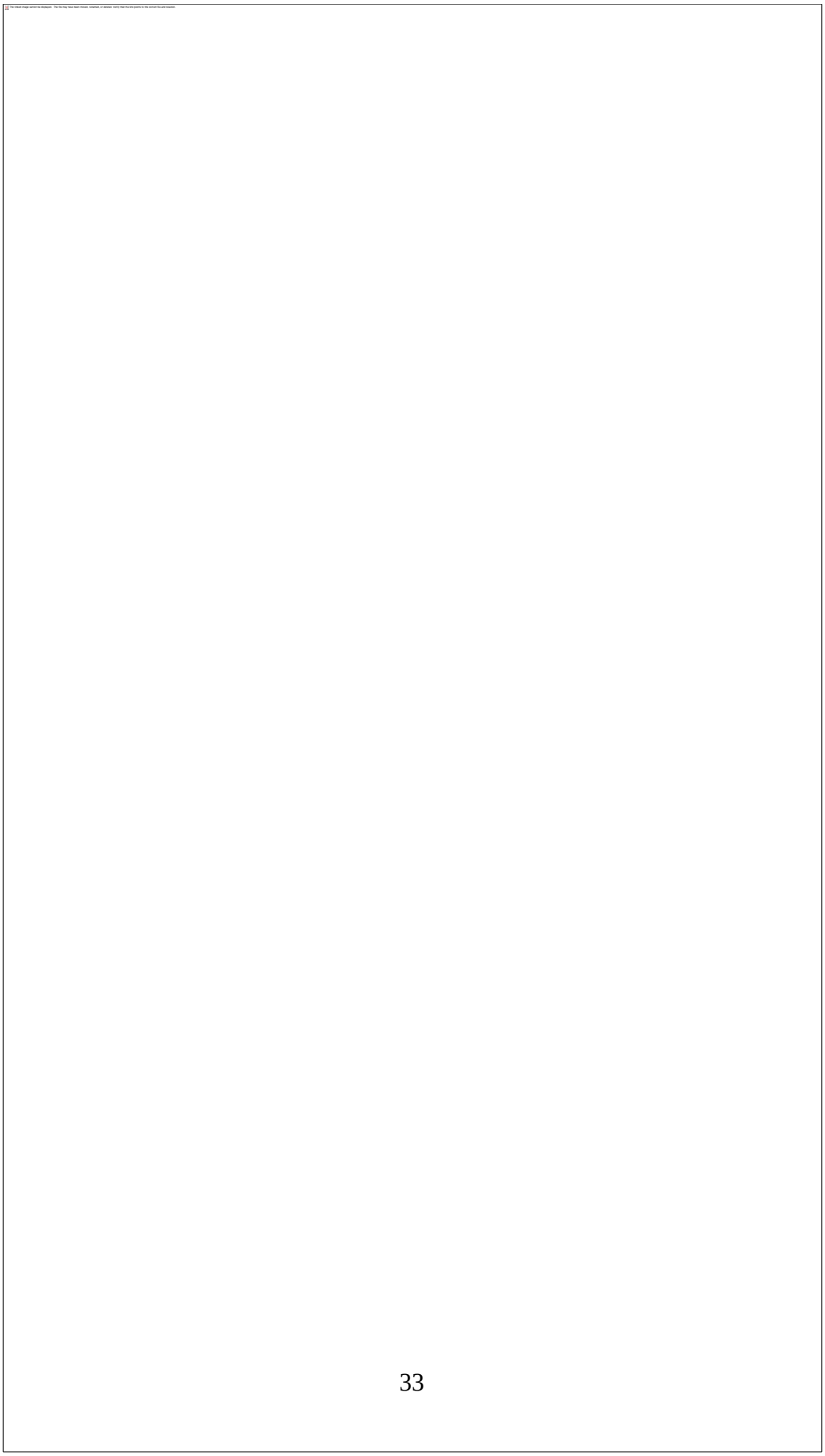


1.4.2.16 ComplexChebyCoeff divergence (const BasisFunc & *phi*)

References BasisFunc::a(), BasisFunc::b(), diff(), div(), I(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::Ny(), pi, Spectral, and BasisFunc::state().

```
755                                         {
756   assert(phi.state() == Spectral);
757   ComplexChebyCoeff div(phi.Ny(), phi.a(), phi.b(), Spectral);
758   ComplexChebyCoeff tmp(phi.Ny(), phi.a(), phi.b(), Spectral);
759   tmp = phi[0];
760   tmp *= (2*pi*phi.kx()/phi.Lx())*I;
761   div += tmp;
762 
763   diff(phi[1], tmp);
764   div += tmp;
765 
766   tmp = phi[2];
767   tmp *= (2*pi*phi.kz()/phi.Lz())*I;
768   div += tmp;
769 
770   return div;
771 }
```

Here is the call graph for this function:



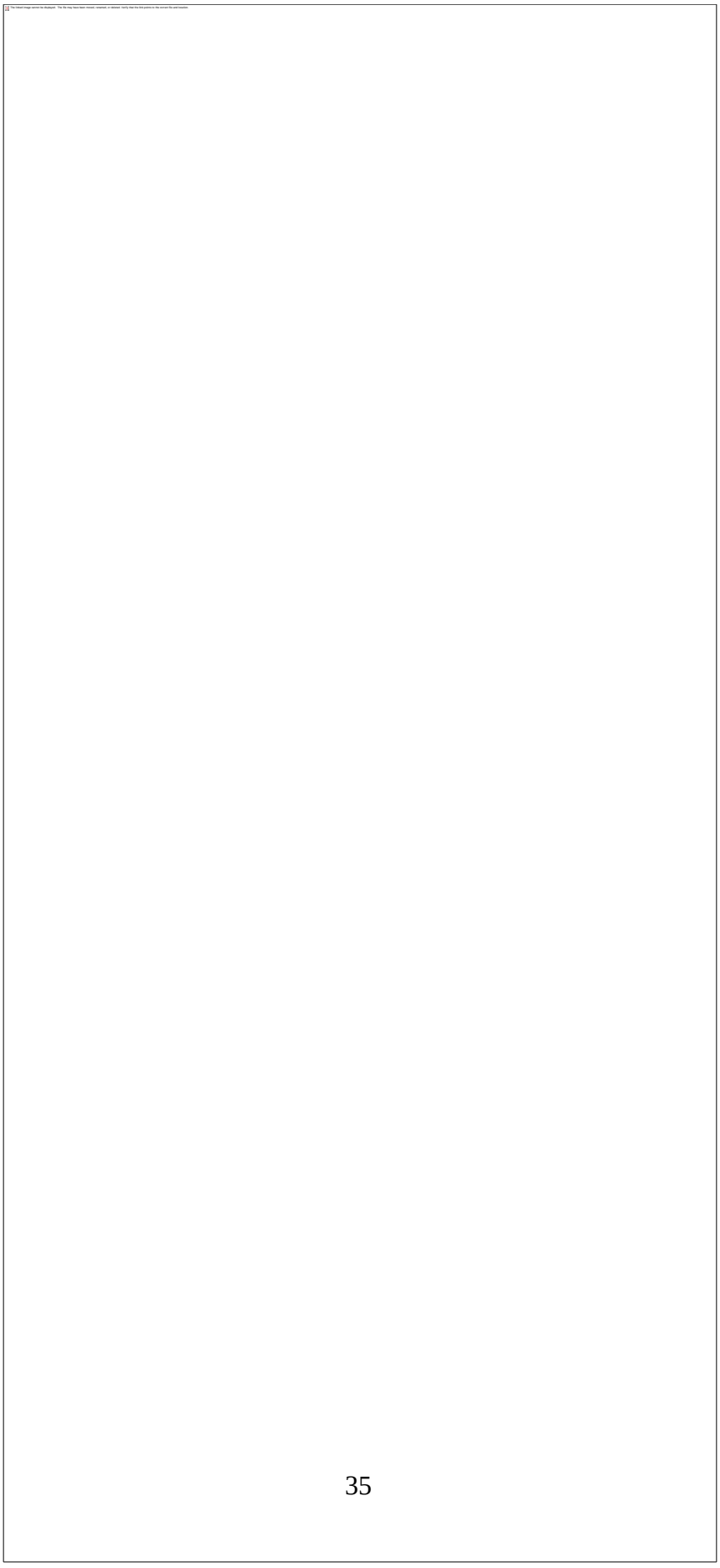
1.4.2.17 Real divNorm (const BasisFunc & *f*, bool *normalize*)

References divNorm2().

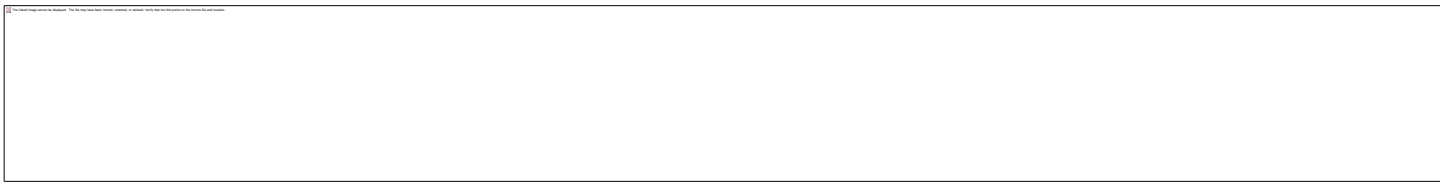
Referenced by chebyProfile(), and randomProfile().

```
677                                     {
678   return sqrt(divNorm2(f, normalize));
679 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



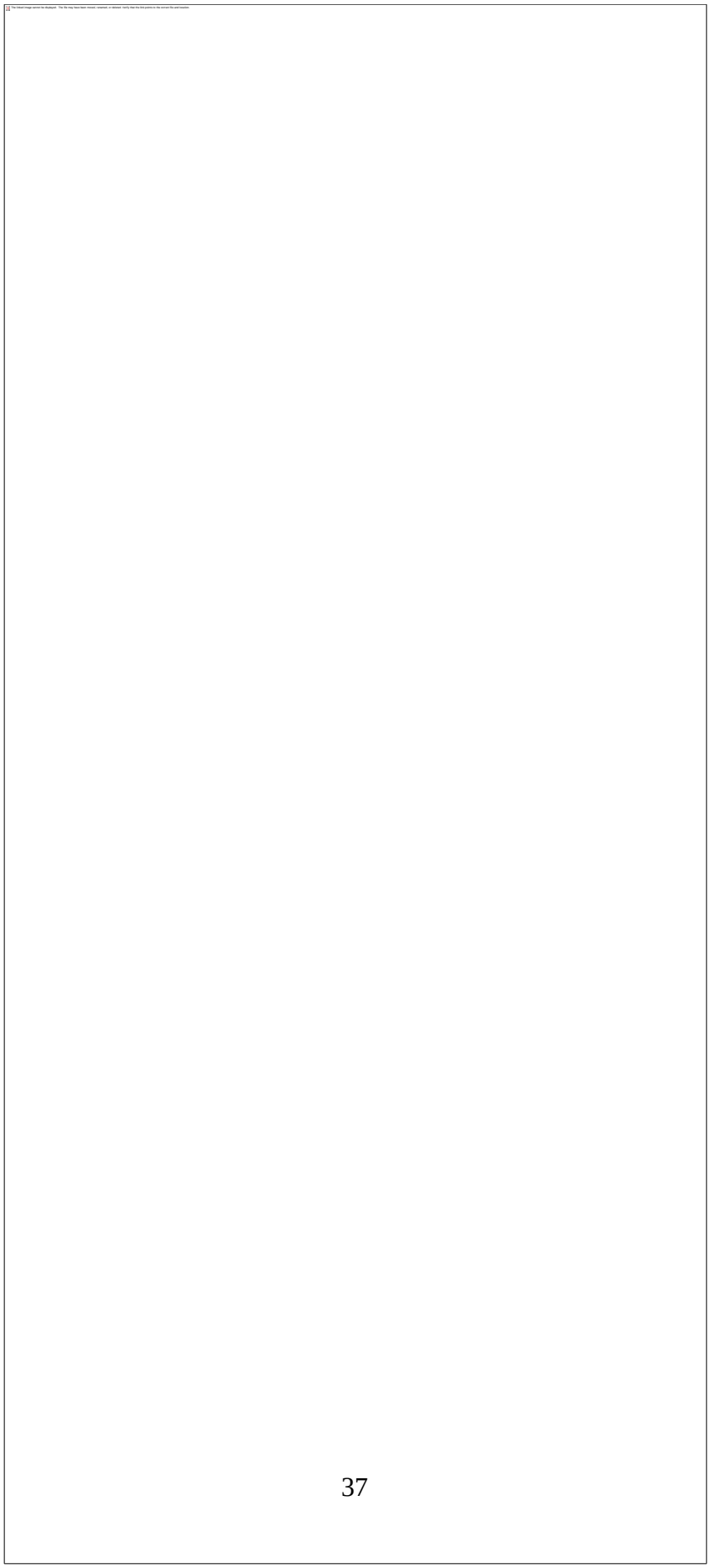
1.4.2.18 Real divNorm2 (const BasisFunc & *f*, bool *normalize*)

References diff(), div(), I(), BasisFunc::kx(), BasisFunc::kz(), L2Norm2(), BasisFunc::Lx(), BasisFunc::Lz(), pi, Spectral, BasisFunc::state(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

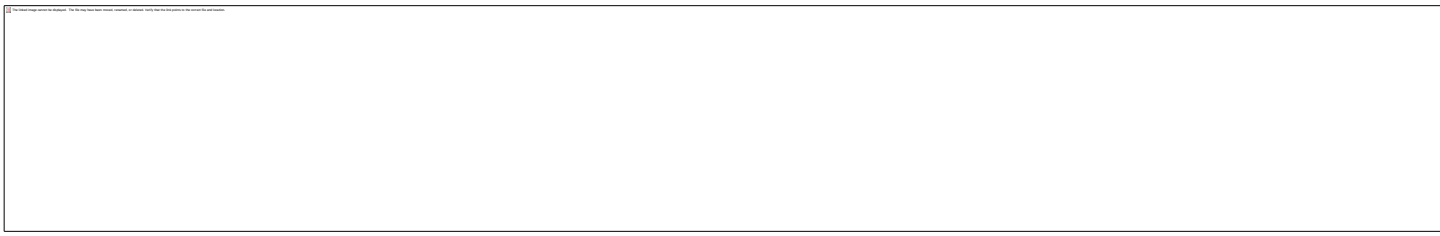
Referenced by divNorm(), and divNorm2().

```
665                                         {
666   assert(f.state() == Spectral);
667   ComplexChebyCoeff div = f.u();
668   div *= (2*pi*f.kx()/f.Lx())*I;
669   ComplexChebyCoeff tmp = f.w();
670   tmp *= (2*pi*f.kz()/f.Lz())*I;
671   div += tmp;
672   diff(f.v(), tmp);
673   div += tmp;
674   return L2Norm2(div, normalize);
675 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



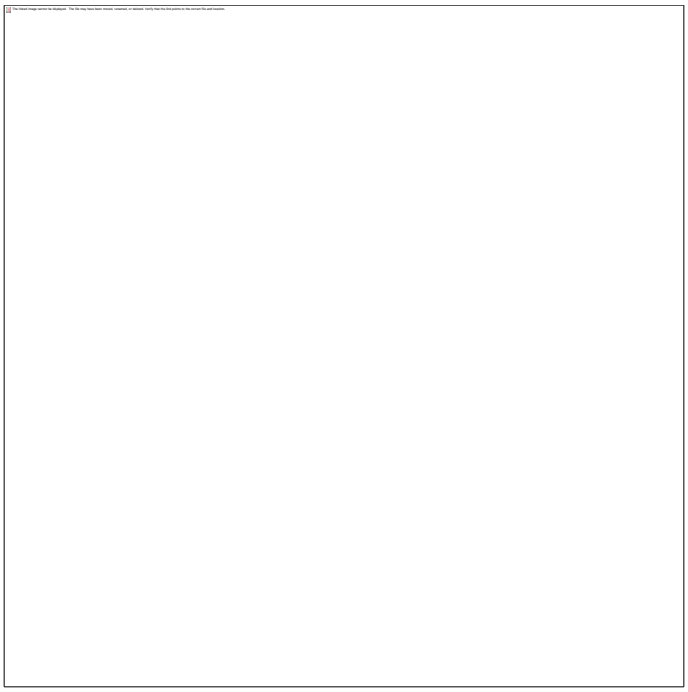
1.4.2.19 ComplexChebyCoeff dot (const BasisFunc & *f_*, const BasisFunc & *g_*)

References BasisFunc::a(), BasisFunc::b(), BasisFunc::geometryCongruent(), BasisFunc::makePhysical(), BasisFunc::makeState(), BasisFunc::Ny(), Physical, and BasisFunc::state().

Referenced by dot().

```
819                                                 {
820   BasisFunc& f = (BasisFunc&) f_;
821   BasisFunc& g = (BasisFunc&) g_;
822   assert(f.geometryCongruent(g));
823   int Ny= f.Ny();
824   fieldstate fstate = f.state();
825   fieldstate gstate = g.state();
826   ChebyTransform t(Ny);
827   f.makePhysical(t);
828   g.makePhysical(t);
829
830   ComplexChebyCoeff fg(Ny, f.a(), f.b(), Physical);
831   for (int ny=0; ny<Ny; ++ny)
832     fg.set(ny, f[0][ny]*g[0][ny] + f[1][ny]*g[1][ny] + f[2][ny]*g[2][ny]);
833
834
835   f.makeState(fstate,t);
836   g.makeState(gstate,t);
837   fg.makeState(fstate, t);
838   return fg;
839 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.4.2.20 BasisFunc dotgrad (const BasisFunc & *phi_n_arg*, const BasisFunc & *phi_p_arg*)

References BasisFunc::a(), BasisFunc::b(), dotgrad(), gradient(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::makePhysical(), BasisFunc::makeSpectral(), BasisFunc::Ny(), Physical, Spectral, and BasisFunc::state().

```
963                                                     {
964   BasisFunc& phi_n = (BasisFunc&) phi_n_arg;
965   BasisFunc& phi_p = (BasisFunc&) phi_p_arg;
966
967   fieldstate original_n_state = phi_n.state();
968   fieldstate original_p_state = phi_p.state();
969
970   int Ny=phi_n.Ny();
971   ChebyTransform trans(Ny);
972   phi_p.makeSpectral(trans);   // Now phi_p is spectral
973
974   BasisFunc rtn(Ny, phi_n.kx()+phi_p.kx(), phi_n.kz()+phi_p.kz(),
975                 phi_n.Lx(), phi_n.Lz(), phi_n.a(), phi_n.b(), Spectral);
976
977   BasisFunc grad_phi_pu = gradient(phi_p, 0);
```

```
978   BasisFunc grad_phi_pv = gradient(phi_p, 1);
979   BasisFunc grad_phi_pw = gradient(phi_p, 2);
980   ComplexChebyCoeff tmp(Ny, phi_n.a(), phi_n.b(), Physical);
981
982   phi_n.makePhysical(trans);  // Now phi_n is physical
983   grad_phi_pu.makePhysical(trans);
984   grad_phi_pv.makePhysical(trans);
985   grad_phi_pw.makePhysical(trans);
986
987   dotgrad(phi_n, grad_phi_pu, grad_phi_pv, grad_phi_pw, tmp, rtn);
988
989   // phi_n is physical,
990   if (original_n_state == Spectral) {
991     phi_n.makeSpectral(trans);
992     rtn.makeSpectral(trans);
993   }
994   // phi_p is spectral.
995   if (original_p_state == Physical)
996     phi_p.makePhysical(trans);
997
998   return rtn;
999 }
```

Here is the call graph for this function:

1.4.2.21 void dotgrad (const BasisFunc & *phi_n*, const BasisFunc & *grad_phi_pu*, const BasisFunc & *grad_phi_pv*, const BasisFunc & *grad_phi_pw*, ComplexChebyCoeff & *tmp*, BasisFunc & *rtn*)

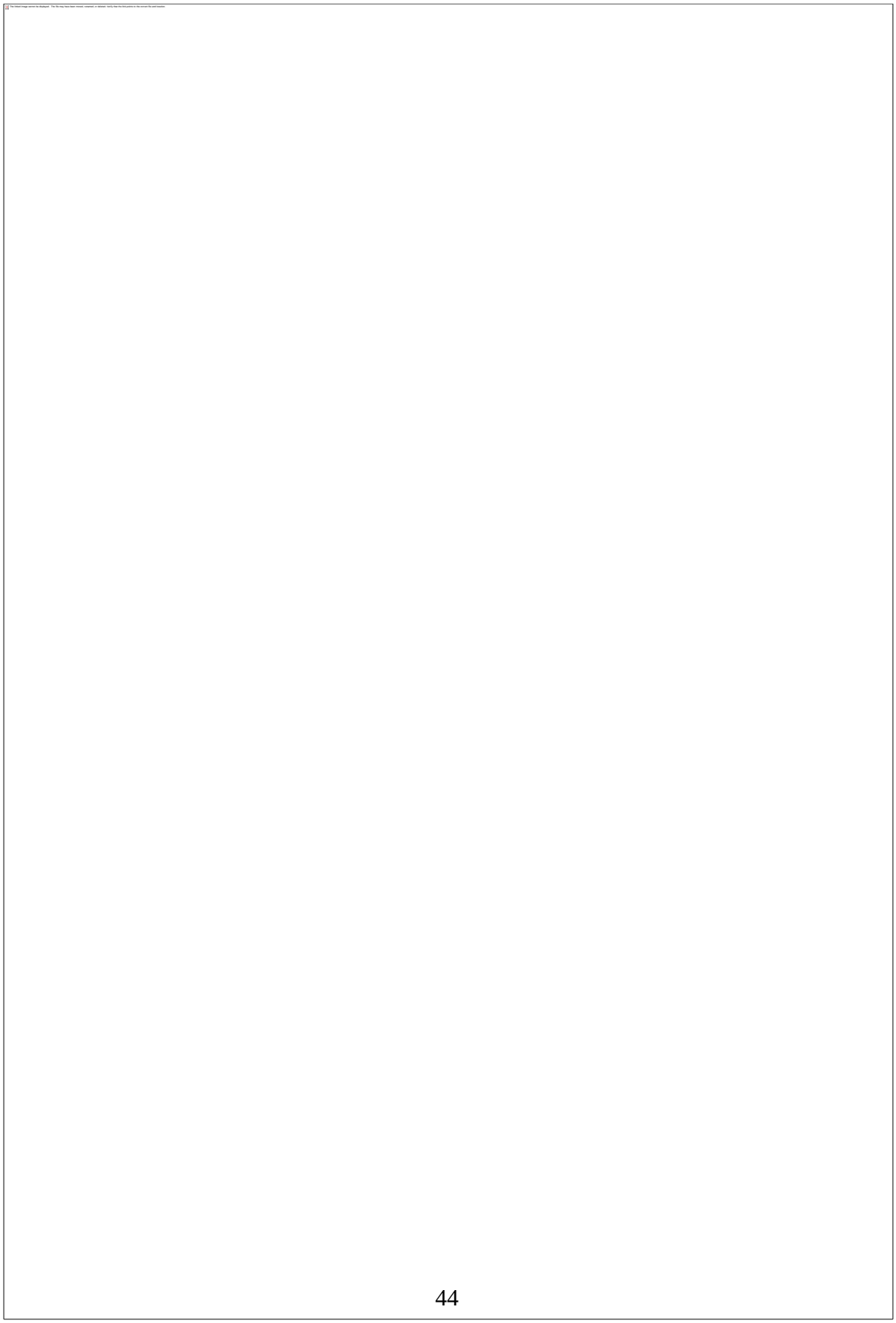
References BasisFunc::a(), BasisFunc::b(), BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), Physical, ComplexChebyCoeff::setBounds(), BasisFunc::setBounds(), BasisFunc::setkxkz(), ComplexChebyCoeff::setState(), BasisFunc::setState(), ComplexChebyCoeff::setToZero(), BasisFunc::setToZero(), BasisFunc::state(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

Referenced by dotgrad().

```
915 {
916   assert(phi_n.state() == Physical);
917   assert(grad_phi_pu.state() == Physical);
918   assert(grad_phi_pv.state() == Physical);
919   assert(grad_phi_pw.state() == Physical);
920   assert(phi_n.geometryCongruent(grad_phi_pu));
921   assert(phi_n.geometryCongruent(grad_phi_pv));
922   assert(phi_n.geometryCongruent(grad_phi_pw));
923   rtn.setToZero();
924   rtn.setkxkz(phi_n.kx()+grad_phi_pu.kx(), phi_n.kz()+grad_phi_pu.kz());
925   rtn.setBounds(phi_n.Lx(), phi_n.Lz(), phi_n.a(), phi_n.b());
926   rtn.setState(Physical);
927
928   tmp.setToZero();
929   tmp.setBounds(phi_n.a(), phi_n.b());
930   tmp.setState(Physical);
931
932   // ====================================
933   // Calculate rtn.u =   phi_n.u d/dx phi_p.u
934   //                   + phi_n.v d/dy phi_p.u
935   //                   + phi_n.w d/dz phi_p.u
936   int i; // MSVC++ FOR-SCOPE BUG
937   for (i=0; i<3; ++i) {
938     tmp = phi_n[i];
939     tmp *= grad_phi_pu[i];
940     rtn.u() += tmp;
941   }
942   // ====================================
943   // Calculate rtn.v =   phi_n.u d/dx phi_p.v
944   //                   + phi_n.v d/dy phi_p.v
945   //                   + phi_n.w d/dz phi_p.v
946   for (i=0; i<3; ++i) {
947     tmp = phi_n[i];
948     tmp *= grad_phi_pv[i];
949     rtn.v() += tmp;
950   }
951   // ====================================
952   // Calculate rtn.w =   phi_n.u d/dx phi_p.w
953   //                   + phi_n.v d/dy phi_p.w
```

```
954   //                  + phi_n.w d/dz phi_p.w
955   for (i=0; i<3; ++i) {
956     tmp = phi_n[i];
957     tmp *= grad_phi_pw[i];
958     rtn.w() += tmp;
959   }
960   return;
961 }
```

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.

Here is the caller graph for this function:

1.4.2.22 BasisFunc gradient (const BasisFunc & *phi*, int *i*)

References BasisFunc::a(), BasisFunc::b(), diff(), grad(), I(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::Ny(), pi, Spectral, and BasisFunc::state().

```
802                                         {
803   assert(phi.state() == Spectral);
804
805   BasisFunc grad(phi.Ny(), phi.kx(), phi.kz(), phi.Lx(), phi.Lz(), phi.a(), phi.b(),
phi.state());
806   const ComplexChebyCoeff& phi_i = phi[i];
807
808   grad.u() = phi_i;
809   grad.u() *= (2*pi*phi.kx()/phi.Lx())*I;
810
811   diff(phi_i, grad.v());
812
813   grad.w() = phi_i;
814   grad.w() *= (2*pi*phi.kz()/phi.Lz())*I;
815   return grad;
816 }
```

Here is the call graph for this function:

1.4.2.23 BasisFunc gradient (const ComplexChebyCoeff & *f*, int *kx*, int *kz*, Real *Lx*, Real *Lz*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff(), grad(), I(), ComplexChebyCoeff::numModes(), pi, Spectral, and ComplexChebyCoeff::state().

Referenced by dotgrad().

```
791                                                                 {
792   assert(f.state() == Spectral);
793   BasisFunc grad(f.numModes(), kx, kz, Lx, Lz, f.a(), f.b(), f.state());
794   grad.u() = f;
795   grad.u() *= (2*pi*kx/Lx)*I;
796   diff(f, grad.v());
797   grad.w() = f;
798   grad.w() *= (2*pi*kz/Lz)*I;
799   return grad;
800 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.4.2.24 Complex innerProduct (const BasisFunc & *f*, const BasisFunc & *g*, NormType *n*, bool *nrmlz*)

References chebyInnerProduct(), L2InnerProduct(), and L2NORM.

```
658                                {
659   return (n==L2NORM) ? L2InnerProduct(f,g,nrmlz) : chebyInnerProduct(f,g,nrmlz);
660 }
```

Here is the call graph for this function:

1.4.2.25 Real L2Dist (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize*)

References L2Dist2().

Referenced by dist(), TauSolver::tauDist(), TauSolver::verify(), PressureSolver::verify(), PoissonSolver::verify(), and TauSolver::verify_P_and_v().

```
575                                                {
576     return sqrt(L2Dist2(f,g,normalize));
577 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.4.2.26 Real L2Dist2 (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize*)

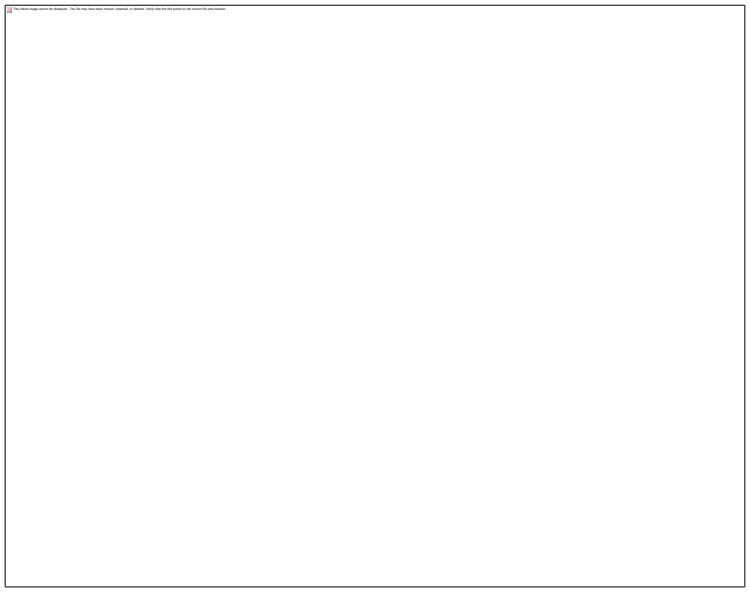
References BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), L2Dist2(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

Referenced by dist2(), L2Dist(), and L2Dist2().

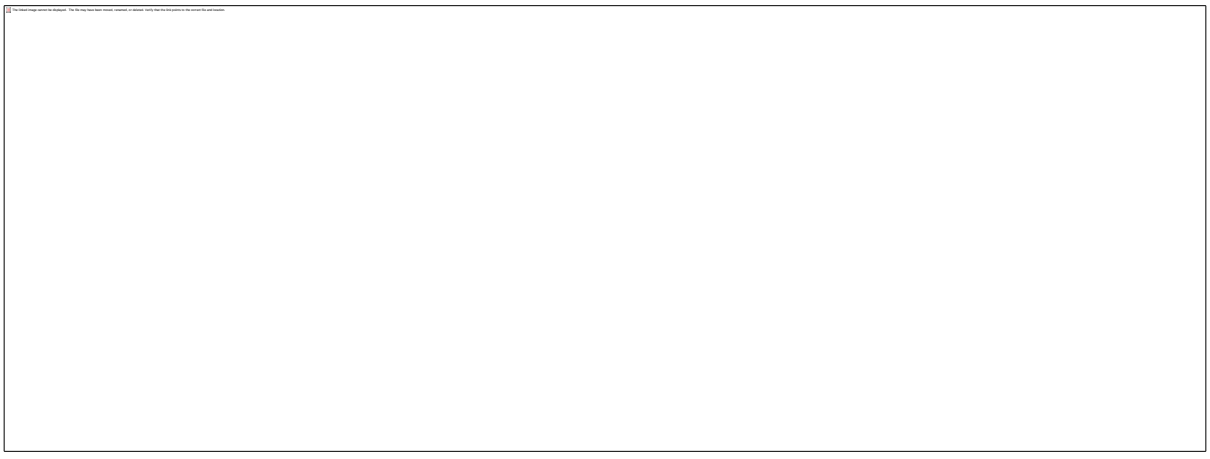
```
563                                                 {
564   assert(f.geometryCongruent(g));
565   Real rtn=0.0;
566   if (f.kx()==g.kx() && f.kz()==g.kz())
```

```
567     for (int n=0; n<f.Nd(); ++n)
568       rtn += L2Dist2(f[n], g[n], normalize);
569   if (!normalize)
570     rtn *= f.Lx()*f.Lz();
571   return rtn;
572 
573 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



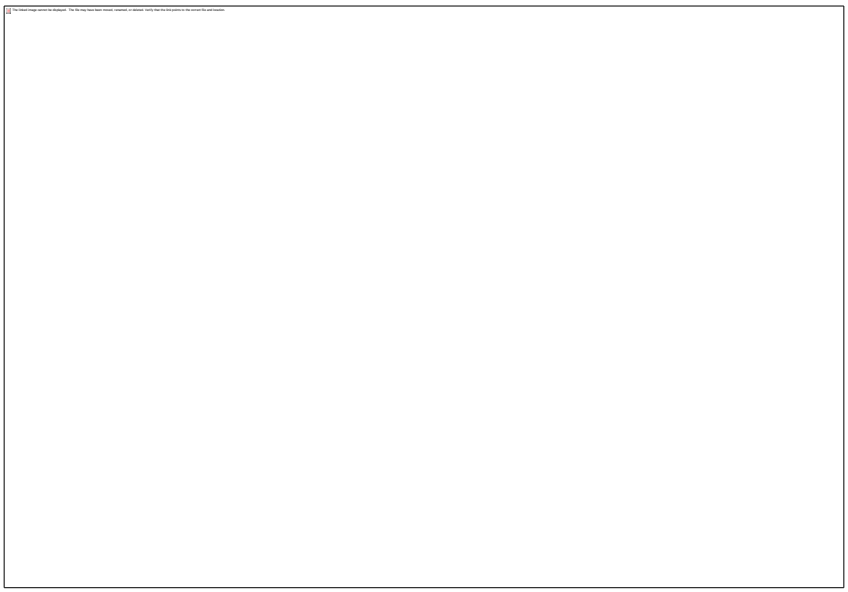
1.4.2.27 Complex L2InnerProduct (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize*)

References BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), L2InnerProduct(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

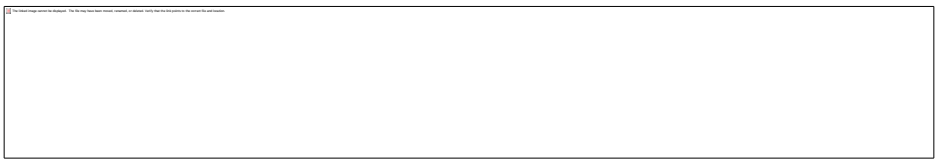
Referenced by innerProduct(), and L2InnerProduct().

```
580                                 {
581   assert(f.geometryCongruent(g));
582   Complex rtn(0.0, 0.0);
583   if (f.kx()==g.kx() && f.kz()==g.kz())
584     for (int n=0; n<f.Nd(); ++n)
585       rtn += L2InnerProduct(f[n], g[n], normalize);
586   if (!normalize)
587     rtn *= f.Lx() * f.Lz();
588   return rtn;
589 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



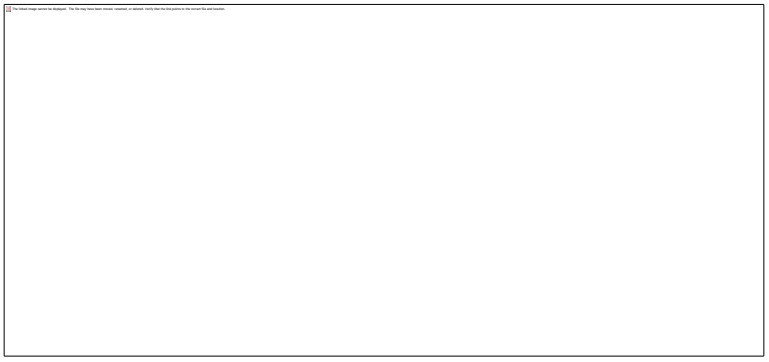
1.4.2.28 Real L2Norm (const BasisFunc & *phi*, bool *normalize*)

References L2Norm2(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

Referenced by chebyProfile(), makemeen(), makeprtb(), makeroll(), makewave(), makewobl(), norm(), randomProfile(), TauSolver::tauNorm(), TauSolver::verify(), PressureSolver::verify(), PoissonSolver::verify(), and TauSolver::verify_P_and_v().

```
545                                     {
546   Real rtn2 = 0.0;
547   for (int n=0; n<phi.Nd(); ++n)
548     rtn2 += L2Norm2(phi[n], normalize);
549   if (!normalize)
550     rtn2 *= phi.Lx()*phi.Lz();
551   return sqrt(rtn2);
552 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

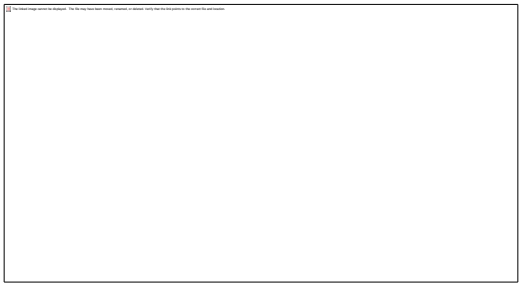
1.4.2.29 Real L2Norm2 (const BasisFunc & *phi*, bool *normalize*)

References L2Norm2(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

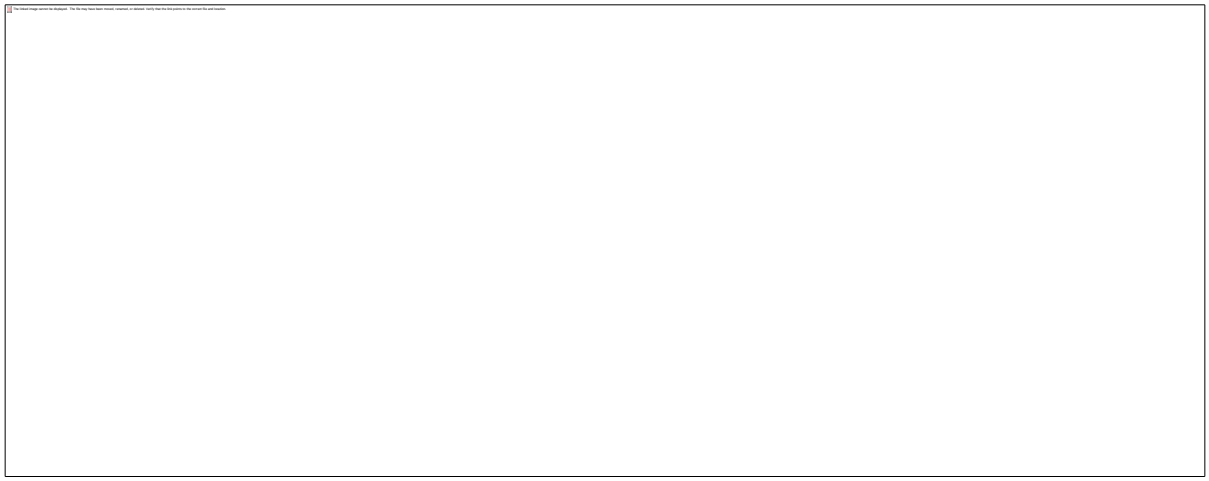
Referenced by divDist2(), divNorm2(), FlowField::energy(), L2Dist2(), L2Norm(), L2Norm2(), norm2(), and FlowField::saveSpectrum().

```
554                                     {
555   Real rtn = 0.0;
556   for (int n=0; n<phi.Nd(); ++n)
557     rtn += L2Norm2(phi[n], normalize);
558   if (!normalize)
559     rtn *= phi.Lx()*phi.Lz();
560   return rtn;
561 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.4.2.30 BasisFunc laplacian (const BasisFunc & *phi*)

References diff2(), BasisFunc::kx(), BasisFunc::kz(), lapl(), BasisFunc::Lx(), BasisFunc::Lz(), pi, square(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

```
774                                 {
775   BasisFunc lapl(phi);
776   Real c = -4*square(pi)*(square(phi.kx()/phi.Lx())+square(phi.kz()/phi.Lz()));
777   lapl.u() *= c;
778   lapl.v() *= c;
779   lapl.w() *= c;
780
781   ComplexChebyCoeff tmp;
782   diff2(phi.u(), tmp);
783   lapl.u() += tmp;
784   diff2(phi.v(), tmp);
785   lapl.v() += tmp;
786   diff2(phi.w(), tmp);
787   lapl.w() += tmp;
788   return lapl;
789 }
```

Here is the call graph for this function:

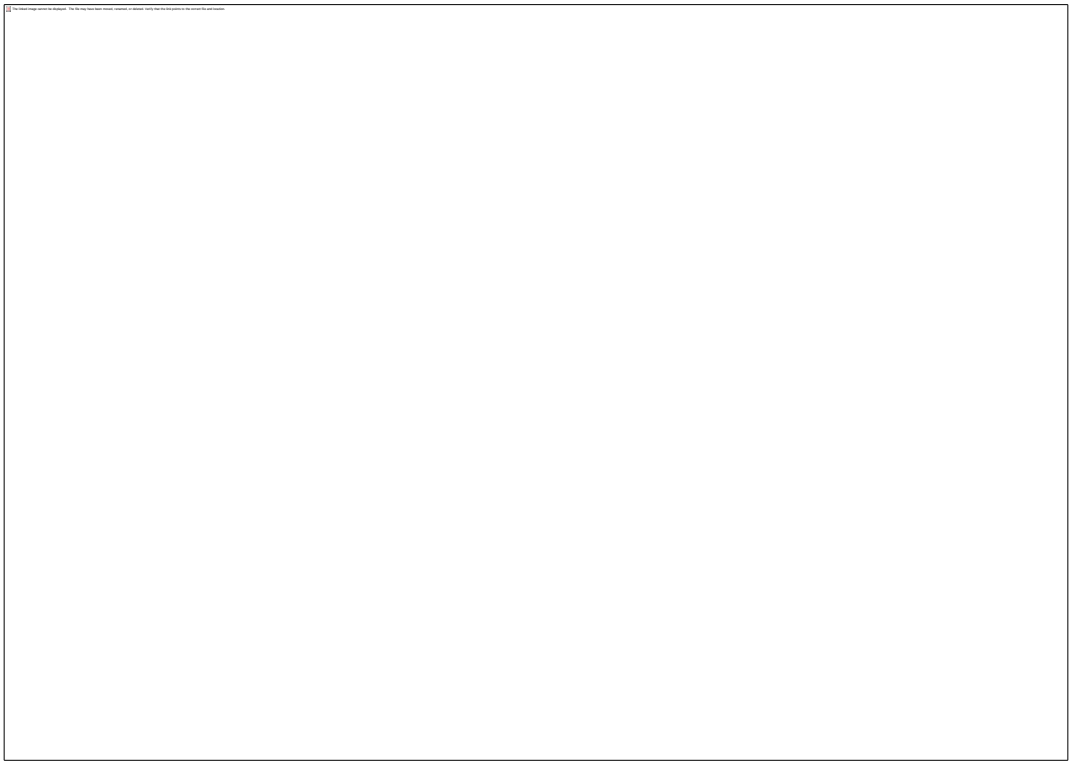
1.4.2.31 Real norm (const BasisFunc & *phi*, NormType *n*, bool *normalize*)

References chebyNorm(), L2Norm(), and L2NORM.

Referenced by abs2(), FlowField::addPerturbations(), norm(), and HelmholtzSolver::verify().

```
641                                                 {
642   return (n==L2NORM) ? L2Norm(phi,normalize) : chebyNorm(phi,normalize);
643 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

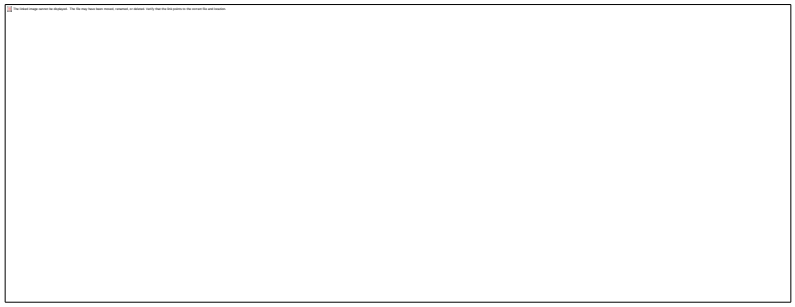
1.4.2.32 Real norm2 (const BasisFunc & *phi*, NormType *n*, bool *normalize*)

References chebyNorm2(), L2NORM, and L2Norm2().

Referenced by norm2().

```
645                                                 {
646   return (n==L2NORM) ? L2Norm2(phi,normalize) : chebyNorm2(phi,normalize);
647 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.4.2.33 bool operator!= (const BasisFunc & *f*, const BasisFunc & *g*)

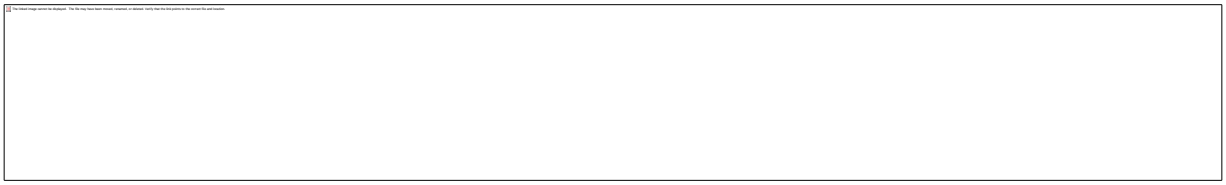
```
527                                                 {
528   return (f == g) ? false : true;
529 }
```

1.4.2.34 bool operator== (const BasisFunc & *f*, const BasisFunc & *g*)

References BasisFunc::congruent(), and BasisFunc::Nd().

```
531                                                 {
532   if (!f.congruent(g))
533     return false;
534
535   for (int n=0; n<f.Nd(); ++n)
536     if (f[n] != g[n])
537       return false;
538
539   return true;
540 }
```

Here is the call graph for this function:



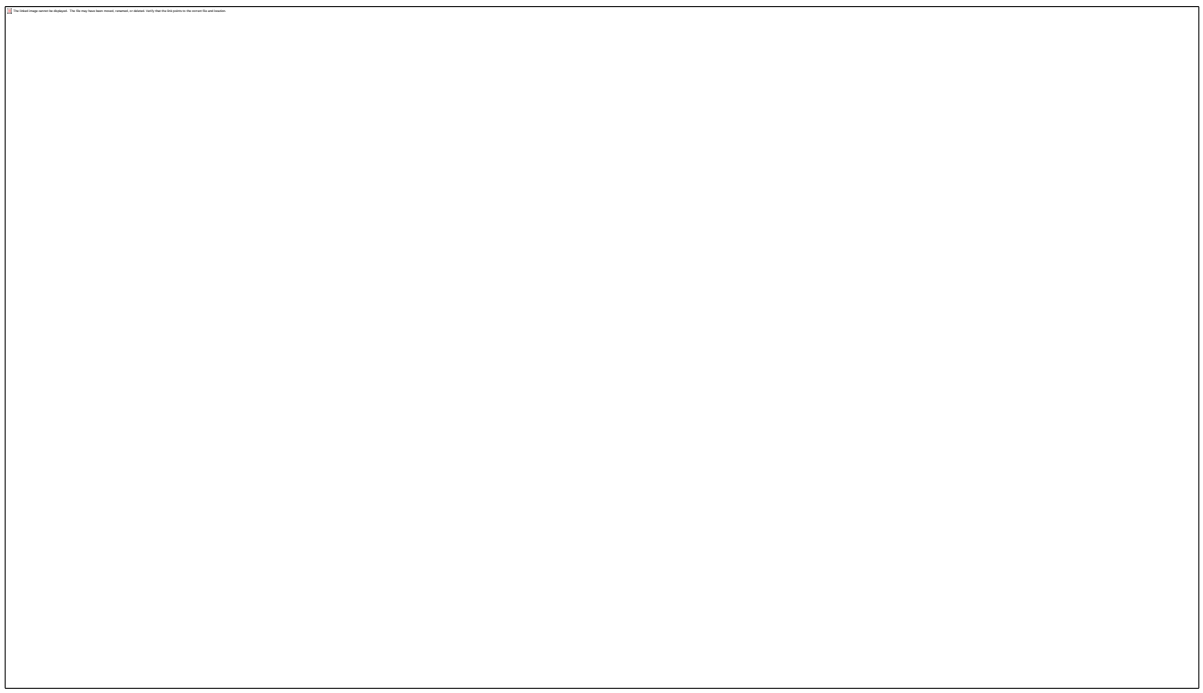
1.4.2.35 void ubcFix (ComplexChebyCoeff & *u*, bool *a_dirichlet*, bool *b_dirichlet*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), ComplexChebyCoeff::setBounds(), and ComplexChebyCoeff::sub().

```
1018                                                {
1019   Real ua = u.a();
1020   Real ub = u.b();
1021   u.setBounds(-1,1);
1022
1023   if (a_dirichlet && b_dirichlet) {
```

```
1024     u.sub(0, 0.5*(u.eval_b() + u.eval_a()));
1025     u.sub(1, 0.5*(u.eval_b() - u.eval_a()));
1026   }
1027   else if (a_dirichlet)
1028     u.sub(0, u.eval_a());
1029   else if (a_dirichlet)
1030     u.sub(0, u.eval_b());
1031
1032   u.setBounds(ua, ub);
1033 }
```

Here is the call graph for this function:



1.4.2.36 void ubcFix (ChebyCoeff & *u*, bool *a_dirichlet*, bool *b_dirichlet*)

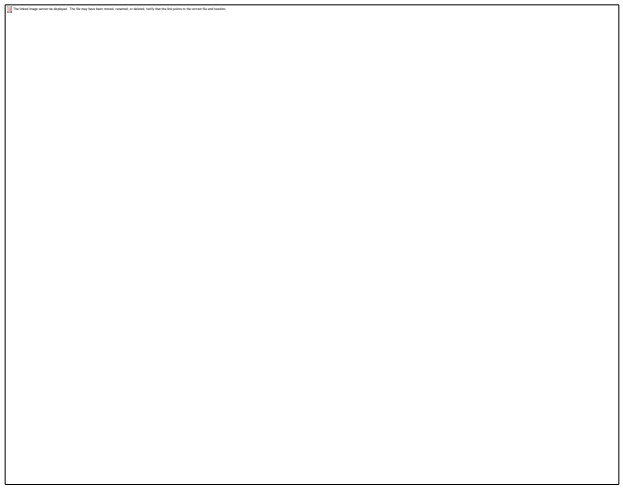
References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), and ChebyCoeff::setBounds().

Referenced by BasisFunc::randomize().

```
1001                                                {
1002   Real ua = u.a();
1003   Real ub = u.b();
1004   u.setBounds(-1,1);
1005
1006   if (a_dirichlet && b_dirichlet) {
1007     u[0] -= 0.5*(u.eval_b() + u.eval_a());
```

```
1008     u[1] -= 0.5*(u.eval_b() - u.eval_a());
1009   }
1010   else if (a_dirichlet)
1011     u[0] -= u.eval_a();
1012   else if (a_dirichlet)
1013     u[0] -= u.eval_b();
1014
1015   u.setBounds(ua, ub);
1016 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

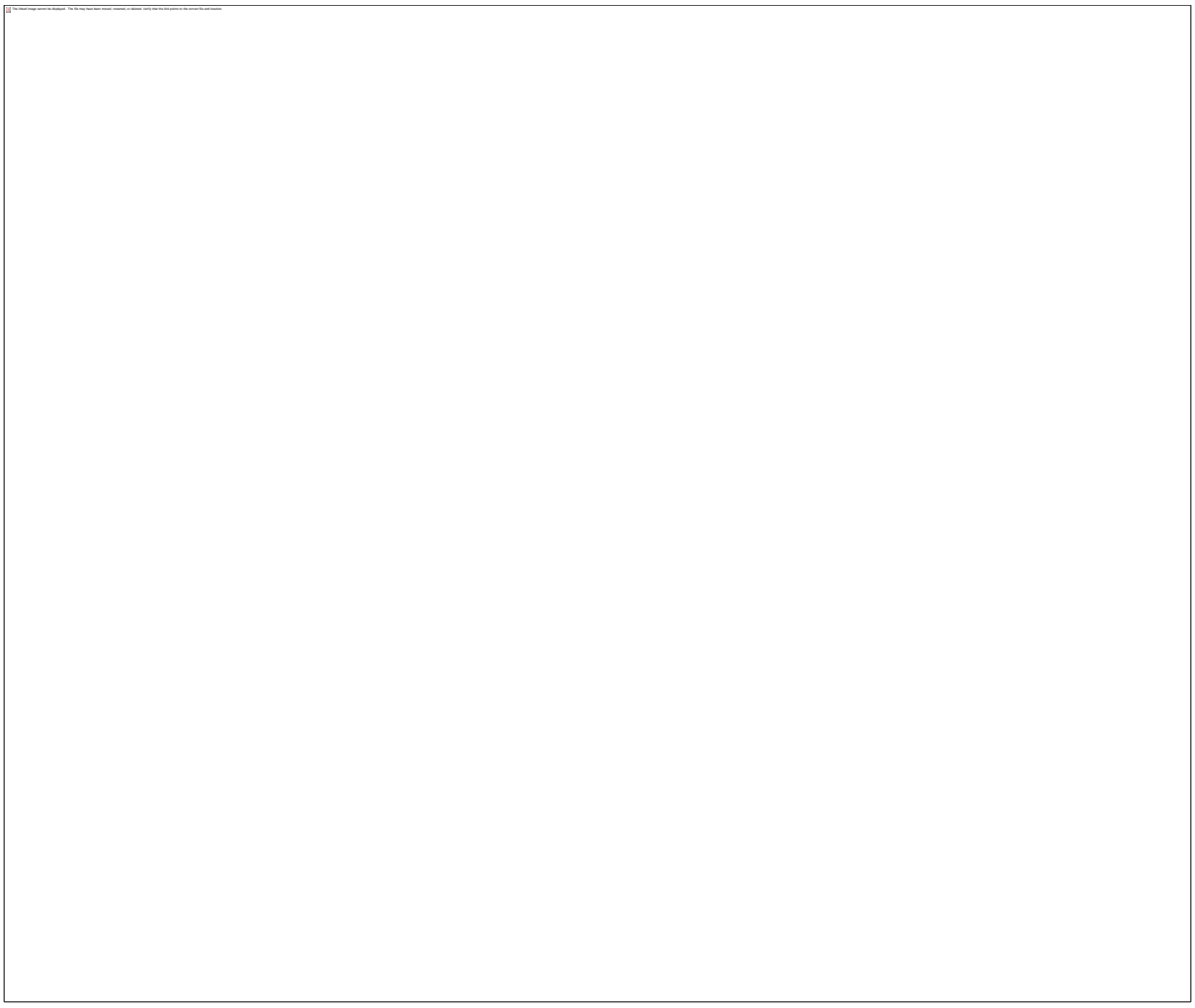
1.4.2.37 void vbcFix (ComplexChebyCoeff & *v*, bool *a_dirichlet*, bool *b_dirichlet*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), ComplexChebyCoeff::setBounds(), and ComplexChebyCoeff::sub().

```
1067                                                    {
1068
1069   Real va = v.a();
1070   Real vb = v.b();
1071   v.setBounds(-1,1);
1072
1073   if (a_dirichlet && b_dirichlet) {
1074     // Adjust v to match v(+-1)=v'(+-1)=0 BCs
```

```
1075    ComplexChebyCoeff vy = diff(v);
1076    Complex a = v.eval_a();
1077    Complex b = v.eval_b();
1078    Complex c = vy.eval_a();
1079    Complex d = vy.eval_b();
1080
1081    v.sub(0, 0.5*(a + b) + 0.125*(c - d));
1082    v.sub(1, 0.5625*(b - a) - 0.0625*(c + d));
1083    v.sub(2, 0.125*(d - c));
1084    v.sub(3, 0.0625*(a - b + c + d));
1085   }
1086   else if (a_dirichlet) {
1087    ComplexChebyCoeff vy = diff(v);
1088    v.sub(1, vy.eval_a());
1089    v.sub(0,  v.eval_a());
1090   }
1091   else if (b_dirichlet) {
1092    ComplexChebyCoeff vy = diff(v);
1093    v.sub(1, vy.eval_b());
1094    v.sub(0,  v.eval_b());
1095   }
1096   v.setBounds(va,vb);
1097 }
```

Here is the call graph for this function:



1.4.2.38 void vbcFix (ChebyCoeff & *v*, bool *a_dirichlet*, bool *b_dirichlet*)

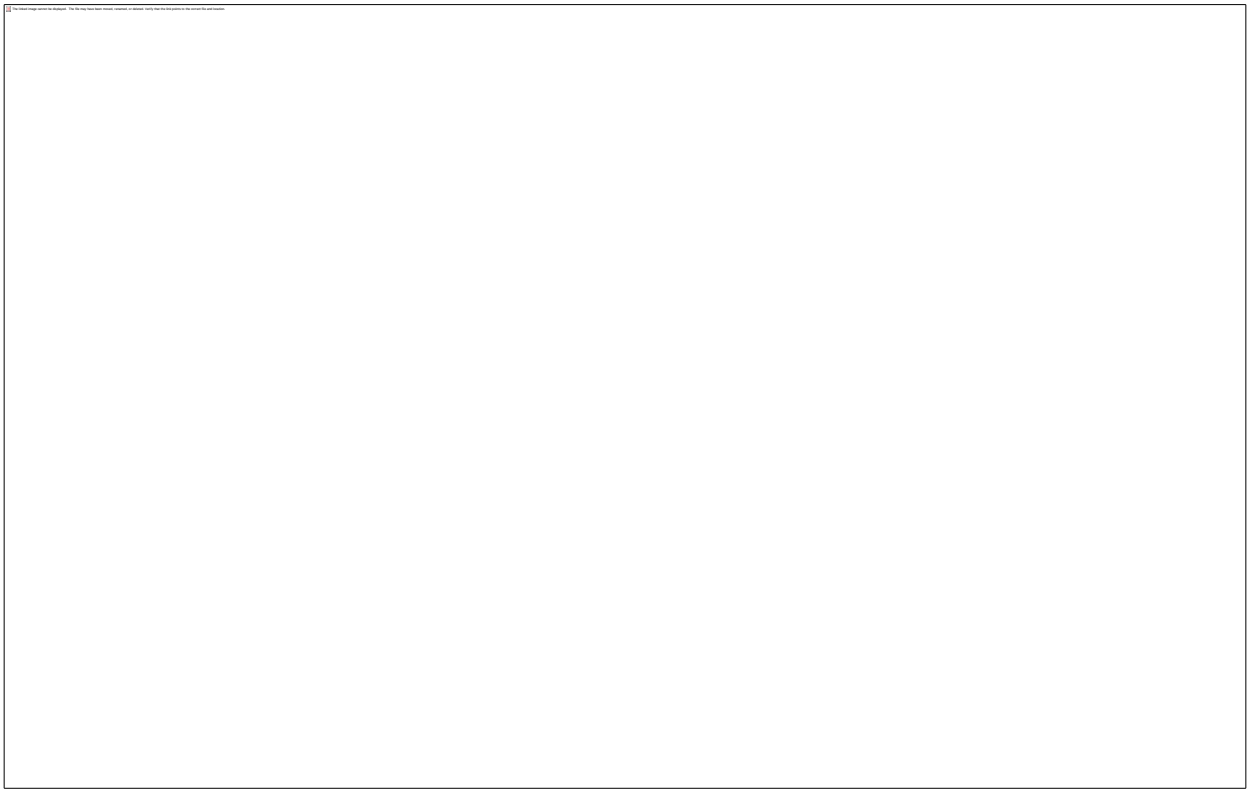
References ChebyCoeff::a(), ChebyCoeff::b(), diff(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), and ChebyCoeff::setBounds().

Referenced by BasisFunc::randomize().

```
1035                                                {
1036
1037   Real va = v.a();
1038   Real vb = v.b();
1039   v.setBounds(-1,1);
1040
1041   if (a_dirichlet && b_dirichlet) {
1042     // Adjust v to match v(+-1)=v'(+-1)=0 BCs
1043     ChebyCoeff vy = diff(v);
1044     Real a = v.eval_a();
1045     Real b = v.eval_b();
```

```
1046     Real c = vy.eval_a();
1047     Real d = vy.eval_b();
1048 
1049     v[0] -= 0.5*(a + b) + 0.125*(c - d);
1050     v[1] -= 0.5625*(b - a) - 0.0625*(c + d);
1051     v[2] -= 0.125*(d - c);
1052     v[3] -= 0.0625*(a - b + c + d);
1053   }
1054   else if (a_dirichlet) {
1055     ChebyCoeff vy = diff(v);
1056     v[1] -= vy.eval_a();
1057     v[0] -= v.eval_a();
1058   }
1059   else if (b_dirichlet) {
1060     ChebyCoeff vy = diff(v);
1061     v[1] -= vy.eval_b();
1062     v[0] -= v.eval_b();
1063   }
1064   v.setBounds(va,vb);
1065 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

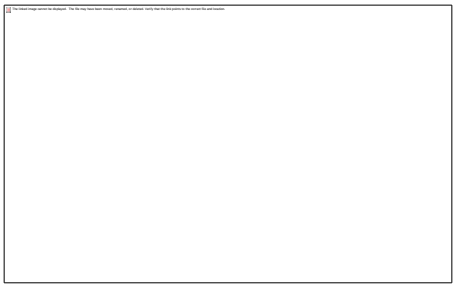
1.4.2.39 BasisFunc xdiff (const BasisFunc & *phi*)

References I(), BasisFunc::kx(), BasisFunc::Lx(), and pi.

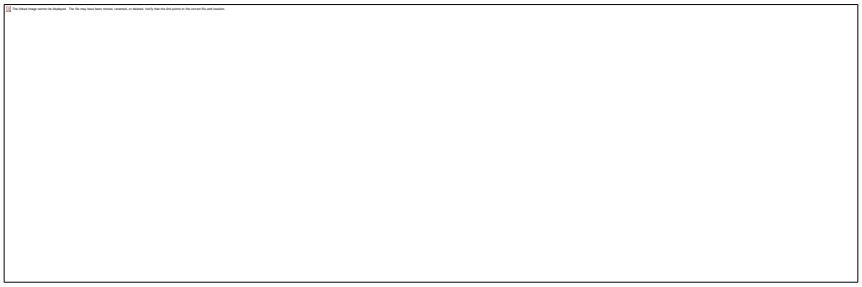
Referenced by polyInterp(), and xdiff().

```
733                                 {
734   BasisFunc phix(phi);
735   phix *= (2*pi*phi.kx()/phi.Lx())*I;
736   return phix;
737 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.4.2.40 BasisFunc ydiff (const BasisFunc & *phi*)

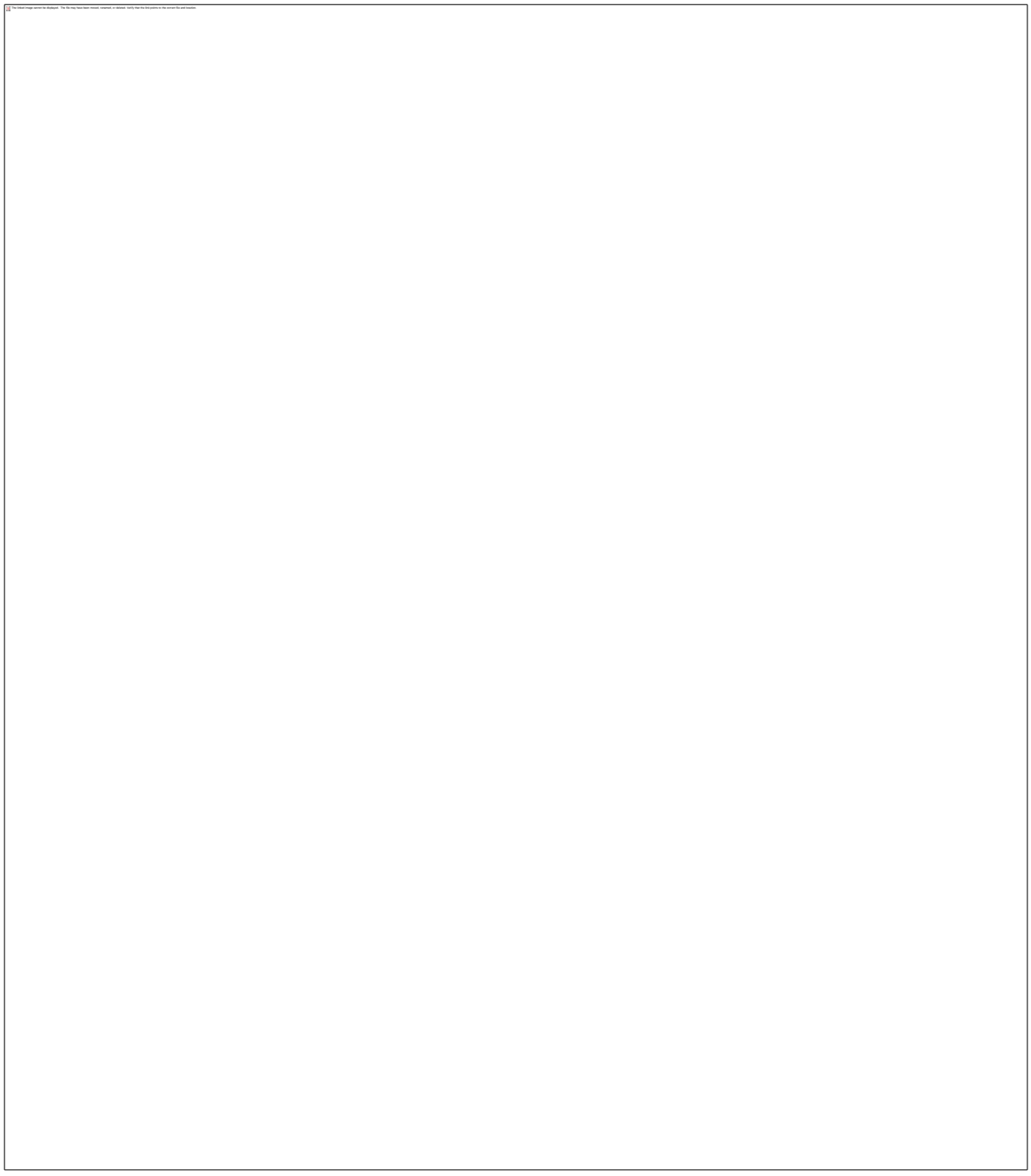
References BasisFunc::a(), BasisFunc::b(), diff(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::Ny(), Spectral, BasisFunc::state(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

Referenced by curl(), PressureSolver::verify(), and ydiff().

```
740                                 {
741   assert(phi.state() == Spectral);
```

```
742   BasisFunc phiy(phi.Ny(), phi.kx(), phi.kz(), phi.Lx(), phi.Lz(), phi.a(),
phi.b(), Spectral);
743   diff(phi.u(), phiy.u());
744   diff(phi.v(), phiy.v());
745   diff(phi.w(), phiy.w());
746   return phiy;
747 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

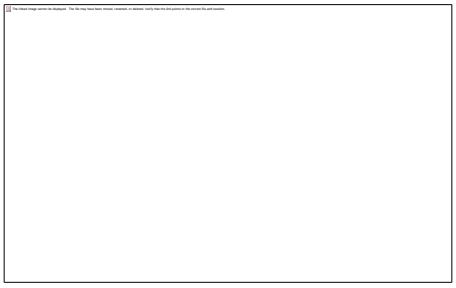
1.4.2.41 BasisFunc zdiff (const BasisFunc & *phi*)

References I(), BasisFunc::kz(), BasisFunc::Lz(), and pi.

Referenced by zdiff().

```
749                                 {
750   BasisFunc phiz(phi);
751   phiz *= (2*pi*phi.kz()/phi.Lz())*I;
752   return phiz;
753 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.4.2.42

1.5 /_cuda/channelflow-0.9.22/channelflow/basisfunc.h File Reference

#include "channelflow/mathdefs.h"
#include "channelflow/chebyshev.h"
#include "channelflow/array.h"
Include dependency graph for basisfunc.h:

This graph shows which files directly or indirectly include this file:

1.5.1 Classes

- class BasisFunc

1.5.2 Functions

- bool operator== (const BasisFunc &f, const BasisFunc &g)
- bool operator!= (const BasisFunc &f, const BasisFunc &g)
- Real L2Norm (const BasisFunc &f, bool normalize=true)
- Real L2Norm2 (const BasisFunc &f, bool normalize=true)
- Real L2Dist (const BasisFunc &f, const BasisFunc &g, bool normalize=true)
- Real L2Dist2 (const BasisFunc &f, const BasisFunc &g, bool normalize=true)
- Complex L2InnerProduct (const BasisFunc &f0, const BasisFunc &f1, bool normalize=true)
- Real chebyNorm (const BasisFunc &f, bool normalize=true)
- Real chebyNorm2 (const BasisFunc &f, bool normalize=true)

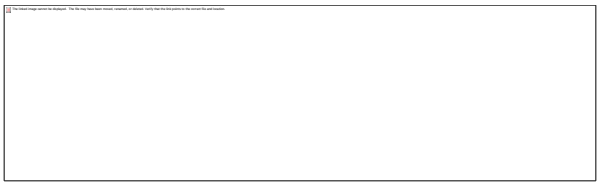
- Real chebyDist (const BasisFunc &f, const BasisFunc &g, bool normalize=true)
- Real chebyDist2 (const BasisFunc &f, const BasisFunc &g, bool normalize=true)
- Complex chebyInnerProduct (const BasisFunc &f0, const BasisFunc &f1, bool normalize=true)
- Real norm (const BasisFunc &f, NormType n, bool normalize=true)
- Real norm2 (const BasisFunc &f, NormType n, bool normalize=true)
- Real dist (const BasisFunc &f, const BasisFunc &g, NormType n, bool nrmlz=true)
- Real dist2 (const BasisFunc &f, const BasisFunc &g, NormType n, bool nrmlz=true)
- Complex innerProduct (const BasisFunc &f0, const BasisFunc &f1, NormType n, bool normalize=true)
- Real bcNorm (const BasisFunc &f, bool normalize=true)
- Real bcNorm2 (const BasisFunc &f, bool normalize=true)
- Real bcDist (const BasisFunc &f, const BasisFunc &g, bool normalize=true)
- Real bcDist2 (const BasisFunc &f, const BasisFunc &g, bool normalize=true)
- Real divNorm (const BasisFunc &f, bool normalize=true)
- Real divNorm2 (const BasisFunc &f, bool normalize=true)
- Real divDist (const BasisFunc &f, const BasisFunc &g, bool normalize=true)
- Real divDist2 (const BasisFunc &f, const BasisFunc &g, bool normalize=true)
- BasisFunc xdiff (const BasisFunc &f)
- BasisFunc ydiff (const BasisFunc &f)
- BasisFunc zdiff (const BasisFunc &f)
- BasisFunc laplacian (const BasisFunc &f)
- BasisFunc gradient (const BasisFunc &f, int i)
- BasisFunc gradient (const ComplexChebyCoeff &f, int kx, int kz, Real Lx, Real Lz)
- ComplexChebyCoeff divergence (const BasisFunc &f)
- void dotgrad (const BasisFunc &f_n, const BasisFunc &grad_f_pu, const BasisFunc &grad_f_pv, const BasisFunc &grad_f_pw, ComplexChebyCoeff &tmp, BasisFunc &rtn)
- BasisFunc dotgrad (const BasisFunc &f_n, const BasisFunc &f_p)
- ComplexChebyCoeff dot (const BasisFunc &f, const BasisFunc &g)
- BasisFunc cross (const BasisFunc &f, const BasisFunc &g)
- BasisFunc curl (const BasisFunc &f)
- void ubcFix (ChebyCoeff &u, bool a_dirichlet, bool b_dirichlet)
- void ubcFix (ComplexChebyCoeff &u, bool a_dirichlet, bool b_dirichlet)
- void vbcFix (ChebyCoeff &v, bool a_dirichlet, bool b_dirichlet)
- void vbcFix (ComplexChebyCoeff &v, bool a_dirichlet, bool b_dirichlet)

1.5.3 Function Documentation

1.5.3.1 Real bcDist (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize* = true)

Referenced by PressureSolver::verify(), and PoissonSolver::verify().

Here is the caller graph for this function:



1.5.3.2 Real bcDist2 (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize* = true)

References abs2(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

Referenced by bcDist().

```
716                                                 {
717   Real bc2 =
718     abs2(f.u().eval_a() - g.u().eval_a()) +
719     abs2(f.u().eval_b() - g.u().eval_b()) +
720     abs2(f.v().eval_a() - g.v().eval_a()) +
721     abs2(f.v().eval_b() - g.v().eval_b()) +
722     abs2(f.w().eval_a() - g.w().eval_a()) +
723     abs2(f.w().eval_b() - g.w().eval_b());
724   if (!normalize)
725     bc2 *= f.Lx()*f.Lz();
726   return bc2;
727 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

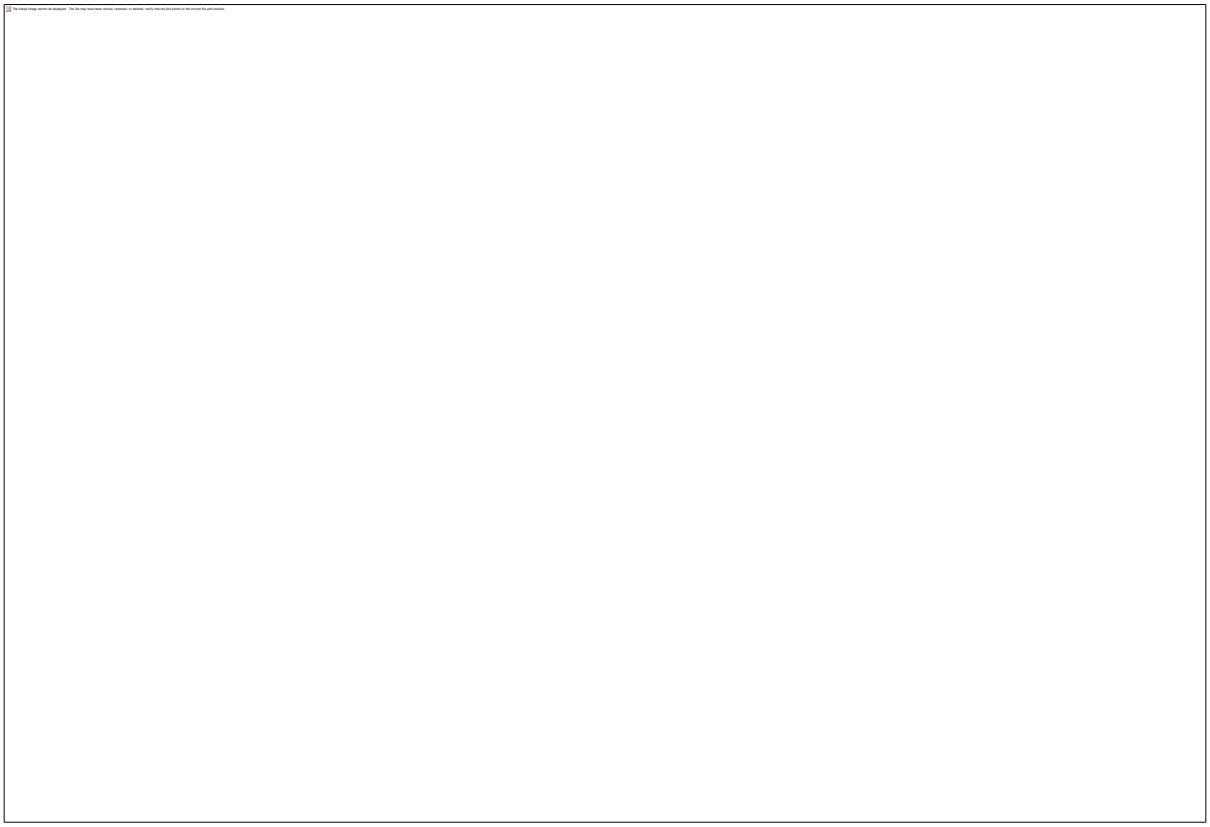
1.5.3.3 Real bcNorm (const BasisFunc & *f*, bool *normalize* = true)

References bcNorm2().

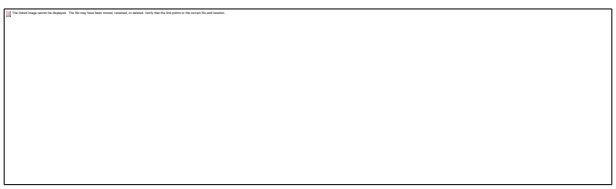
Referenced by PressureSolver::verify(), and PoissonSolver::verify().

```
712                                         {
713   return sqrt(bcNorm2(f, normalize));
714 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



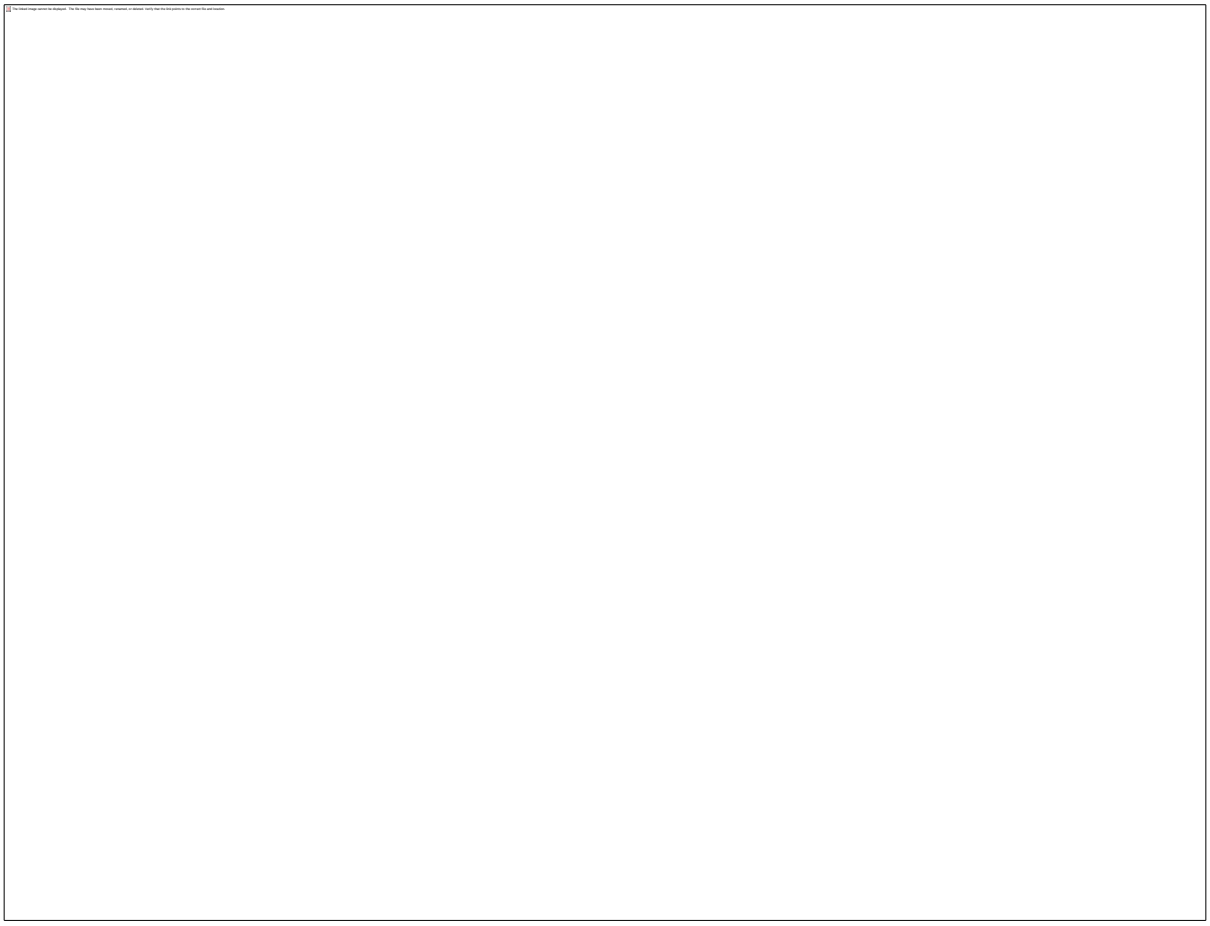
1.5.3.4 Real bcNorm2 (const BasisFunc & *f*, bool *normalize* = true)

References abs2(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

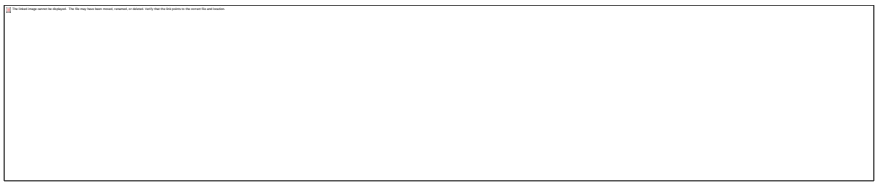
Referenced by bcNorm().

```
702                                         {
703   Real bc2 =
704     abs2(f.u().eval_a()) + abs2(f.u().eval_b()) +
705     abs2(f.v().eval_a()) + abs2(f.v().eval_b()) +
706     abs2(f.w().eval_a()) + abs2(f.w().eval_b());
707   if (!normalize)
708     bc2 *= f.Lx()*f.Lz();
709   return bc2;
710 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.5.3.5 Real chebyDist (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize* = true)

References chebyDist2().

Referenced by dist().

```
623                                                 {
624     return sqrt(chebyDist2(f,g,normalize));
625 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

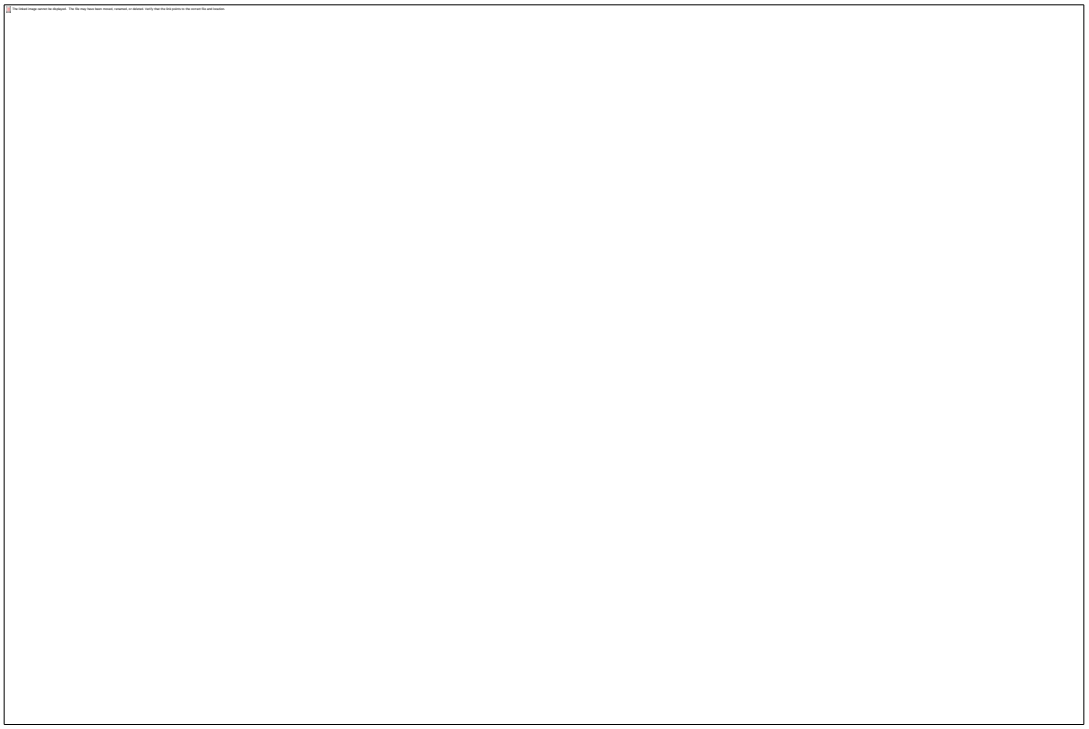
1.5.3.6 Real chebyDist2 (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize* = true)

References chebyDist2(), BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

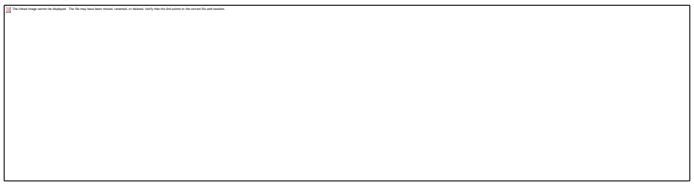
Referenced by chebyDist(), chebyDist2(), and dist2().

```
611                                                 {
612   assert(f.geometryCongruent(g));
613   Real rtn=0.0;
614   if (f.kx()==g.kx() && f.kz()==g.kz())
615     for (int n=0; n<f.Nd(); ++n)
616       rtn += chebyDist2(f[n], g[n], normalize);
617   if (!normalize)
618     rtn *= f.Lx()*f.Lz();
619   return rtn;
620 
621 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



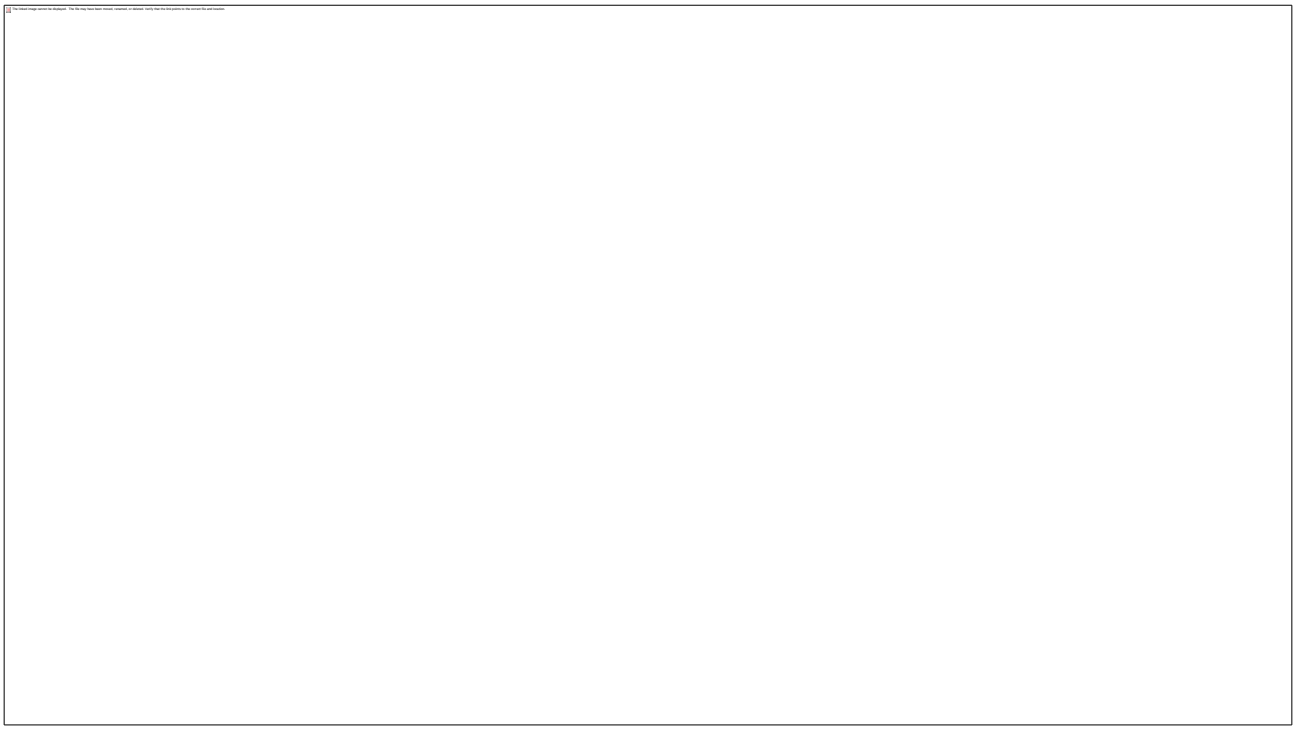
1.5.3.7 Complex chebyInnerProduct (const BasisFunc & *f0*, const BasisFunc & *f1*, bool *normalize* = true)

References chebyInnerProduct(), BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

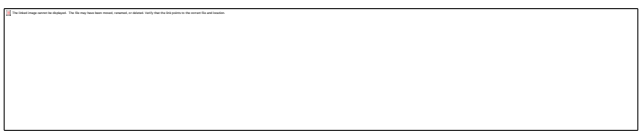
Referenced by chebyInnerProduct(), and innerProduct().

```
628                                 {
629   assert(f.geometryCongruent(g));
630   Complex rtn(0.0, 0.0);
631   if (f.kx()==g.kx() && f.kz()==g.kz())
632     for (int n=0; n<f.Nd(); ++n)
633       rtn += chebyInnerProduct(f[n], g[n], normalize);
634   if (!normalize)
635     rtn *= f.Lx() * f.Lz();
636   return rtn;
637 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



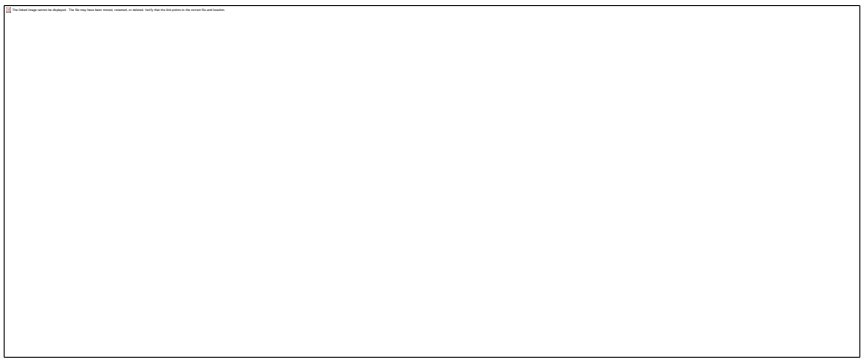
1.5.3.8 Real chebyNorm (const BasisFunc & *f*, bool *normalize* = true)

References chebyNorm2(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

Referenced by norm().

```
593                                         {
594   Real rtn2 = 0.0;
595   for (int n=0; n<phi.Nd(); ++n)
596     rtn2 += chebyNorm2(phi[n], normalize);
597   if (!normalize)
598     rtn2 *= phi.Lx()*phi.Lz();
599   return sqrt(rtn2);
600 }
```

Here is the call graph for this function:



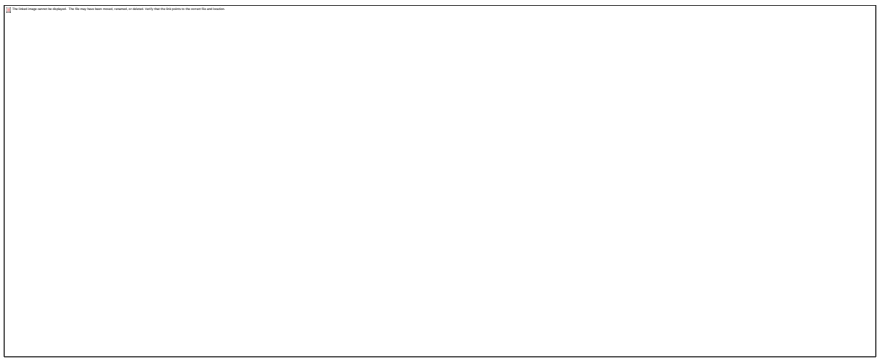
Here is the caller graph for this function:

1.5.3.9 Real chebyNorm2 (const BasisFunc & *f*, bool *normalize* = true)

References chebyNorm2(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().
Referenced by chebyNorm(), chebyNorm2(), and norm2().

```
602                                         {
603   Real rtn = 0.0;
604   for (int n=0; n<phi.Nd(); ++n)
605     rtn += chebyNorm2(phi[n], normalize);
606   if (!normalize)
607     rtn *= phi.Lx()*phi.Lz();
608   return rtn;
609 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.5.3.10 BasisFunc cross (const BasisFunc & *f*, const BasisFunc & *g*)

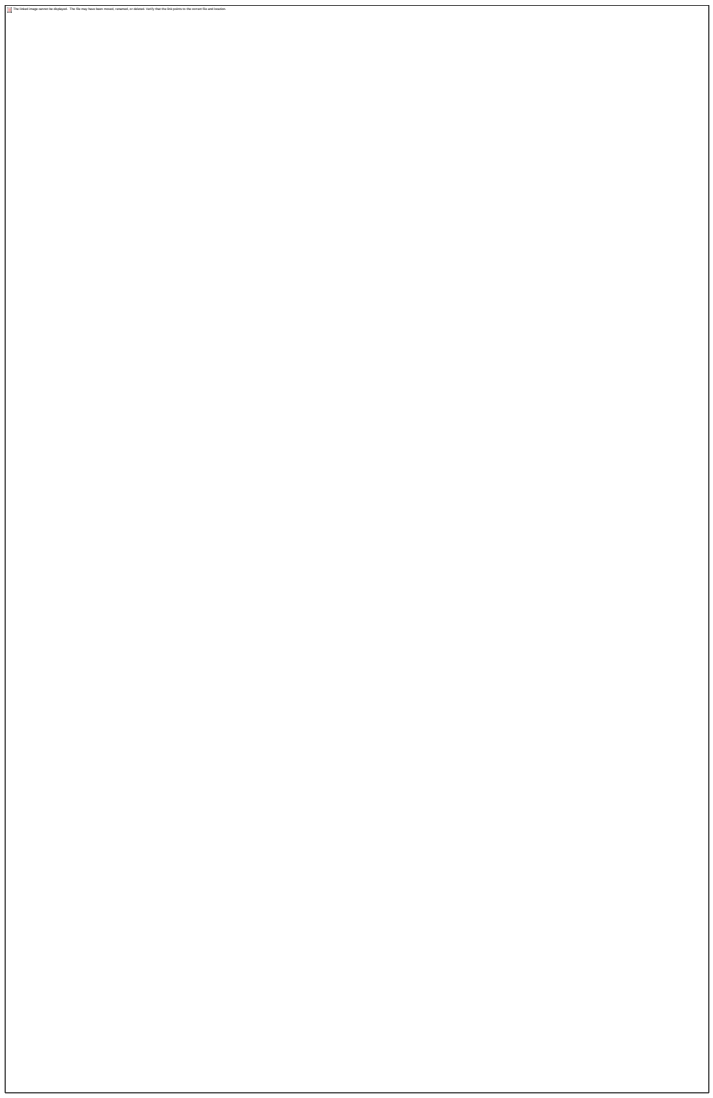
References BasisFunc::a(), BasisFunc::b(), BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::makePhysical(), BasisFunc::makeSpectral(), BasisFunc::Ny(), Physical, Spectral, and BasisFunc::state().

Referenced by cross().

```
842                                         {
843   BasisFunc& f = (BasisFunc&) f_;
844   BasisFunc& g = (BasisFunc&) g_;
845   assert(f.geometryCongruent(g));
846   int Ny= f.Ny();
847
848   ChebyTransform t(Ny);
849   fieldstate entrystate = f.state();
850   f.makePhysical(t);
851   g.makePhysical(t);
852
853
854   BasisFunc rtn(Ny, f.kx()+g.kx(), f.kz()+g.kz(), f.Lx(), f.Lz(), f.a(), f.b(),
```

```
855                 Physical);
856   for (int ny=0; ny<Ny; ++ny) {
857     Complex fu = f[0][ny];
858     Complex fv = f[1][ny];
859     Complex fw = f[2][ny];
860     Complex gu = g[0][ny];
861     Complex gv = g[1][ny];
862     Complex gw = g[2][ny];
863
864     rtn[0].set(ny, fv*gw - fw*gv);
865     rtn[1].set(ny, fw*gu - fu*gw);
866     rtn[2].set(ny, fu*gv - fv*gu);
867   }
868
869   if (entrystate==Spectral) {
870     f.makeSpectral(t);
871     g.makeSpectral(t);
872     rtn.makeSpectral(t);
873   }
874   return rtn;
875 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.5.3.11 BasisFunc curl (const BasisFunc & *f*)

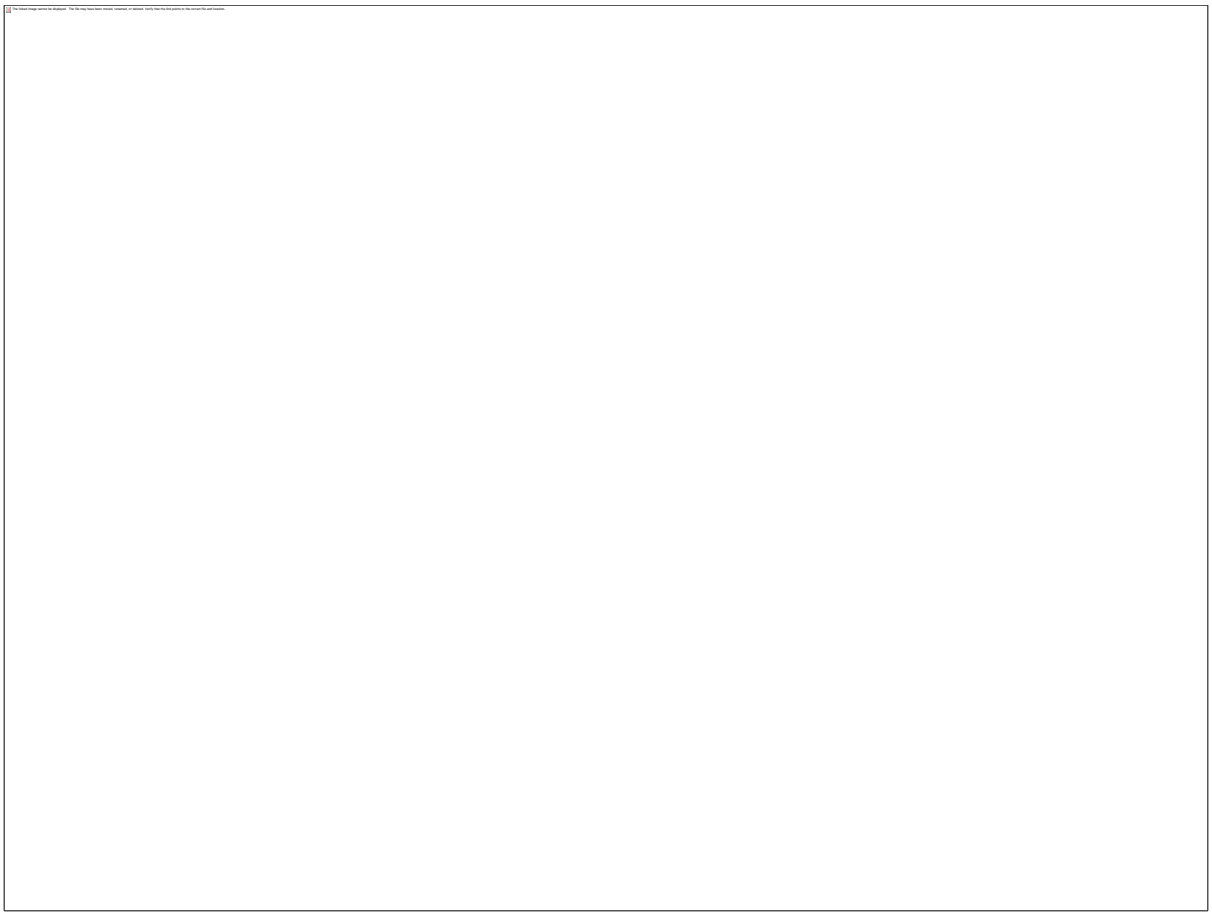
References I(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::makeSpectral(), BasisFunc::Ny(), pi, BasisFunc::setToZero(), Spectral, BasisFunc::state(), and ydiff().

Referenced by curl(), and rotationalNL().

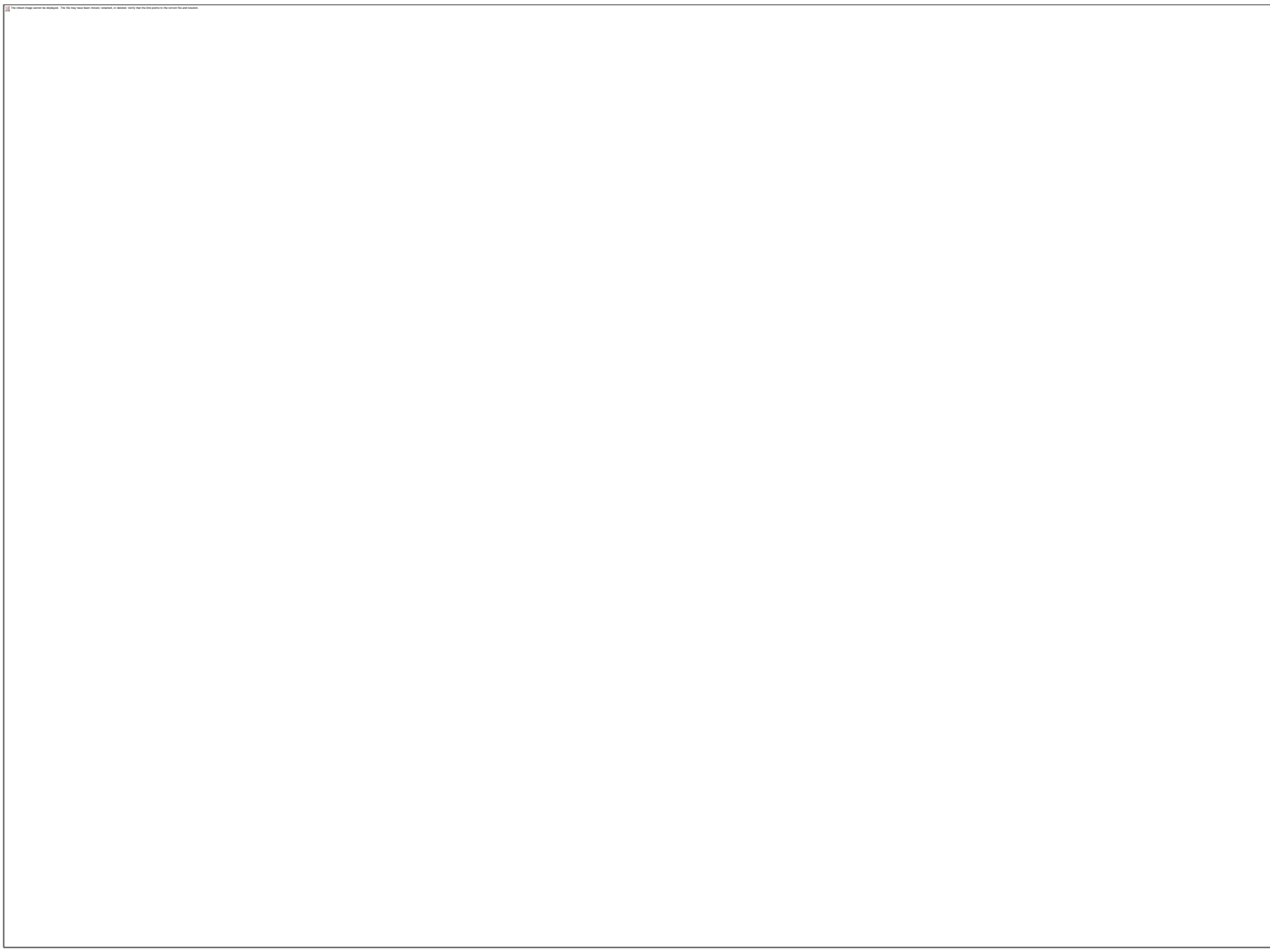
```
877                             {
878
879   BasisFunc& f = (BasisFunc&) f_;
```

```
880   fieldstate entrystate = f.state();
881   int Ny= f.Ny();
882   ChebyTransform t(Ny);
883   f.makeSpectral(t);
884
885   BasisFunc f_y = ydiff(f);
886   Complex d_dx = (2*pi*f.kx()/f.Lx())*I;
887   Complex d_dz = (2*pi*f.kz()/f.Lz())*I;
888
889   BasisFunc rtn(f);
890
891   rtn.setToZero();
892   for (int ny=0; ny<Ny; ++ny) {
893     Complex u  = f[0][ny];
894     Complex v  = f[1][ny];
895     Complex w  = f[2][ny];
896     Complex u_y = f_y[0][ny];
897     Complex w_y = f_y[2][ny];
898
899     rtn[0].set(ny, w_y - d_dz*v);
900     rtn[1].set(ny, d_dz*u - d_dx*w);
901     rtn[2].set(ny, d_dx*v - u_y);
902   }
903
904   if (entrystate==Spectral) {
905     f.makeSpectral(t);
906     rtn.makeSpectral(t);
907   }
908   return rtn;
909 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.5.3.12 Real dist (const BasisFunc & *f*, const BasisFunc & *g*, NormType *n*, bool *nrmlz* = true)

References chebyDist(), L2Dist(), and L2NORM.

```
653                                                 {
654   return (n==L2NORM) ? L2Dist(f,g,nrmlz) : chebyDist(f,g,nrmlz);
655 }
```

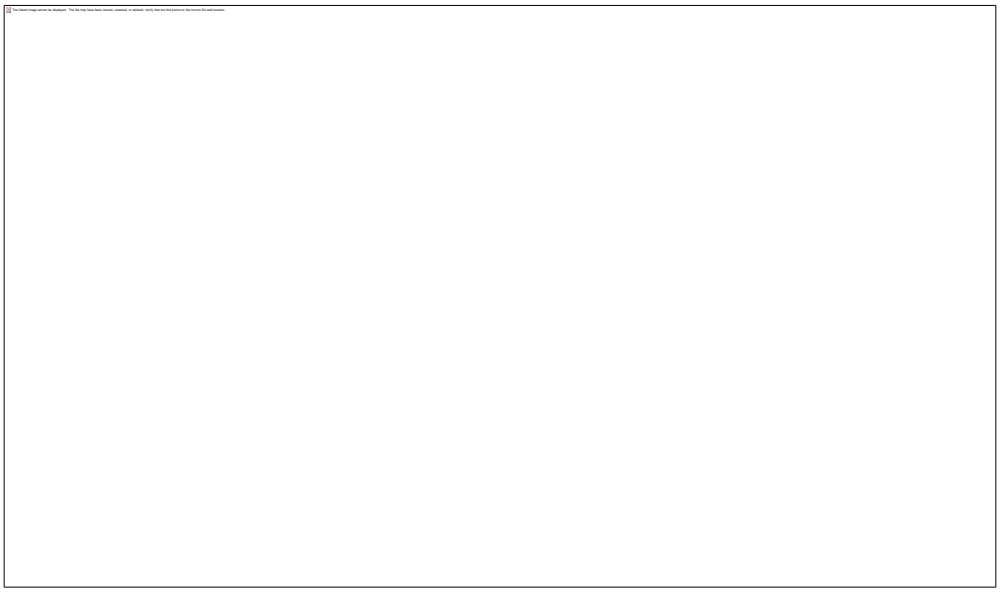
Here is the call graph for this function:

1.5.3.13 Real dist2 (const BasisFunc & *f*, const BasisFunc & *g*, NormType *n*, bool *nrmlz* = true)

References chebyDist2(), L2Dist2(), and L2NORM.

```
649                                                 {
650   return (n==L2NORM) ? L2Dist2(f,g,nrmlz) : chebyDist2(f,g,nrmlz);
651 }
```

Here is the call graph for this function:



1.5.3.14 Real divDist (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize* = true)

References divDist2().

```
698                                                 {
699   return sqrt(divDist2(f,g, normalize));
700 }
```

Here is the call graph for this function:

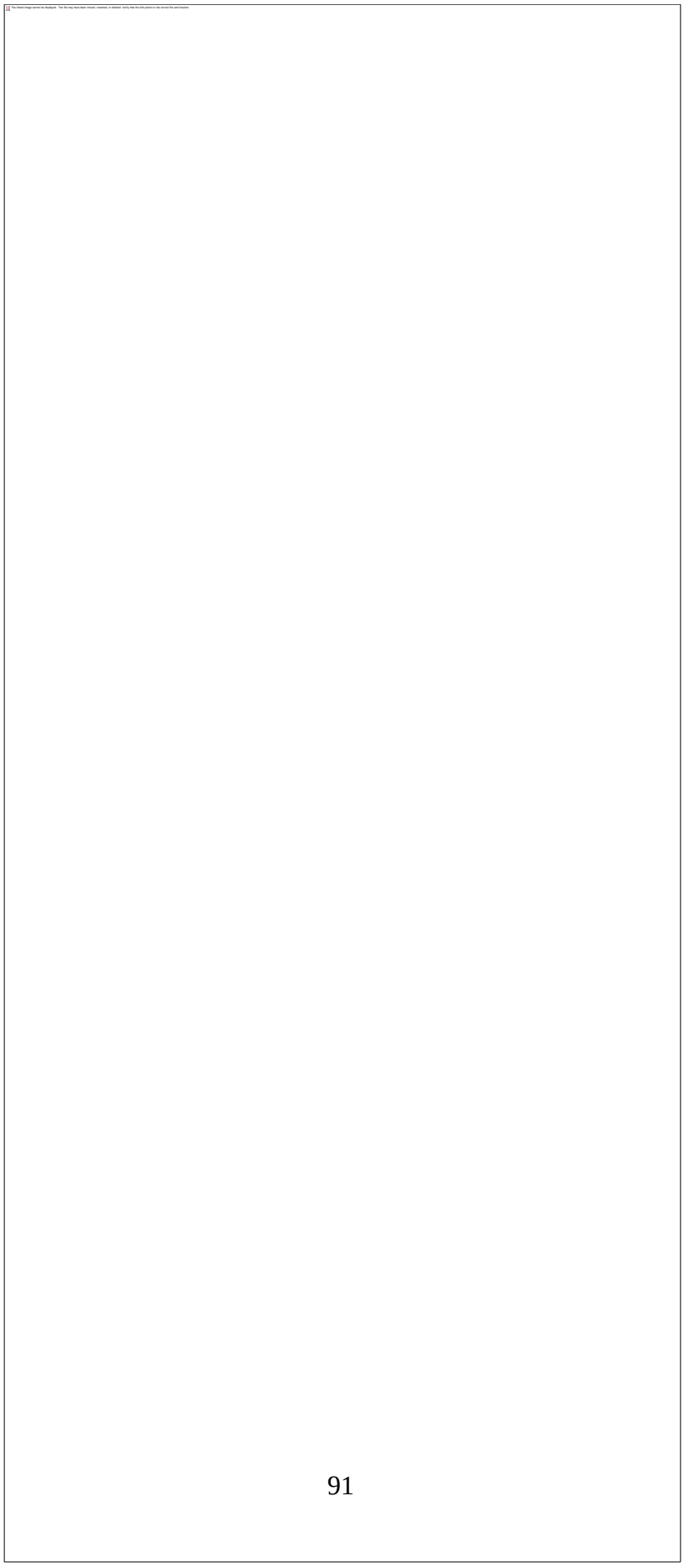
1.5.3.15 Real divDist2 (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize* = true)

References BasisFunc::congruent(), diff(), div(), I(), BasisFunc::kx(), BasisFunc::kz(), L2Norm2(), BasisFunc::Lx(), BasisFunc::Lz(), pi, Spectral, BasisFunc::state(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

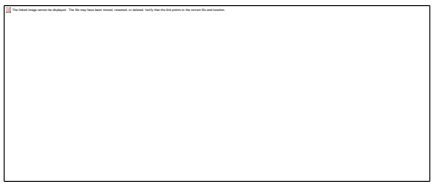
Referenced by divDist(), and divDist2().

```
681                                                 {
682   assert(f.state() == Spectral && g.state() == Spectral);
683   assert(f.congruent(g));
684   ComplexChebyCoeff div = f.u();
685   div -= g.u();
686   div *= (2*pi*f.kx()/f.Lx())*I;
687   ComplexChebyCoeff tmp = f.w();
688   tmp -= g.w();
689   tmp *= (2*pi*f.kz()/f.Lz())*I;
690   div += tmp;
691   diff(f.v(), tmp);
692   div += tmp;
693   diff(g.v(), tmp);
694   div -= tmp;
695   return L2Norm2(div, normalize);
696 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.5.3.16 ComplexChebyCoeff divergence (const BasisFunc & *f*)

References BasisFunc::a(), BasisFunc::b(), diff(), div(), I(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::Ny(), pi, Spectral, and BasisFunc::state().

```
755                                         {
756   assert(phi.state() == Spectral);
757   ComplexChebyCoeff div(phi.Ny(), phi.a(), phi.b(), Spectral);
758   ComplexChebyCoeff tmp(phi.Ny(), phi.a(), phi.b(), Spectral);
759   tmp = phi[0];
760   tmp *= (2*pi*phi.kx()/phi.Lx())*I;
761   div += tmp;
762 
763   diff(phi[1], tmp);
764   div += tmp;
765 
766   tmp = phi[2];
767   tmp *= (2*pi*phi.kz()/phi.Lz())*I;
768   div += tmp;
769 
770   return div;
771 }
```

Here is the call graph for this function:

1.5.3.17 Real divNorm (const BasisFunc & *f*, bool *normalize* = true)

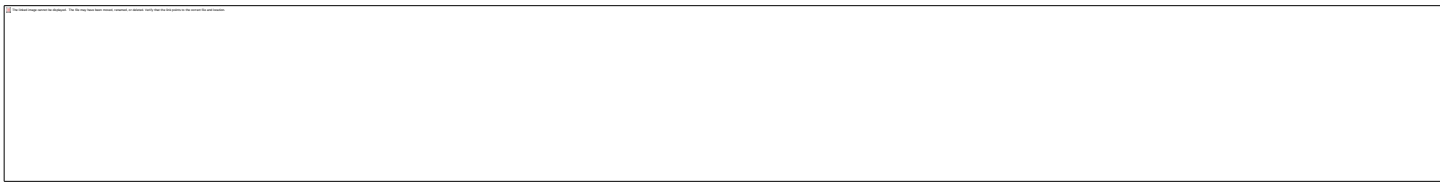
References divNorm2().

Referenced by chebyProfile(), and randomProfile().

```
677                                     {
678   return sqrt(divNorm2(f, normalize));
679 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.5.3.18 Real divNorm2 (const BasisFunc & *f*, bool *normalize* = true)

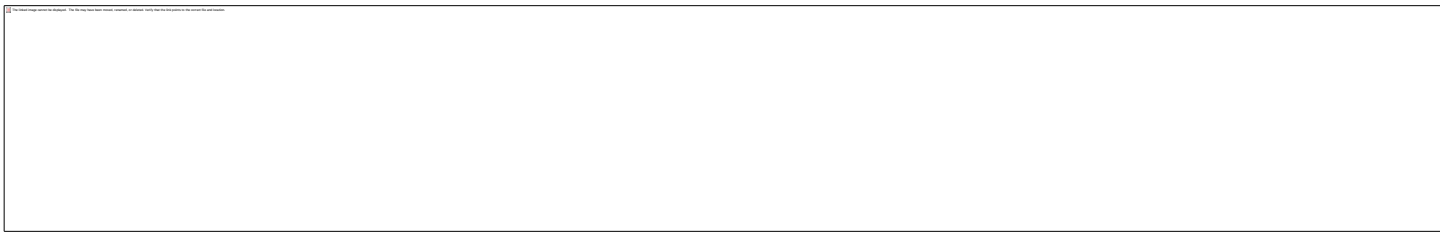
References diff(), div(), I(), BasisFunc::kx(), BasisFunc::kz(), L2Norm2(), BasisFunc::Lx(), BasisFunc::Lz(), pi, Spectral, BasisFunc::state(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

Referenced by divNorm(), and divNorm2().

```
665                                         {
666   assert(f.state() == Spectral);
667   ComplexChebyCoeff div = f.u();
668   div *= (2*pi*f.kx()/f.Lx())*I;
669   ComplexChebyCoeff tmp = f.w();
670   tmp *= (2*pi*f.kz()/f.Lz())*I;
671   div += tmp;
672   diff(f.v(), tmp);
673   div += tmp;
674   return L2Norm2(div, normalize);
675 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



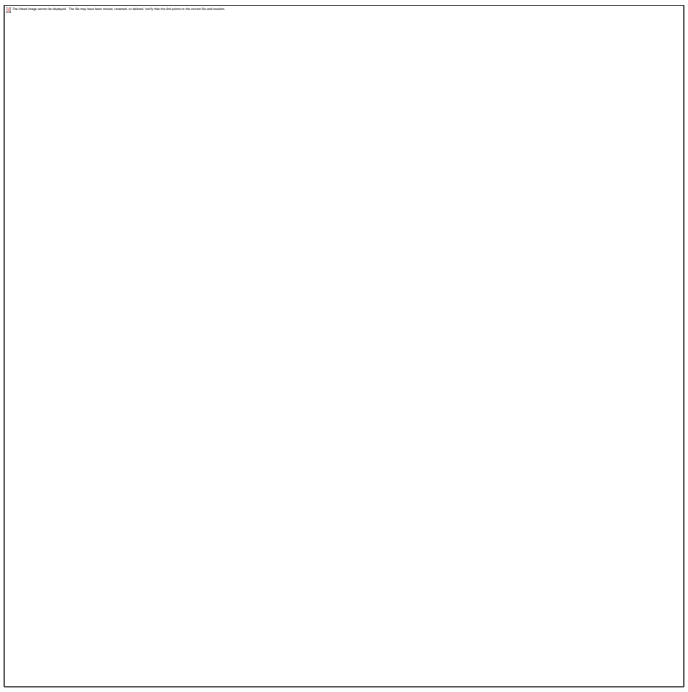
1.5.3.19 ComplexChebyCoeff dot (const BasisFunc & *f*, const BasisFunc & *g*)

References BasisFunc::a(), BasisFunc::b(), BasisFunc::geometryCongruent(), BasisFunc::makePhysical(), BasisFunc::makeState(), BasisFunc::Ny(), Physical, and BasisFunc::state().

Referenced by dot().

```
819                                                 {
820   BasisFunc& f = (BasisFunc&) f_;
821   BasisFunc& g = (BasisFunc&) g_;
822   assert(f.geometryCongruent(g));
823   int Ny= f.Ny();
824   fieldstate fstate = f.state();
825   fieldstate gstate = g.state();
826   ChebyTransform t(Ny);
827   f.makePhysical(t);
828   g.makePhysical(t);
829
830   ComplexChebyCoeff fg(Ny, f.a(), f.b(), Physical);
831   for (int ny=0; ny<Ny; ++ny)
832     fg.set(ny, f[0][ny]*g[0][ny] + f[1][ny]*g[1][ny] + f[2][ny]*g[2][ny]);
833
834
835   f.makeState(fstate,t);
836   g.makeState(gstate,t);
837   fg.makeState(fstate, t);
838   return fg;
839 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.5.3.20 BasisFunc dotgrad (const BasisFunc & *f_n*, const BasisFunc & *f_p*)

References BasisFunc::a(), BasisFunc::b(), dotgrad(), gradient(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::makePhysical(), BasisFunc::makeSpectral(), BasisFunc::Ny(), Physical, Spectral, and BasisFunc::state().

```
963                                                     {
964   BasisFunc& phi_n = (BasisFunc&) phi_n_arg;
965   BasisFunc& phi_p = (BasisFunc&) phi_p_arg;
966
967   fieldstate original_n_state = phi_n.state();
968   fieldstate original_p_state = phi_p.state();
969
970   int Ny=phi_n.Ny();
971   ChebyTransform trans(Ny);
972   phi_p.makeSpectral(trans);   // Now phi_p is spectral
973
974   BasisFunc rtn(Ny, phi_n.kx()+phi_p.kx(), phi_n.kz()+phi_p.kz(),
975                 phi_n.Lx(), phi_n.Lz(), phi_n.a(), phi_n.b(), Spectral);
976
977   BasisFunc grad_phi_pu = gradient(phi_p, 0);
```

```
978   BasisFunc grad_phi_pv = gradient(phi_p, 1);
979   BasisFunc grad_phi_pw = gradient(phi_p, 2);
980   ComplexChebyCoeff tmp(Ny, phi_n.a(), phi_n.b(), Physical);
981
982   phi_n.makePhysical(trans);  // Now phi_n is physical
983   grad_phi_pu.makePhysical(trans);
984   grad_phi_pv.makePhysical(trans);
985   grad_phi_pw.makePhysical(trans);
986
987   dotgrad(phi_n, grad_phi_pu, grad_phi_pv, grad_phi_pw, tmp, rtn);
988
989   // phi_n is physical,
990   if (original_n_state == Spectral) {
991     phi_n.makeSpectral(trans);
992     rtn.makeSpectral(trans);
993   }
994   // phi_p is spectral.
995   if (original_p_state == Physical)
996     phi_p.makePhysical(trans);
997
998   return rtn;
999 }
```

Here is the call graph for this function:

1.5.3.21 void dotgrad (const BasisFunc & *f_n*, const BasisFunc & *grad_f_pu*, const BasisFunc & *grad_f_pv*, const BasisFunc & *grad_f_pw*, ComplexChebyCoeff & *tmp*, BasisFunc & *rtn*)

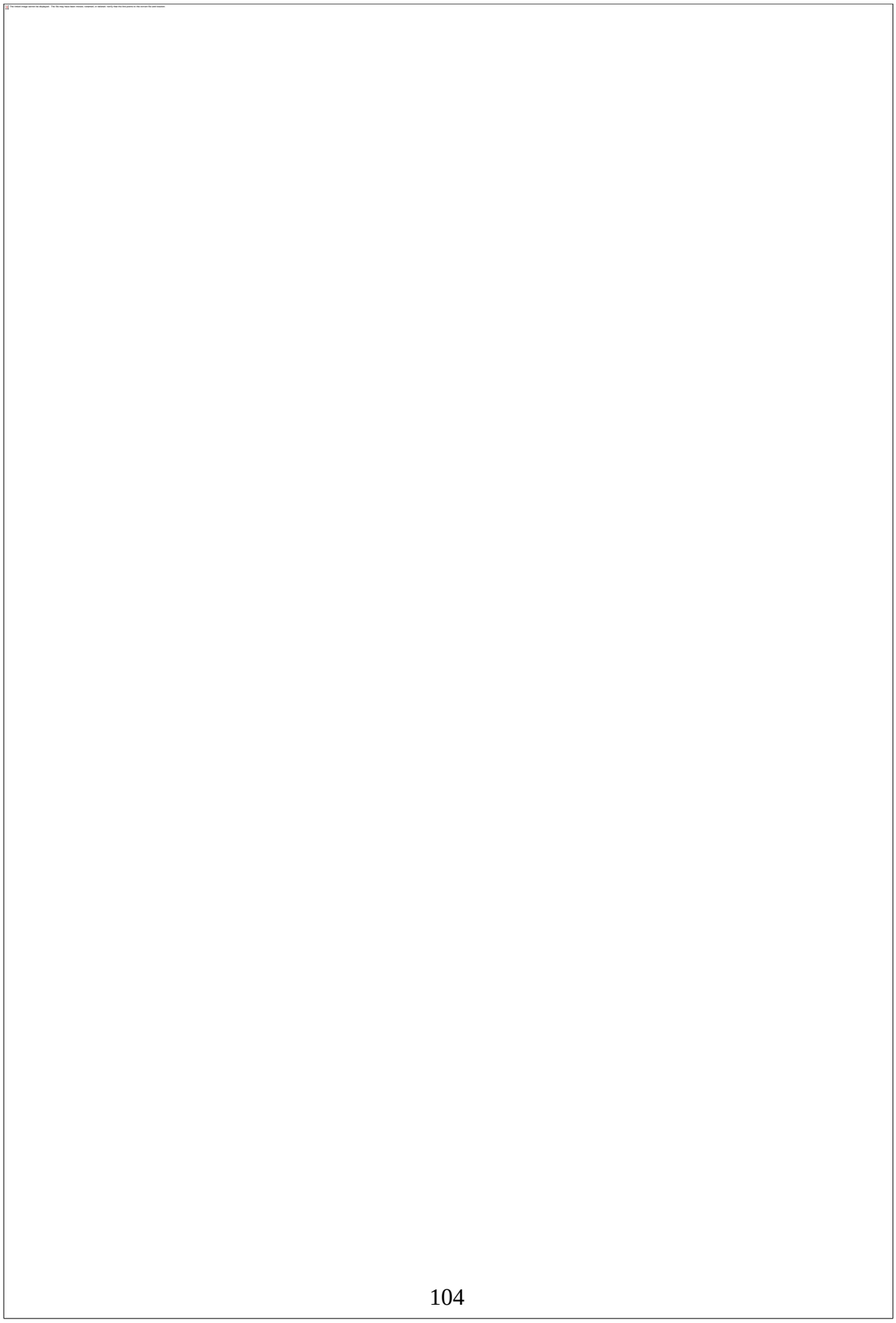
References BasisFunc::a(), BasisFunc::b(), BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), Physical, ComplexChebyCoeff::setBounds(), BasisFunc::setBounds(), BasisFunc::setkxkz(), ComplexChebyCoeff::setState(), BasisFunc::setState(), ComplexChebyCoeff::setToZero(), BasisFunc::setToZero(), BasisFunc::state(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

Referenced by dotgrad().

```
915 {
916   assert(phi_n.state() == Physical);
917   assert(grad_phi_pu.state() == Physical);
918   assert(grad_phi_pv.state() == Physical);
919   assert(grad_phi_pw.state() == Physical);
920   assert(phi_n.geometryCongruent(grad_phi_pu));
921   assert(phi_n.geometryCongruent(grad_phi_pv));
922   assert(phi_n.geometryCongruent(grad_phi_pw));
923   rtn.setToZero();
924   rtn.setkxkz(phi_n.kx()+grad_phi_pu.kx(), phi_n.kz()+grad_phi_pu.kz());
925   rtn.setBounds(phi_n.Lx(), phi_n.Lz(), phi_n.a(), phi_n.b());
926   rtn.setState(Physical);
927
928   tmp.setToZero();
929   tmp.setBounds(phi_n.a(), phi_n.b());
930   tmp.setState(Physical);
931
932   // ====================================
933   // Calculate rtn.u =   phi_n.u d/dx phi_p.u
934   //                   + phi_n.v d/dy phi_p.u
935   //                   + phi_n.w d/dz phi_p.u
936   int i; // MSVC++ FOR-SCOPE BUG
937   for (i=0; i<3; ++i) {
938     tmp = phi_n[i];
939     tmp *= grad_phi_pu[i];
940     rtn.u() += tmp;
941   }
942   // ====================================
943   // Calculate rtn.v =   phi_n.u d/dx phi_p.v
944   //                   + phi_n.v d/dy phi_p.v
945   //                   + phi_n.w d/dz phi_p.v
946   for (i=0; i<3; ++i) {
947     tmp = phi_n[i];
948     tmp *= grad_phi_pv[i];
949     rtn.v() += tmp;
950   }
951   // ====================================
952   // Calculate rtn.w =   phi_n.u d/dx phi_p.w
953   //                   + phi_n.v d/dy phi_p.w
```

```
954   //                  + phi_n.w d/dz phi_p.w
955   for (i=0; i<3; ++i) {
956     tmp = phi_n[i];
957     tmp *= grad_phi_pw[i];
958     rtn.w() += tmp;
959   }
960   return;
961 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.5.3.22 BasisFunc gradient (const ComplexChebyCoeff & *f*, int *kx*, int *kz*, Real *Lx*, Real *Lz*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff(), grad(), I(), ComplexChebyCoeff::numModes(), pi, Spectral, and ComplexChebyCoeff::state().

Referenced by dotgrad().

```
791                                                    {
792   assert(f.state() == Spectral);
793   BasisFunc grad(f.numModes(), kx, kz, Lx, Lz, f.a(), f.b(), f.state());
794   grad.u() = f;
795   grad.u() *= (2*pi*kx/Lx)*I;
796   diff(f, grad.v());
797   grad.w() = f;
798   grad.w() *= (2*pi*kz/Lz)*I;
799   return grad;
800 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.5.3.23 BasisFunc gradient (const BasisFunc & *f*, int *i*)

References BasisFunc::a(), BasisFunc::b(), diff(), grad(), I(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::Ny(), pi, Spectral, and BasisFunc::state().

```
802                                                 {
803   assert(phi.state() == Spectral);
804
805   BasisFunc grad(phi.Ny(), phi.kx(), phi.kz(), phi.Lx(), phi.Lz(), phi.a(), phi.b(),
phi.state());
806   const ComplexChebyCoeff& phi_i = phi[i];
807
808   grad.u() = phi_i;
809   grad.u() *= (2*pi*phi.kx()/phi.Lx())*I;
810
811   diff(phi_i, grad.v());
812
813   grad.w() = phi_i;
814   grad.w() *= (2*pi*phi.kz()/phi.Lz())*I;
815   return grad;
816 }
```

Here is the call graph for this function:

1.5.3.24 Complex innerProduct (const BasisFunc & *f0*, const BasisFunc & *f1*, NormType *n*, bool *normalize* = true)

References chebyInnerProduct(), L2InnerProduct(), and L2NORM.

```
658                                 {
659   return (n==L2NORM) ? L2InnerProduct(f,g,nrmlz) : chebyInnerProduct(f,g,nrmlz);
660 }
```

Here is the call graph for this function:

1.5.3.25 Real L2Dist (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize* = true)

References L2Dist2().

Referenced by dist(), TauSolver::tauDist(), TauSolver::verify(), PressureSolver::verify(), PoissonSolver::verify(), and TauSolver::verify_P_and_v().

```
575                                                 {
576     return sqrt(L2Dist2(f,g,normalize));
577 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.5.3.26 Real L2Dist2 (const BasisFunc & *f*, const BasisFunc & *g*, bool *normalize* = true)

References BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), L2Dist2(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

Referenced by dist2(), L2Dist(), and L2Dist2().

```
563                                                 {
564   assert(f.geometryCongruent(g));
565   Real rtn=0.0;
566   if (f.kx()==g.kx() && f.kz()==g.kz())
```

```
567     for (int n=0; n<f.Nd(); ++n)
568       rtn += L2Dist2(f[n], g[n], normalize);
569   if (!normalize)
570     rtn *= f.Lx()*f.Lz();
571   return rtn;
572 
573 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.5.3.27 Complex L2InnerProduct (const BasisFunc & *f0*, const BasisFunc & *f1*, bool *normalize* = true)

References BasisFunc::geometryCongruent(), BasisFunc::kx(), BasisFunc::kz(), L2InnerProduct(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

Referenced by innerProduct(), and L2InnerProduct().

```
580                                 {
581   assert(f.geometryCongruent(g));
582   Complex rtn(0.0, 0.0);
583   if (f.kx()==g.kx() && f.kz()==g.kz())
584     for (int n=0; n<f.Nd(); ++n)
585       rtn += L2InnerProduct(f[n], g[n], normalize);
586   if (!normalize)
587     rtn *= f.Lx() * f.Lz();
588   return rtn;
589 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

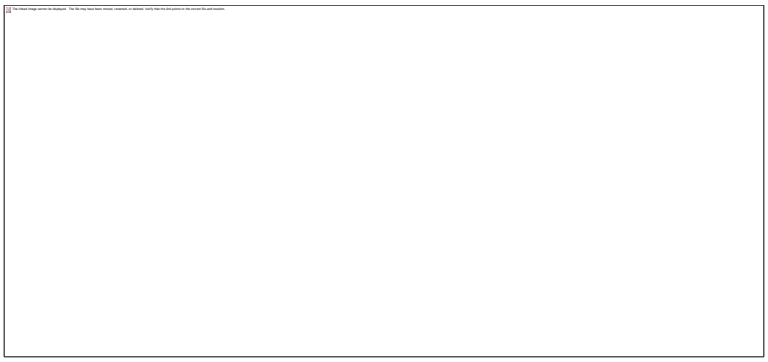
1.5.3.28 Real L2Norm (const BasisFunc & *f*, bool *normalize* = true)

References L2Norm2(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

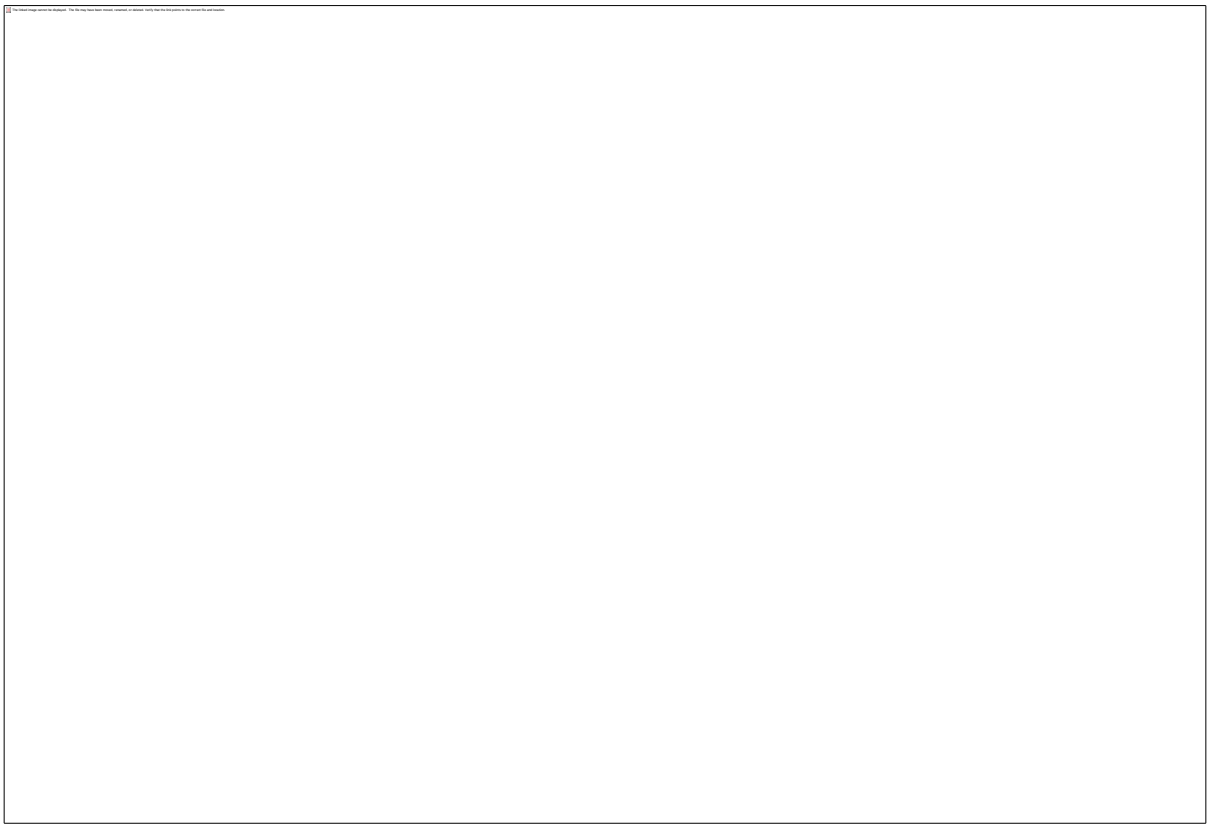
Referenced by chebyProfile(), makemeen(), makeprtb(), makeroll(), makewave(), makewobl(), norm(), randomProfile(), TauSolver::tauNorm(), TauSolver::verify(), PressureSolver::verify(), PoissonSolver::verify(), and TauSolver::verify_P_and_v().

```
545                                     {
546   Real rtn2 = 0.0;
547   for (int n=0; n<phi.Nd(); ++n)
548     rtn2 += L2Norm2(phi[n], normalize);
549   if (!normalize)
550     rtn2 *= phi.Lx()*phi.Lz();
551   return sqrt(rtn2);
552 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.5.3.29 Real L2Norm2 (const BasisFunc & *f*, bool *normalize* = true)

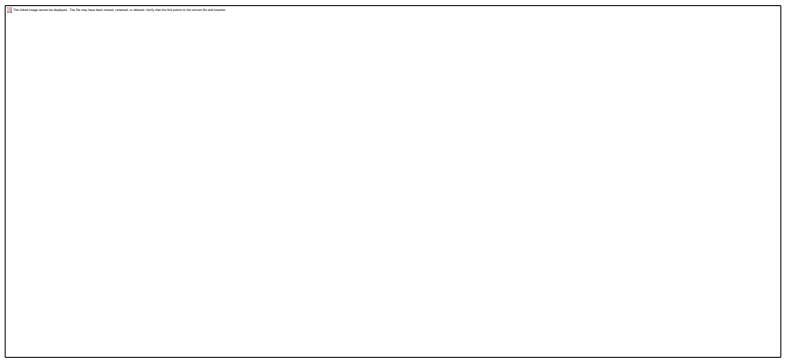
References L2Norm2(), BasisFunc::Lx(), BasisFunc::Lz(), and BasisFunc::Nd().

Referenced by divDist2(), divNorm2(), FlowField::energy(), L2Dist2(), L2Norm(), L2Norm2(), norm2(), and FlowField::saveSpectrum().

```
554                                     {
555   Real rtn = 0.0;
556   for (int n=0; n<phi.Nd(); ++n)
```

```
557     rtn += L2Norm2(phi[n], normalize);
558   if (!normalize)
559     rtn *= phi.Lx()*phi.Lz();
560   return rtn;
561 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.5.3.30 BasisFunc laplacian (const BasisFunc & *f*)

References diff2(), BasisFunc::kx(), BasisFunc::kz(), lapl(), BasisFunc::Lx(), BasisFunc::Lz(), pi, square(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

```
774                                 {
775   BasisFunc lapl(phi);
776   Real c = -4*square(pi)*(square(phi.kx()/phi.Lx())+square(phi.kz()/phi.Lz()));
777   lapl.u() *= c;
778   lapl.v() *= c;
779   lapl.w() *= c;
```

```
780
781   ComplexChebyCoeff tmp;
782   diff2(phi.u(), tmp);
783   lapl.u() += tmp;
784   diff2(phi.v(), tmp);
785   lapl.v() += tmp;
786   diff2(phi.w(), tmp);
787   lapl.w() += tmp;
788   return lapl;
789 }
```

Here is the call graph for this function:

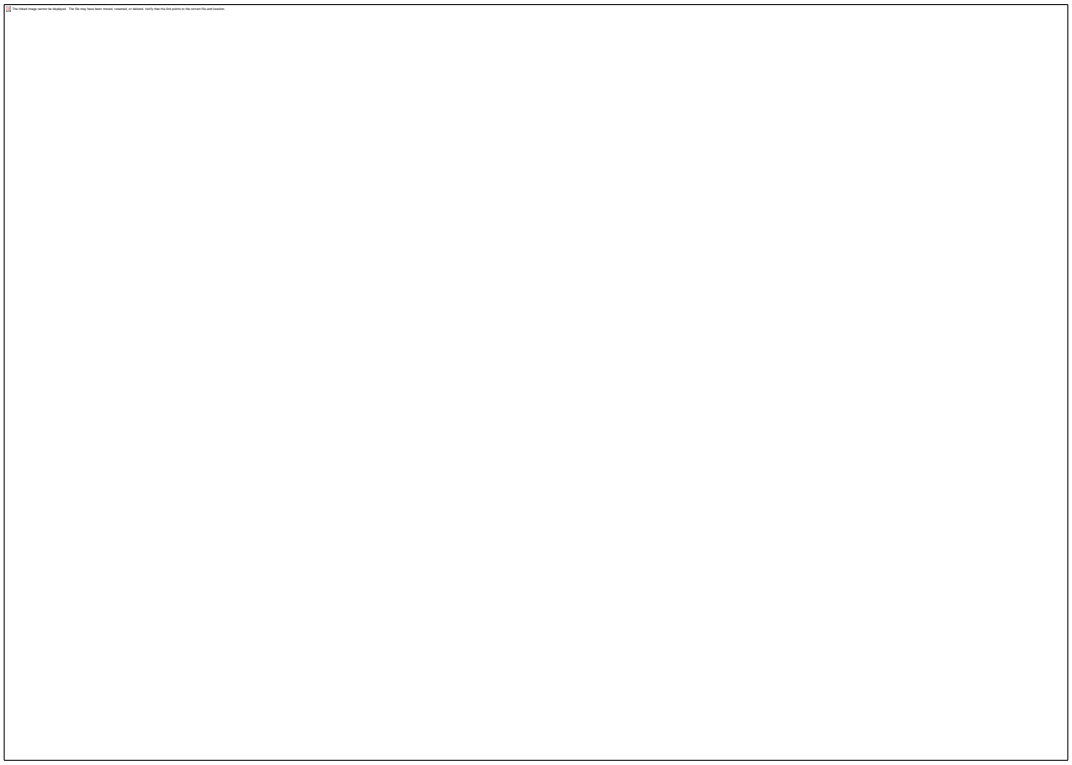
1.5.3.31 Real norm (const BasisFunc & *f*, NormType *n*, bool *normalize* = true)

References chebyNorm(), L2Norm(), and L2NORM.

Referenced by abs2(), FlowField::addPerturbations(), norm(), and HelmholtzSolver::verify().

```
641                                                 {
642   return (n==L2NORM) ? L2Norm(phi,normalize) : chebyNorm(phi,normalize);
643 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

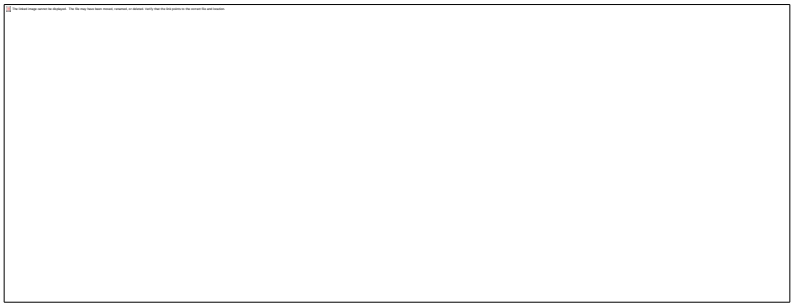
1.5.3.32 Real norm2 (const BasisFunc & *f*, NormType *n*, bool *normalize* = true)

References chebyNorm2(), L2NORM, and L2Norm2().

Referenced by norm2().

```
645                                                 {
646   return (n==L2NORM) ? L2Norm2(phi,normalize) : chebyNorm2(phi,normalize);
647 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.5.3.33 bool operator!= (const BasisFunc & *f*, const BasisFunc & *g*)

```
527                                         {
```

```
528   return (f == g) ? false : true;
529 }
```

1.5.3.34 bool operator== (const BasisFunc & *f*, const BasisFunc & *g*)

References BasisFunc::congruent(), and BasisFunc::Nd().

```
531                                                 {
532   if (!f.congruent(g))
533     return false;
534
535   for (int n=0; n<f.Nd(); ++n)
536     if (f[n] != g[n])
537       return false;
538
539   return true;
540 }
```

Here is the call graph for this function:

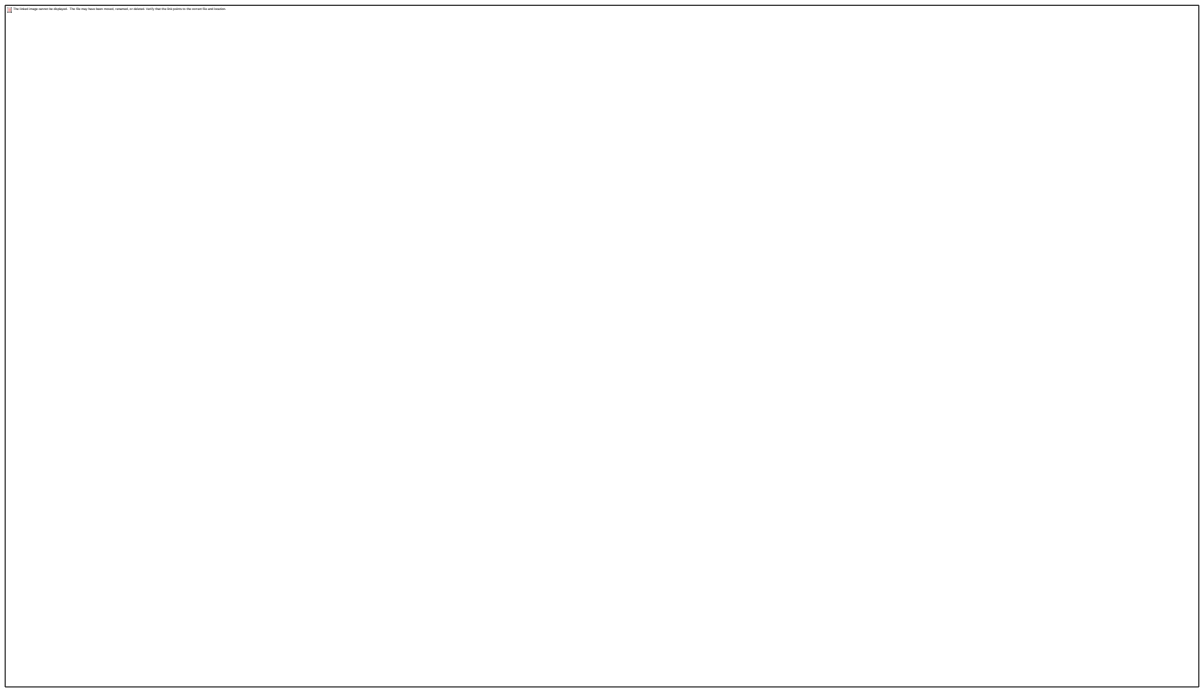
1.5.3.35 void ubcFix (ComplexChebyCoeff & *u*, bool *a_dirichlet*, bool *b_dirichlet*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), ComplexChebyCoeff::setBounds(), and ComplexChebyCoeff::sub().

```
1018                                                 {
1019   Real ua = u.a();
1020   Real ub = u.b();
1021   u.setBounds(-1,1);
1022
1023   if (a_dirichlet && b_dirichlet) {
1024     u.sub(0, 0.5*(u.eval_b() + u.eval_a()));
1025     u.sub(1, 0.5*(u.eval_b() - u.eval_a()));
1026   }
1027   else if (a_dirichlet)
1028     u.sub(0, u.eval_a());
1029   else if (a_dirichlet)
1030     u.sub(0, u.eval_b());
1031
```

```
1032   u.setBounds(ua, ub);
1033 }
```

Here is the call graph for this function:



1.5.3.36 void ubcFix (ChebyCoeff & *u*, bool *a_dirichlet*, bool *b_dirichlet*)

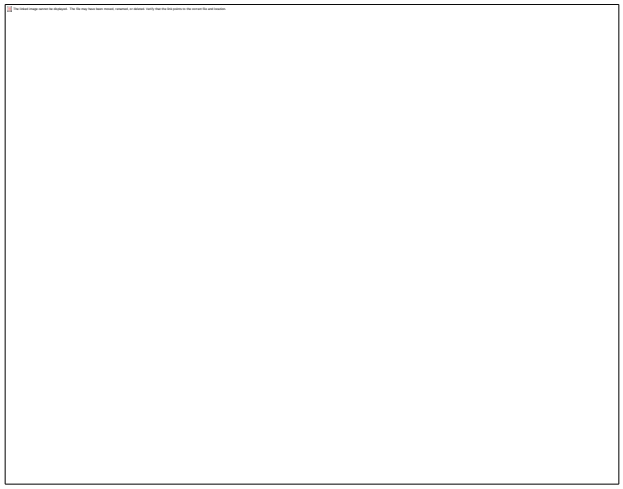
References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), and ChebyCoeff::setBounds().

Referenced by BasisFunc::randomize().

```
1001                                                {
1002   Real ua = u.a();
1003   Real ub = u.b();
1004   u.setBounds(-1,1);
1005
1006   if (a_dirichlet && b_dirichlet) {
1007     u[0] -= 0.5*(u.eval_b() + u.eval_a());
1008     u[1] -= 0.5*(u.eval_b() - u.eval_a());
1009   }
1010   else if (a_dirichlet)
1011     u[0] -= u.eval_a();
1012   else if (a_dirichlet)
1013     u[0] -= u.eval_b();
1014
1015   u.setBounds(ua, ub);
```

```
1016 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

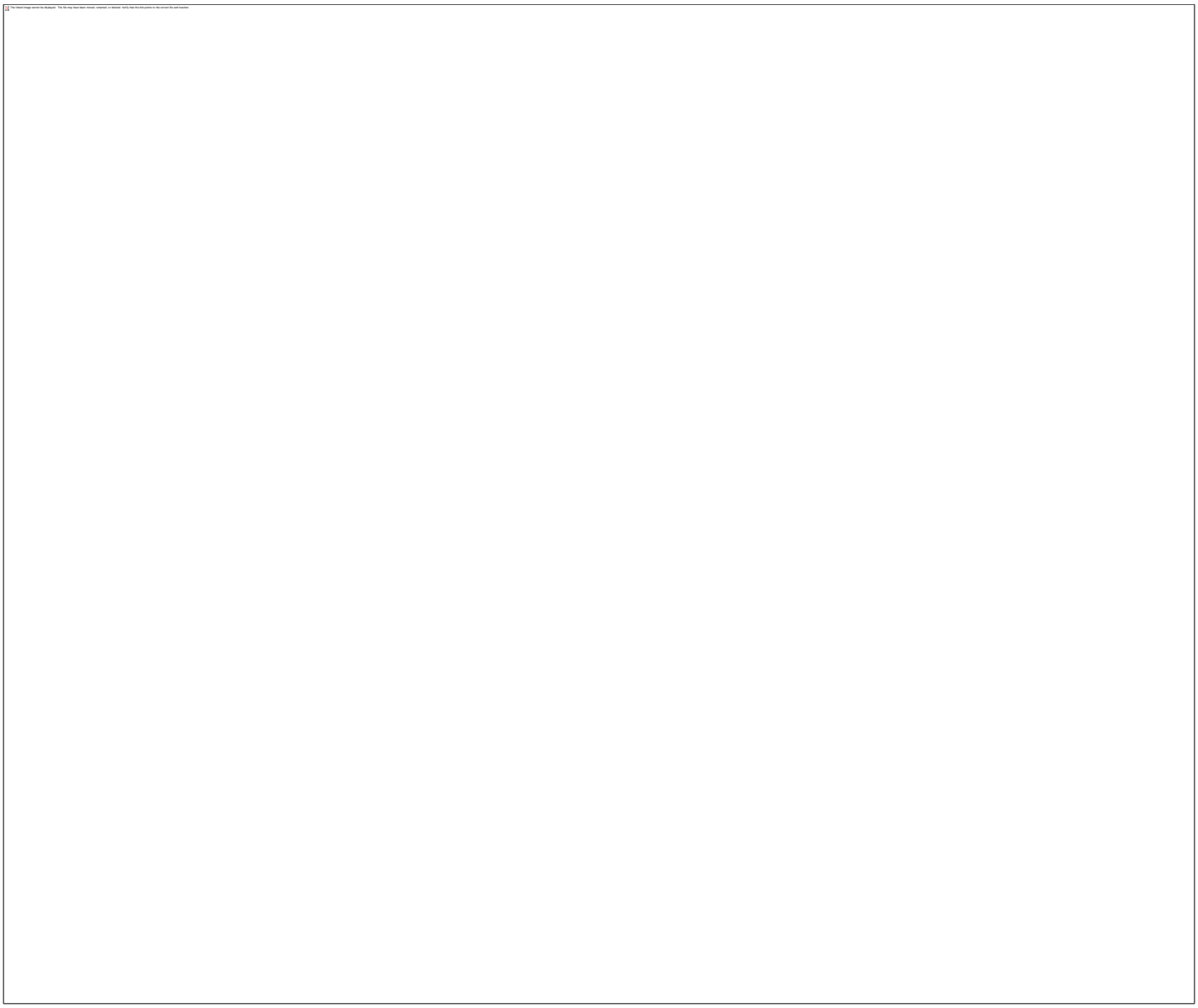
1.5.3.37 void vbcFix (ComplexChebyCoeff & *v*, bool *a_dirichlet*, bool *b_dirichlet*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), ComplexChebyCoeff::setBounds(), and ComplexChebyCoeff::sub().

```
1067                                                {
1068
1069   Real va = v.a();
1070   Real vb = v.b();
1071   v.setBounds(-1,1);
1072
1073   if (a_dirichlet && b_dirichlet) {
1074     // Adjust v to match v(+-1)=v'(+-1)=0 BCs
1075     ComplexChebyCoeff vy = diff(v);
1076     Complex a = v.eval_a();
1077     Complex b = v.eval_b();
1078     Complex c = vy.eval_a();
1079     Complex d = vy.eval_b();
1080
1081     v.sub(0, 0.5*(a + b) + 0.125*(c - d));
1082     v.sub(1, 0.5625*(b - a) - 0.0625*(c + d));
```

```
1083     v.sub(2, 0.125*(d - c));
1084     v.sub(3, 0.0625*(a - b + c + d));
1085   }
1086   else if (a_dirichlet) {
1087     ComplexChebyCoeff vy = diff(v);
1088     v.sub(1, vy.eval_a());
1089     v.sub(0,  v.eval_a());
1090   }
1091   else if (b_dirichlet) {
1092     ComplexChebyCoeff vy = diff(v);
1093     v.sub(1, vy.eval_b());
1094     v.sub(0,  v.eval_b());
1095   }
1096   v.setBounds(va,vb);
1097 }
```

Here is the call graph for this function:



1.5.3.38 void vbcFix (ChebyCoeff & *v*, bool *a_dirichlet*, bool *b_dirichlet*)

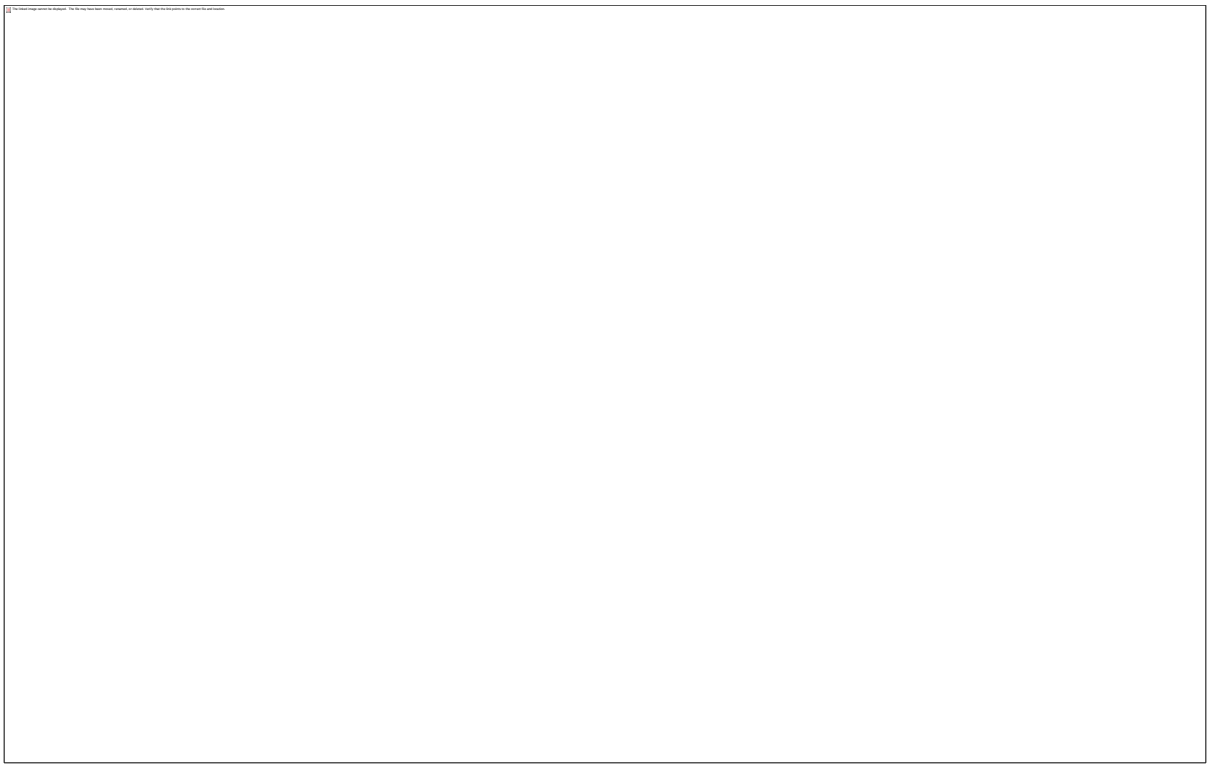
References ChebyCoeff::a(), ChebyCoeff::b(), diff(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), and ChebyCoeff::setBounds().

Referenced by BasisFunc::randomize().

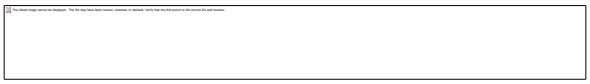
```
1035                                                {
1036
1037   Real va = v.a();
1038   Real vb = v.b();
1039   v.setBounds(-1,1);
1040
```

```
1041   if (a_dirichlet && b_dirichlet) {
1042     // Adjust v to match v(+-1)=v'(+-1)=0 BCs
1043     ChebyCoeff vy = diff(v);
1044     Real a = v.eval_a();
1045     Real b = v.eval_b();
1046     Real c = vy.eval_a();
1047     Real d = vy.eval_b();
1048
1049     v[0] -= 0.5*(a + b) + 0.125*(c - d);
1050     v[1] -= 0.5625*(b - a) - 0.0625*(c + d);
1051     v[2] -= 0.125*(d - c);
1052     v[3] -= 0.0625*(a - b + c + d);
1053   }
1054   else if (a_dirichlet) {
1055     ChebyCoeff vy = diff(v);
1056     v[1] -= vy.eval_a();
1057     v[0] -= v.eval_a();
1058   }
1059   else if (b_dirichlet) {
1060     ChebyCoeff vy = diff(v);
1061     v[1] -= vy.eval_b();
1062     v[0] -= v.eval_b();
1063   }
1064   v.setBounds(va,vb);
1065 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



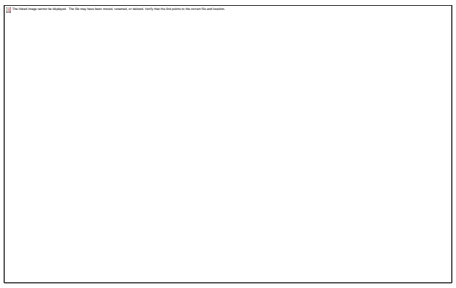
1.5.3.39 BasisFunc xdiff (const BasisFunc & *f*)

References I(), BasisFunc::kx(), BasisFunc::Lx(), and pi.

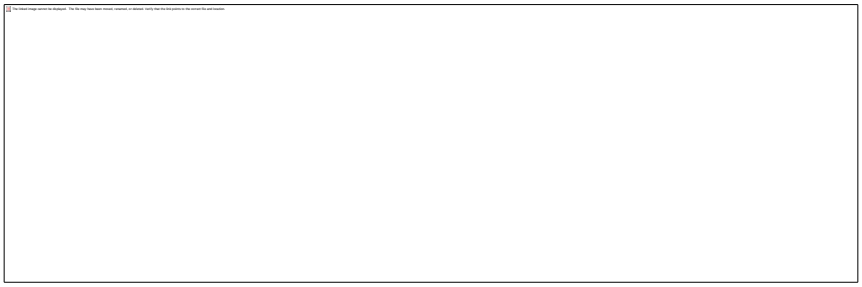
Referenced by polyInterp(), and xdiff().

```
733                                 {
734   BasisFunc phix(phi);
735   phix *= (2*pi*phi.kx()/phi.Lx())*I;
736   return phix;
737 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.5.3.40 BasisFunc ydiff (const BasisFunc & *f*)

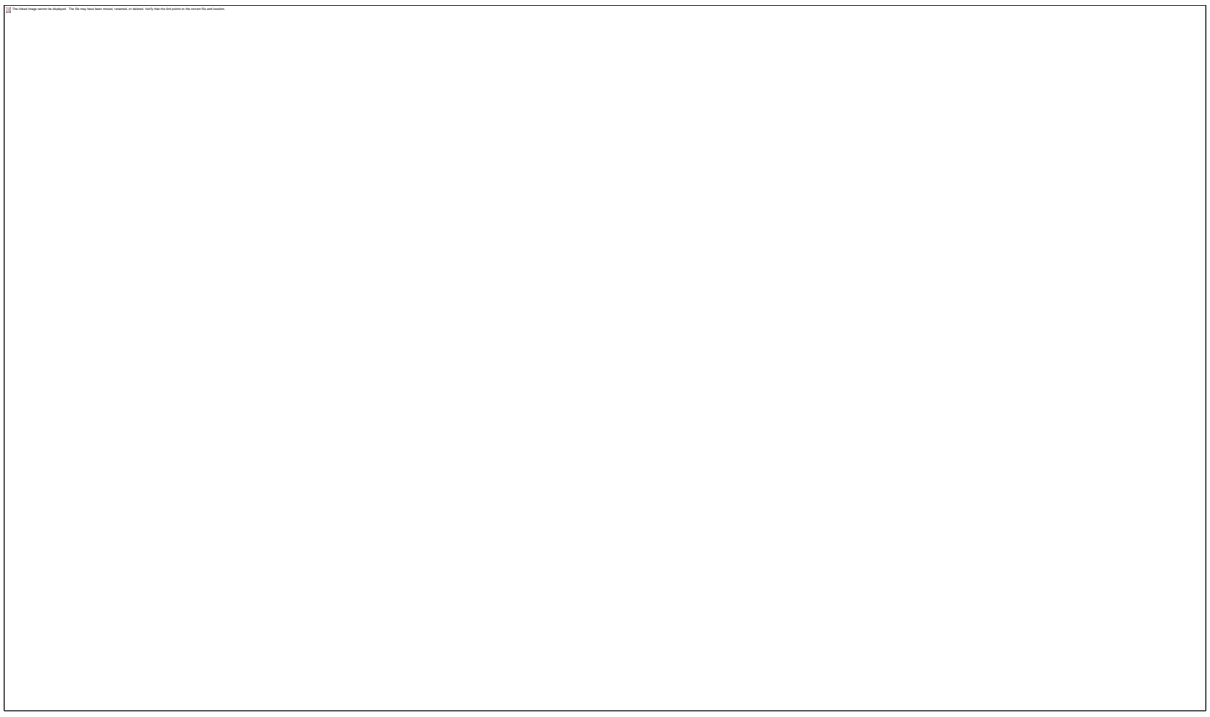
References BasisFunc::a(), BasisFunc::b(), diff(), BasisFunc::kx(), BasisFunc::kz(), BasisFunc::Lx(), BasisFunc::Lz(), BasisFunc::Ny(), Spectral, BasisFunc::state(), BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

Referenced by curl(), PressureSolver::verify(), and ydiff().

```
740                                 {
741   assert(phi.state() == Spectral);
742   BasisFunc phiy(phi.Ny(), phi.kx(), phi.kz(), phi.Lx(), phi.Lz(), phi.a(),
phi.b(), Spectral);
743   diff(phi.u(), phiy.u());
744   diff(phi.v(), phiy.v());
745   diff(phi.w(), phiy.w());
746   return phiy;
747 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

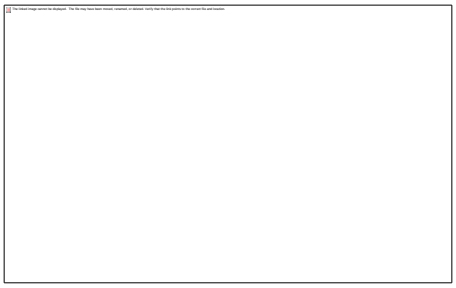


1.5.3.41 BasisFunc zdiff (const BasisFunc & *f*)

References I(), BasisFunc::kz(), BasisFunc::Lz(), and pi.
Referenced by zdiff().

```
749                                 {
750   BasisFunc phiz(phi);
751   phiz *= (2*pi*phi.kz()/phi.Lz())*I;
752   return phiz;
753 }
```

Here is the call graph for this function:

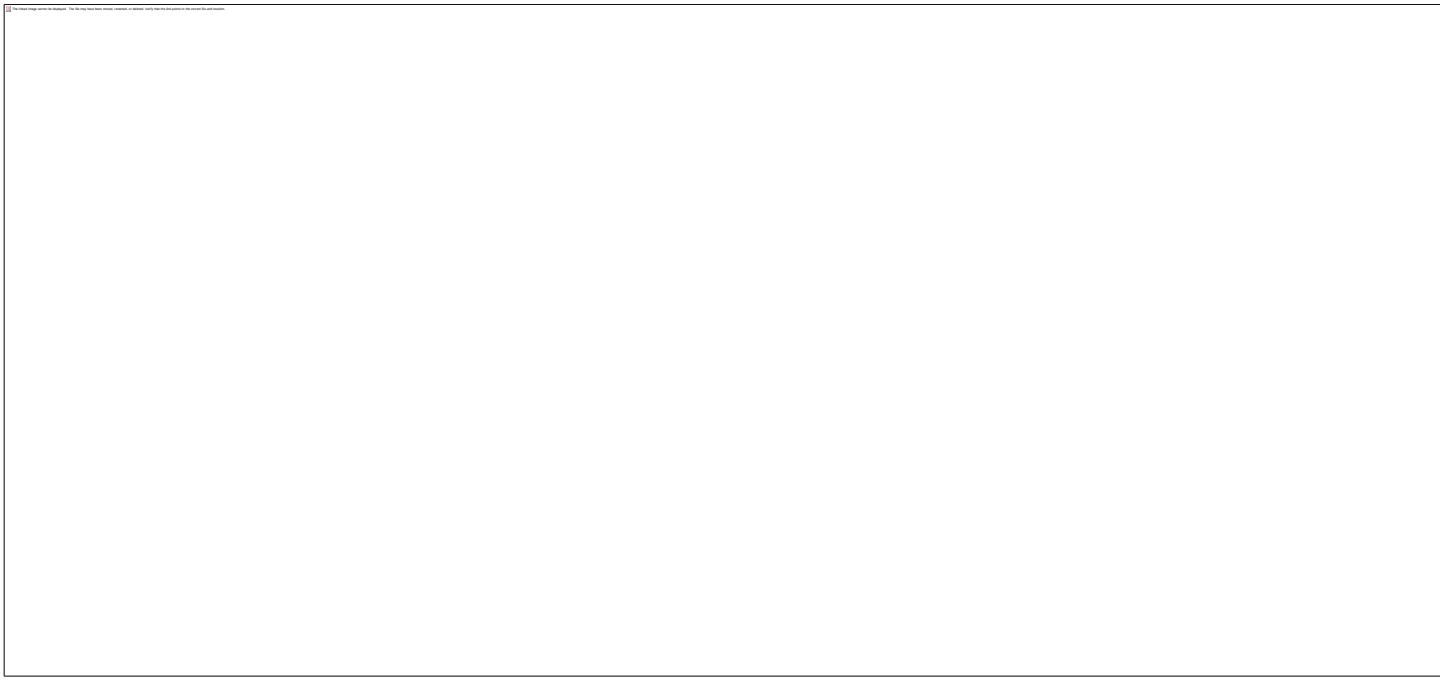


Here is the caller graph for this function:

1.5.3.42

1.6 /_cuda/channelflow-0.9.22/channelflow/chebyshev.cpp File Reference

#include <fstream>
#include <iomanip>
#include "channelflow/chebyshev.h"
#include "config.h"
Include dependency graph for chebyshev.cpp:



1.6.1 Functions

- Real chebyIP (int m, int n)
- Real legendre (int n, Real x)
- Real chebyshev (int n, Real x)
- ChebyCoeff legendre (int N, int n, bool normalize)
- ChebyCoeff chebyshev (int N, int n, bool normalize)
- void gaussLegendreQuadrature (int N, Real a, Real b, Vector &x, Vector &w)
- void fftw_loadwisdom (const char *filename_)
- void fftw_savewisdom (const char *filename_)
- ChebyCoeff operator* (Real c, const ChebyCoeff &v)
- ChebyCoeff operator+ (const ChebyCoeff &u, const ChebyCoeff &v)
- ChebyCoeff operator- (const ChebyCoeff &u, const ChebyCoeff &v)
- bool operator== (const ChebyCoeff &u, const ChebyCoeff &v)
- bool operator!= (const ChebyCoeff &u, const ChebyCoeff &v)
- void swap (ChebyCoeff &f, ChebyCoeff &g)
- void integrate (const ChebyCoeff &dudy, ChebyCoeff &u)
- ChebyCoeff integrate (const ChebyCoeff &dudy)

- void diff (const ChebyCoeff &u, ChebyCoeff &dudy)
- void diff2 (const ChebyCoeff &u, ChebyCoeff &d2udy2)
- void diff2 (const ChebyCoeff &u, ChebyCoeff &d2udy2, ChebyCoeff &tmp)
- void diff (const ChebyCoeff &f, ChebyCoeff &df, int n)
- ChebyCoeff diff (const ChebyCoeff &u)
- ChebyCoeff diff2 (const ChebyCoeff &u)
- ChebyCoeff diff (const ChebyCoeff &f, int n)
- Vector chebypoints (int N, Real a, Real b)
- Vector periodicpoints (int N, Real L)
- Real L2Norm2 (const ChebyCoeff &u, bool normalize)
- Real L2Dist2 (const ChebyCoeff &u, const ChebyCoeff &v, bool normalize)
- Real L2Norm (const ChebyCoeff &u, bool normalize)
- Real L2Dist (const ChebyCoeff &u, const ChebyCoeff &v, bool normalize)
- Real L2InnerProduct (const ChebyCoeff &u, const ChebyCoeff &v, bool normalize)
- Real chebyNorm2 (const ChebyCoeff &u, bool normalize)
- Real chebyDist2 (const ChebyCoeff &u, const ChebyCoeff &v, bool normalize)
- Real chebyNorm (const ChebyCoeff &u, bool normalize)
- Real chebyDist (const ChebyCoeff &u, const ChebyCoeff &v, bool normalize)
- Real chebyInnerProduct (const ChebyCoeff &u, const ChebyCoeff &v, bool normalize)
- Real LinfNorm (const ChebyCoeff &f)
- Real LinfDist (const ChebyCoeff &f, const ChebyCoeff &g)
- Real L1Norm (const ChebyCoeff &f, bool normalize)
- Real L1Dist (const ChebyCoeff &f, const ChebyCoeff &g, bool normalize)
- Real norm2 (const ChebyCoeff &u, NormType n, bool normalize)
- Real dist2 (const ChebyCoeff &u, const ChebyCoeff &v, NormType n, bool normalize)
- Real norm (const ChebyCoeff &u, NormType n, bool normalize)
- Real dist (const ChebyCoeff &u, const ChebyCoeff &v, NormType n, bool normalize)
- Real innerProduct (const ChebyCoeff &u, const ChebyCoeff &v, NormType n, bool normalize)
- bool operator== (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v)
- bool operator!= (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v)
- void swap (ComplexChebyCoeff &f, ComplexChebyCoeff &g)
- ComplexChebyCoeff operator* (Real c, const ComplexChebyCoeff &v)
- ComplexChebyCoeff operator+ (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v)
- ComplexChebyCoeff operator- (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v)
- void diff (const ComplexChebyCoeff &u, ComplexChebyCoeff &du)
- void diff2 (const ComplexChebyCoeff &u, ComplexChebyCoeff &d2u)
- void diff2 (const ComplexChebyCoeff &u, ComplexChebyCoeff &d2u, ComplexChebyCoeff &tmp)
- void diff (const ComplexChebyCoeff &u, ComplexChebyCoeff &du, int n)
- ComplexChebyCoeff diff (const ComplexChebyCoeff &u)
- ComplexChebyCoeff diff2 (const ComplexChebyCoeff &u)

- ComplexChebyCoeff diff (const ComplexChebyCoeff &u, int n)
- void integrate (const ComplexChebyCoeff &du, ComplexChebyCoeff &u)
- ComplexChebyCoeff integrate (const ComplexChebyCoeff &du)
- ostream & operator<< (ostream &os, const ComplexChebyCoeff &u)
- Real L2Norm2 (const ComplexChebyCoeff &u, bool normalize)
- Real L2Dist2 (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, bool normalize)
- Real L2Norm (const ComplexChebyCoeff &u, bool normalize)
- Real L2Dist (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, bool normalize)
- Complex L2InnerProduct (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, bool normalize)
- Real L1Norm (const ComplexChebyCoeff &u, bool normalize)
- Real L1Dist (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, bool normalize)
- Real LinfNorm (const ComplexChebyCoeff &u)
- Real LinfDist (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v)
- Real chebyNorm2 (const ComplexChebyCoeff &u, bool normalize)
- Real chebyDist2 (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, bool normalize)
- Real chebyNorm (const ComplexChebyCoeff &u, bool normalize)
- Real chebyDist (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, bool normalize)
- Complex chebyInnerProduct (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, bool normalize)
- Real norm2 (const ComplexChebyCoeff &u, NormType n, bool normalize)
- Real dist2 (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, NormType n, bool normalize)
- Real norm (const ComplexChebyCoeff &u, NormType n, bool normalize)
- Real dist (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, NormType n, bool normalize)
- Complex innerProduct (const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, NormType n, bool normalize)

1.6.2 Variables

- const Real QUADRATURE_EPSILON = 1e-17

1.6.3 Function Documentation

1.6.3.1 Real chebyDist (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, bool *normalize*)

References chebyDist2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1565                         {
1566   return sqrt(chebyDist2(u.re, v.re, normalize) + chebyDist2(u.im, v.im, normalize));
1567 }
```

Here is the call graph for this function:

1.6.3.2 Real chebyDist (const ChebyCoeff & *u*, const ChebyCoeff & *v*, bool *normalize*)

References chebyDist2().

```
858                                                 {
859   return sqrt(chebyDist2(u,v, normalize));
860 }
```

Here is the call graph for this function:

1.6.3.3 Real chebyDist2 (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, bool *normalize*)

References chebyDist2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1556                         {
1557   return chebyDist2(u.re, v.re, normalize) + chebyDist2(u.im, v.im, normalize);
1558 }
```

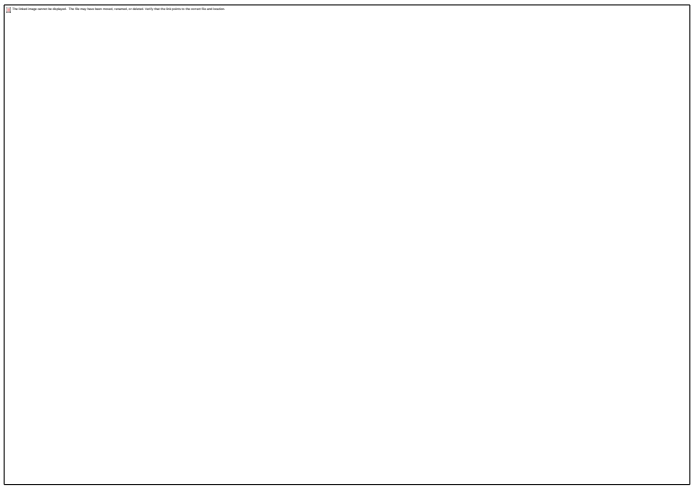
Here is the call graph for this function:

1.6.3.4 Real chebyDist2 (const ChebyCoeff & *u*, const ChebyCoeff & *v*, bool *normalize*)

References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::numModes(), pi, Spectral, square(), and ChebyCoeff::state().

```
839                                                 {
840   assert(u.state()==Spectral && v.state()==Spectral);
841   assert(u.a()==v.a() && u.b()==v.b());
842   assert(u.numModes() ==  v.numModes());
843   int N=u.numModes();
844   Real sum=0.0;
845   for (int m=N-1; m>0; --m)
846     sum += square(u[m]-v[m]);
847   if (N>0)
848     sum += 2*square(u[0]-v[0]);  // coeff of T_0(x) has prefactor of 2 in norm
849   if (!normalize)
850     sum *= u.b()-u.a();
851   return sum*pi/2;
852 }
```

Here is the call graph for this function:



1.6.3.5 Complex chebyInnerProduct (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, bool *normalize*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), ComplexChebyCoeff::im, ComplexChebyCoeff::numModes(), pi, ComplexChebyCoeff::re, Spectral, and ComplexChebyCoeff::state().

```
1571                                        {
1572   assert(u.state()==Spectral && v.state()==Spectral);
1573   assert(u.numModes() == v.numModes());
```

```
1574   int N = u.numModes();
1575
1576   // Unroll the complex operations, for efficiency.
1577   // Profiling an L2InnerProduct-intensive program reveals that
1578   // complex * double is expensive
1579   Real sum_r = 0.0;
1580   Real sum_i = 0.0;
1581   for (int m=N-1; m>0; --m) {
1582     sum_r += u.re[m]*v.re[m] + u.im[m]*v.im[m];
1583     sum_i += u.im[m]*v.re[m] - u.re[m]*v.im[m];
1584   }
1585   if (N>0) {
1586     sum_r += 2*(u.re[0]*v.re[0] + u.im[0]*v.im[0]);
1587     sum_i += 2*(u.im[0]*v.re[0] - u.re[0]*v.im[0]);
1588   }
1589   if (!normalize) {
1590     sum_r *= u.b()-u.a();
1591     sum_i *= u.b()-u.a();
1592   }
1593   return Complex(sum_r*pi/2, sum_i*pi/2);
1594 }
```

Here is the call graph for this function:

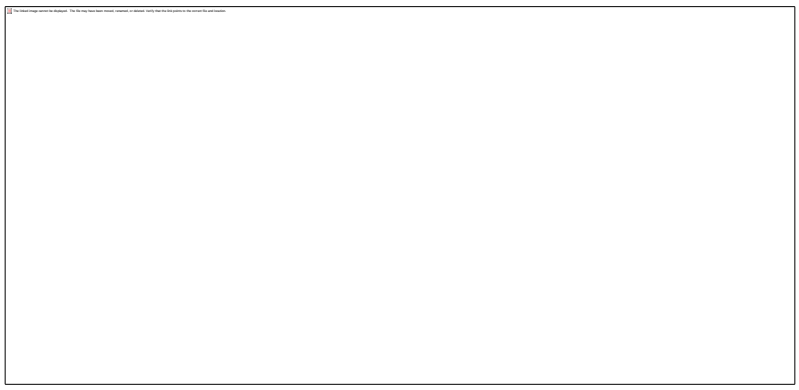
1.6.3.6 Real chebyInnerProduct (const ChebyCoeff & *u*, const ChebyCoeff & *v*, bool *normalize*)

References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::numModes(), pi, Spectral, and ChebyCoeff::state().

```
862                                                    {
863   assert(u.state()==Spectral && v.state()==Spectral);
864   assert(u.a()==v.a() && u.b()==v.b());
865   assert(u.numModes() ==  v.numModes());
```

```
866   int N=u.numModes();
867   Real sum=0.0;
868   for (int m=N-1; m>0; --m)
869     sum += u[m]*v[m];
870   if (N>0)
871     sum += 2*u[0]*v[0];  // coeff of T_0(x) has prefactor of 2 in norm
872   if (!normalize)
873     sum *= u.b()-u.a();
874   return sum*pi/2;
875 }
```

Here is the call graph for this function:



1.6.3.7 Real chebyIP (int *m*, int *n*)

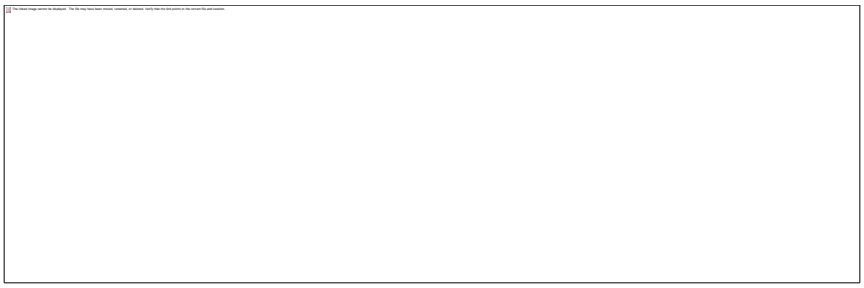
```
46                          {
47   return ((m+n) % 2 == 1) ? 0 :
48     (1.0-m*m-n*n)/((1.0+m-n)*(1.0-m+n)*(1.0+m+n)*(1.0-m-n));
49 }
```

1.6.3.8 Real chebyNorm (const ComplexChebyCoeff & *u*, bool *normalize*)

References chebyNorm2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1560                                                {
1561   return sqrt(chebyNorm2(u.re, normalize) + chebyNorm2(u.im, normalize));
1562 }
```

Here is the call graph for this function:

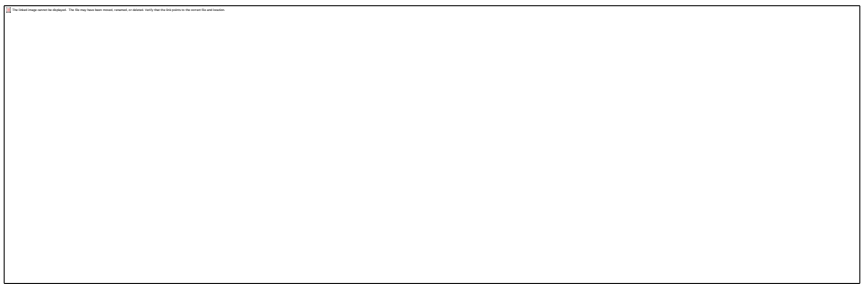


1.6.3.9 Real chebyNorm (const ChebyCoeff & *u*, bool *normalize*)

References chebyNorm2().

```
854                                                 {
855   return sqrt(chebyNorm2(u, normalize));
856 }
```

Here is the call graph for this function:



1.6.3.10 Real chebyNorm2 (const ComplexChebyCoeff & *u*, bool *normalize*)

References chebyNorm2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1551                                                    {
1552   return chebyNorm2(u.re, normalize) + chebyNorm2(u.im, normalize);
1553 }
```

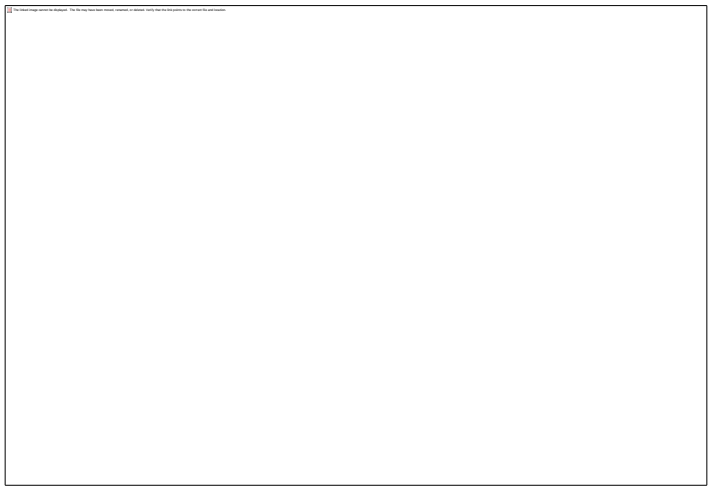
Here is the call graph for this function:

1.6.3.11 Real chebyNorm2 (const ChebyCoeff & *u*, bool *normalize*)

References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::numModes(), pi, Spectral, square(), and ChebyCoeff::state().

```
825                                                 {
826   assert(u.state() == Spectral);
827   int N=u.numModes();
828   Real sum=0.0;
829   for (int m=N-1; m>0; --m)
830     sum += square(u[m]);
831   if (N>0)
832     sum += 2*square(u[0]);  // coeff of T_0(x) has prefactor of 2 in norm
833   if (!normalize)
834     sum *= u.b()-u.a();
835   sum *= pi/2;
836   return sum;
837 }
```

Here is the call graph for this function:



1.6.3.12 Vector chebypoints (int *N*, Real *a*, Real *b*)

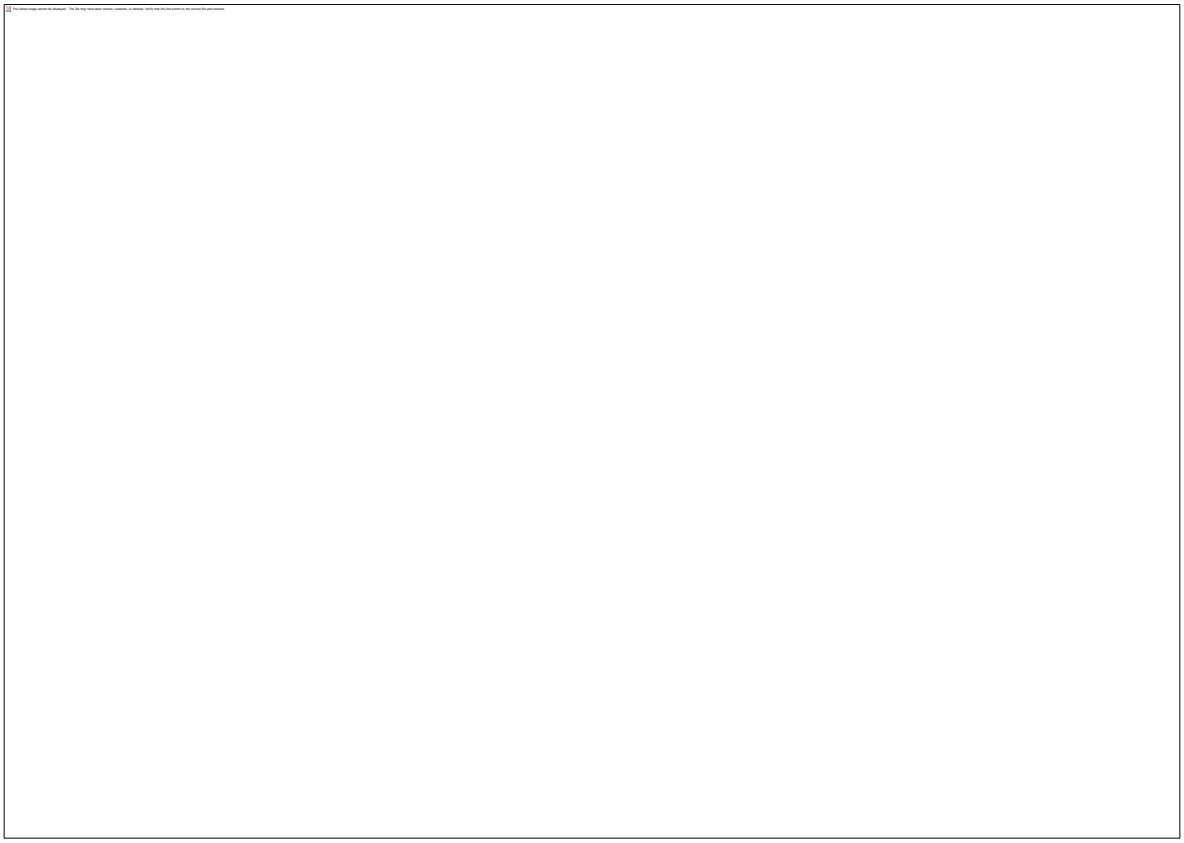
References pi.

Referenced by FlowField::CFLfactor(), makemeen(), makeprtb(), makeroll(), makewave(), makewobl(), and TurbStats::yplus().

```
754                                         {
755   Vector xcheb(N);
756   Real piN = pi/(N-1);
```

```
757   Real radius = (b-a)/2;
758   Real center = (b+a)/2;
759   for (int j=0; j<N; ++j)
760     xcheb[j] = center + radius*cos(j*piN);
761   return xcheb;
762 }
```

Here is the caller graph for this function:



1.6.3.13 ChebyCoeff chebyshev (int *N*, int *n*, bool *normalize*)

References cheby_c(), pi, and Spectral.

```
79                                        {
80   assert(n<N);
81   ChebyCoeff Tn(N, -1 , 1, Spectral);
82   if (!normalize)
83     Tn[n] = 1;
84   else
85     Tn[n] = sqrt(2/(pi*cheby_c(n)));
86   return Tn;
87 }
```

Here is the call graph for this function:

1.6.3.14 Real chebyshev (int *n*, Real *x*)

```
62                          {
63   return cos(n*acos(x));
64 }
```

1.6.3.15 ComplexChebyCoeff diff (const ComplexChebyCoeff & *u*, int *n*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff(), ComplexChebyCoeff::im, ComplexChebyCoeff::numModes(), ComplexChebyCoeff::re, and Spectral.

```
1286                                        {
1287   ComplexChebyCoeff du(u.numModes(), u.a(), u.b(), Spectral);
1288   diff(u.re, du.re, n);
1289   diff(u.im, du.im, n);
1290   return du;
1291 }
```

Here is the call graph for this function:

1.6.3.16 ComplexChebyCoeff diff (const ComplexChebyCoeff & *u*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff(), ComplexChebyCoeff::im, ComplexChebyCoeff::numModes(), ComplexChebyCoeff::re, and Spectral.

```
1274                                        {
1275   ComplexChebyCoeff du(u.numModes(), u.a(), u.b(), Spectral);
1276   diff(u.re, du.re);
1277   diff(u.im, du.im);
1278   return du;
1279 }
```

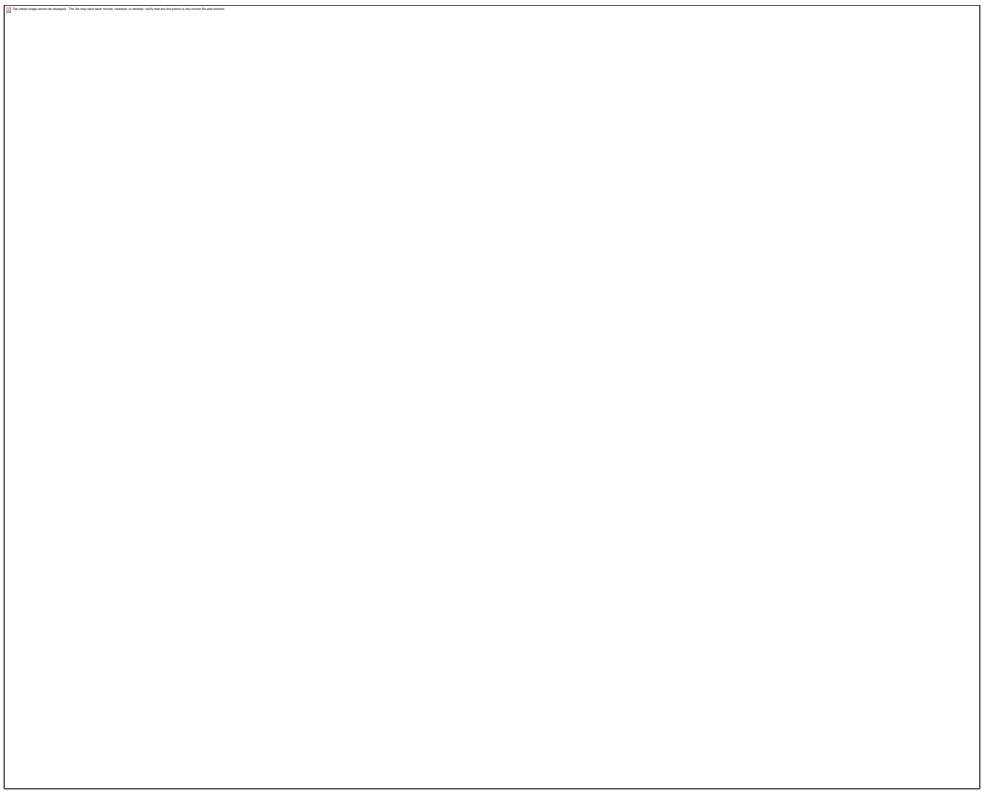
Here is the call graph for this function:

1.6.3.17 void diff (const ComplexChebyCoeff & *u*, ComplexChebyCoeff & *du*, int *n*)

References diff(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1270                                                    {
1271   diff(u.re, du.re, n);
1272   diff(u.im, du.im, n);
1273 }
```

Here is the call graph for this function:

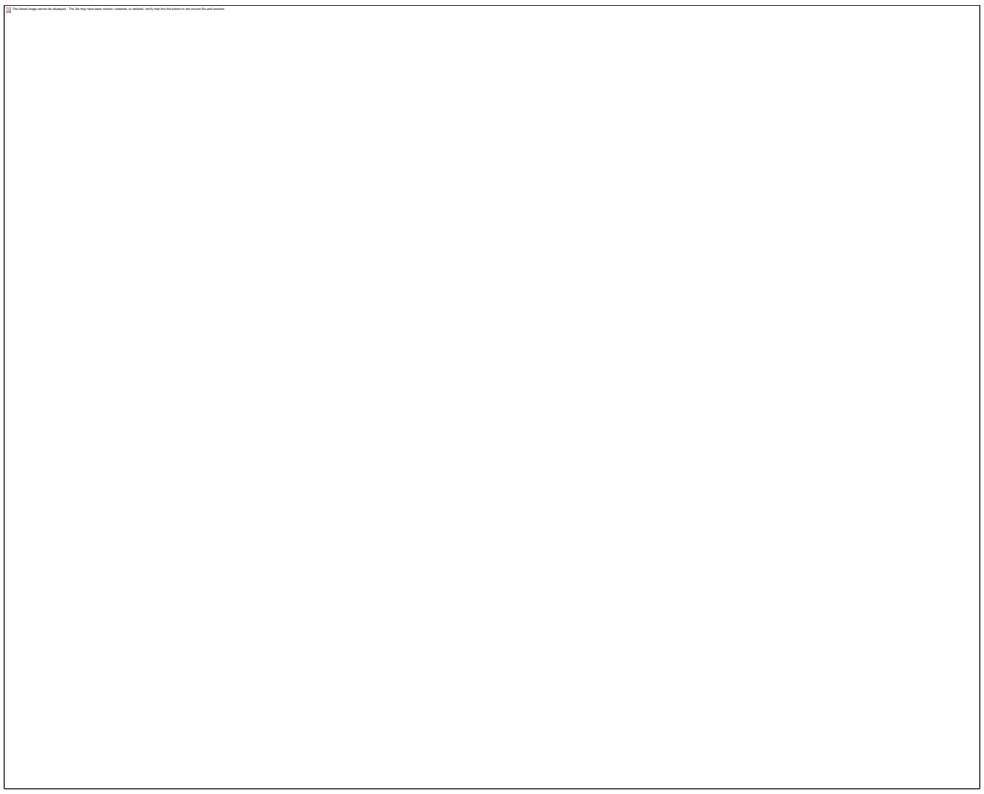


1.6.3.18 void diff (const ComplexChebyCoeff & *u*, ComplexChebyCoeff & *du*)

References diff(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1256                                                {
1257   diff(u.re, du.re);
1258   diff(u.im, du.im);
1259 }
```

Here is the call graph for this function:

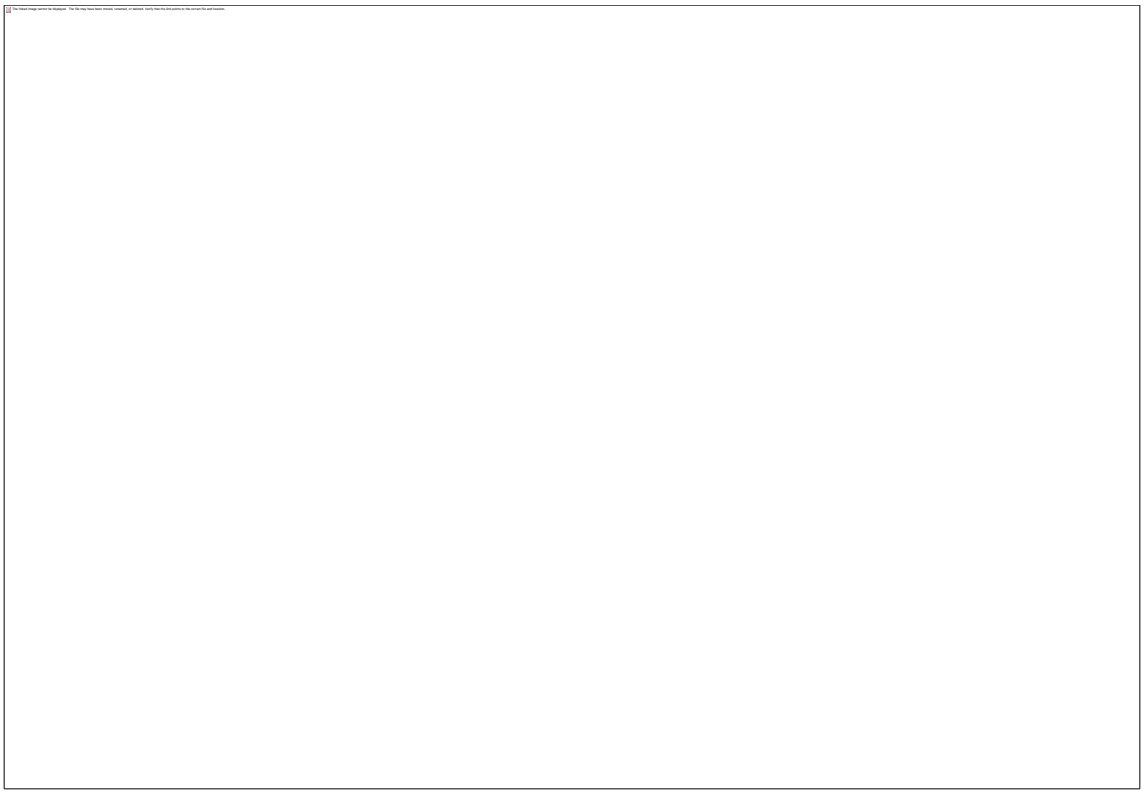


1.6.3.19 ChebyCoeff diff (const ChebyCoeff & *f*, int *n*)

References ChebyCoeff::a(), ChebyCoeff::b(), diff(), ChebyCoeff::N(), Spectral, and swap().

```
742                                     {
743   assert(n>=0);
744   ChebyCoeff  g = f;
745   ChebyCoeff gy(f.N(), f.a(), f.b(), Spectral);
746
747   for (int n_=0; n_<n; ++n_) {
748     diff(g,gy);
749     swap(g,gy);
750   }
751   return g;
752 }
```

Here is the call graph for this function:

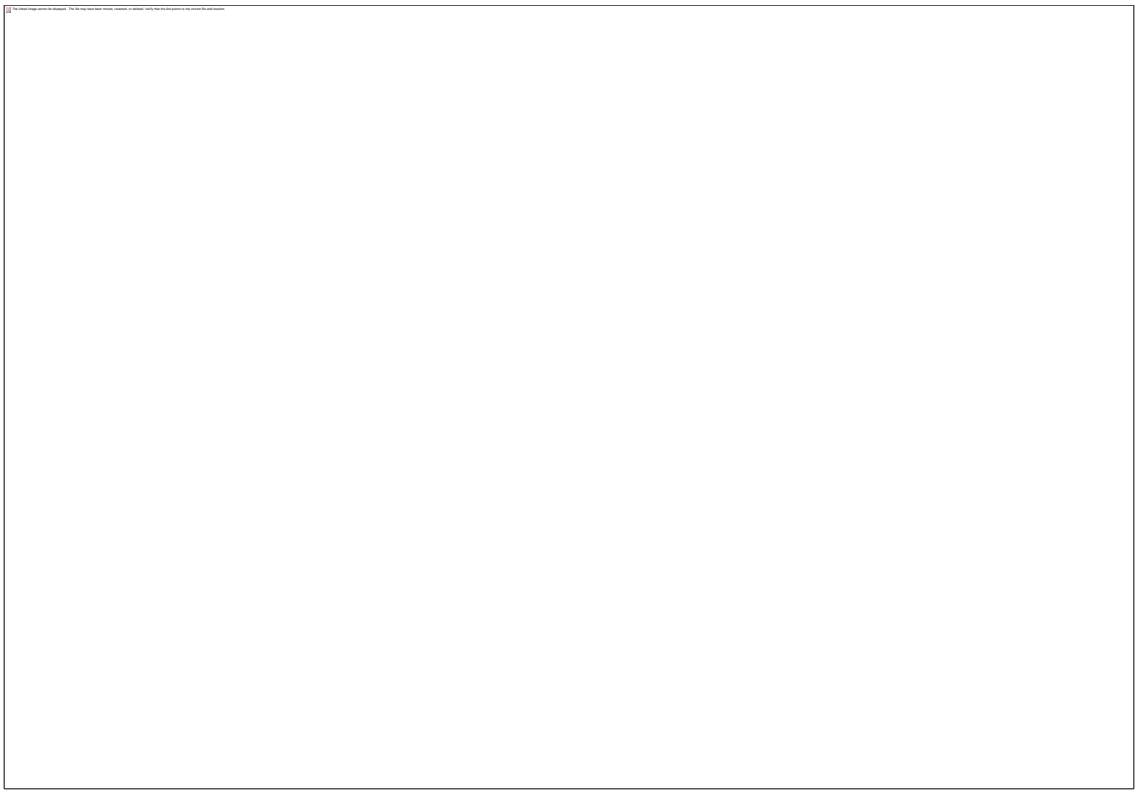


1.6.3.20 ChebyCoeff diff (const ChebyCoeff & *u*)

References ChebyCoeff::a(), ChebyCoeff::b(), diff(), Vector::length(), and Spectral.

```
730                                 {
731   ChebyCoeff du(u.length(), u.a(), u.b(), Spectral);
732   diff(u, du);
733   return du;
734 }
```

Here is the call graph for this function:

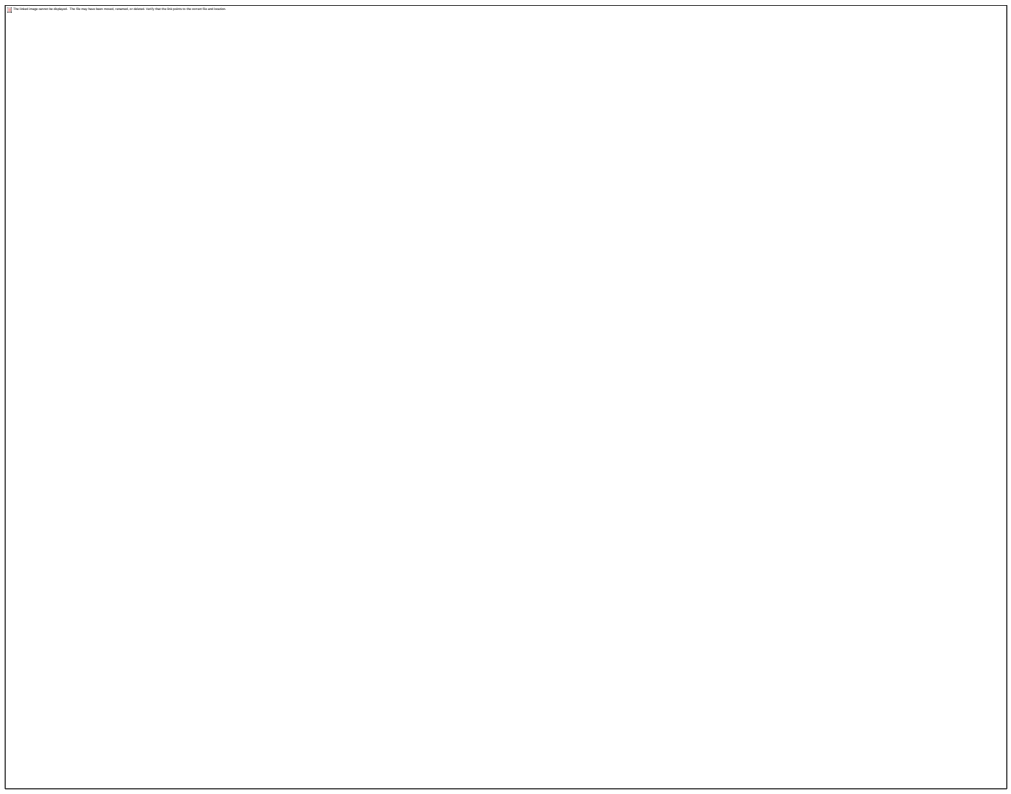


1.6.3.21 void diff (const ChebyCoeff & *f*, ChebyCoeff & *df*, int *n*)

References diff(), Spectral, ChebyCoeff::state(), and swap().

```
718                                             {
719   assert(n>=0);
720   assert(f.state() == Spectral);
721   df = f;
722   ChebyCoeff tmp;
723   for (int k=0; k<n; ++k) {
724     diff(df,tmp);
725     swap(df,tmp);
726   }
727   return;
728 }
```

Here is the call graph for this function:



1.6.3.22 void diff (const ChebyCoeff & *u*, ChebyCoeff & *dudy*)

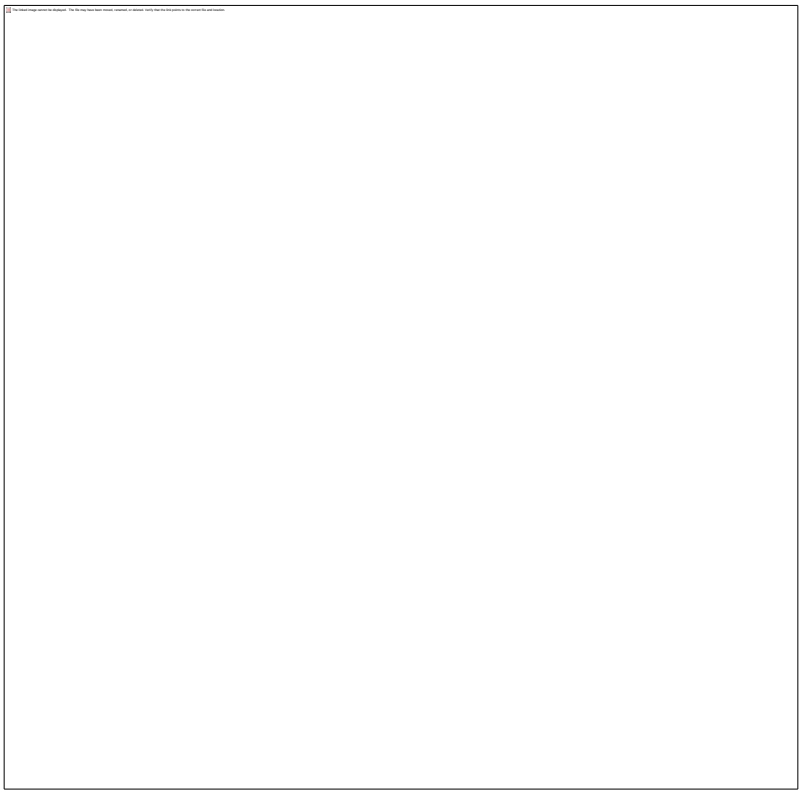
References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::L(), ChebyCoeff::numModes(), Vector::resize(), ChebyCoeff::setBounds(), ChebyCoeff::setState(), Spectral, and ChebyCoeff::state().

Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), bcDist2(), chebyProfile(), chebyVprofile(), curl(), diff(), diff2(), div(), divcheck(), divDist2(), divergence(), divNorm2(), DNSAlgorithm::DNSAlgorithm(), FlowField::dudy_a(), FlowField::dudy_b(), grad(), gradient(), TauSolver::influenceCorrection(), linearizedNL(), makeprtb(), makeroll(), makewave(), makewobl(), BasisFunc::randomize(), randomProfile(), randomVprofile(), rotationalNL(), skewsymmetricNL_task4::Run(), FlowField::saveDivSpectrum(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), TauSolver::solve(), PressureSolver::solve(), TauSolver::solve_P_and_v(), TauSolver::TauSolver(), TurbStats::ustar(), vbcFix(), TauSolver::verify(), PoissonSolver::verify(), TauSolver::verify_P_and_v(), and ydiff().

```
683                                     {
684   assert(u.state() == Spectral);
685   if (dudy.numModes() != u.numModes())
686     dudy.resize(u.numModes());
687   dudy.setBounds(u.a(), u.b());
688   dudy.setState(Spectral);
689
```

```
690   int Nb = u.numModes() - 1;
691   if (Nb==-1)
692     return;
693   if (Nb==0) {
694     dudy[0] = 0.0;
695     return;
696   }
697   Real scale=4.0/u.L();
698   dudy[Nb] = 0.0;
699   dudy[Nb-1] = scale*Nb*u[Nb];
700   for (int n=Nb-2; n>=0; --n)
701     dudy[n] = dudy[n+2] + scale*(n+1)*u[n+1];
702   dudy[0] /= 2.0;
703
704   //dudy *= 2.0/u.L();
705 }
```

Here is the call graph for this function:



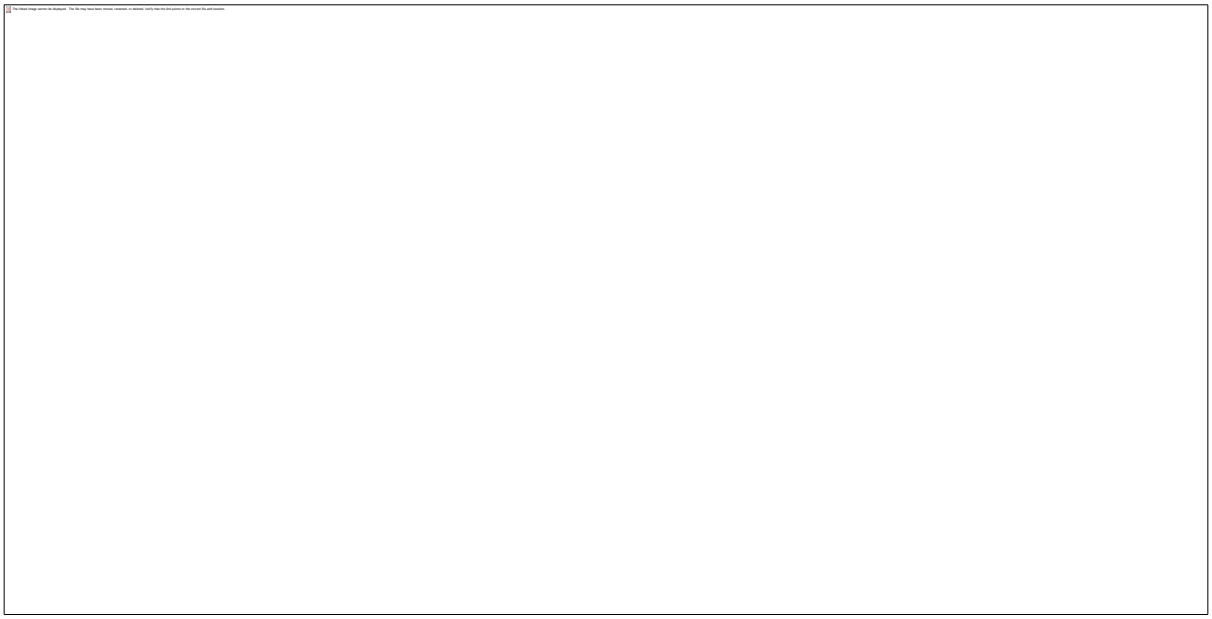
Here is the caller graph for this function:

1.6.3.23 ComplexChebyCoeff diff2 (const ComplexChebyCoeff & *u*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff2(), ComplexChebyCoeff::im, ComplexChebyCoeff::numModes(), ComplexChebyCoeff::re, and Spectral.

```
1280                                                {
1281   ComplexChebyCoeff du2(u.numModes(), u.a(), u.b(), Spectral);
1282   diff2(u.re, du2.re);
1283   diff2(u.im, du2.im);
1284   return du2;
1285 }
```

Here is the call graph for this function:

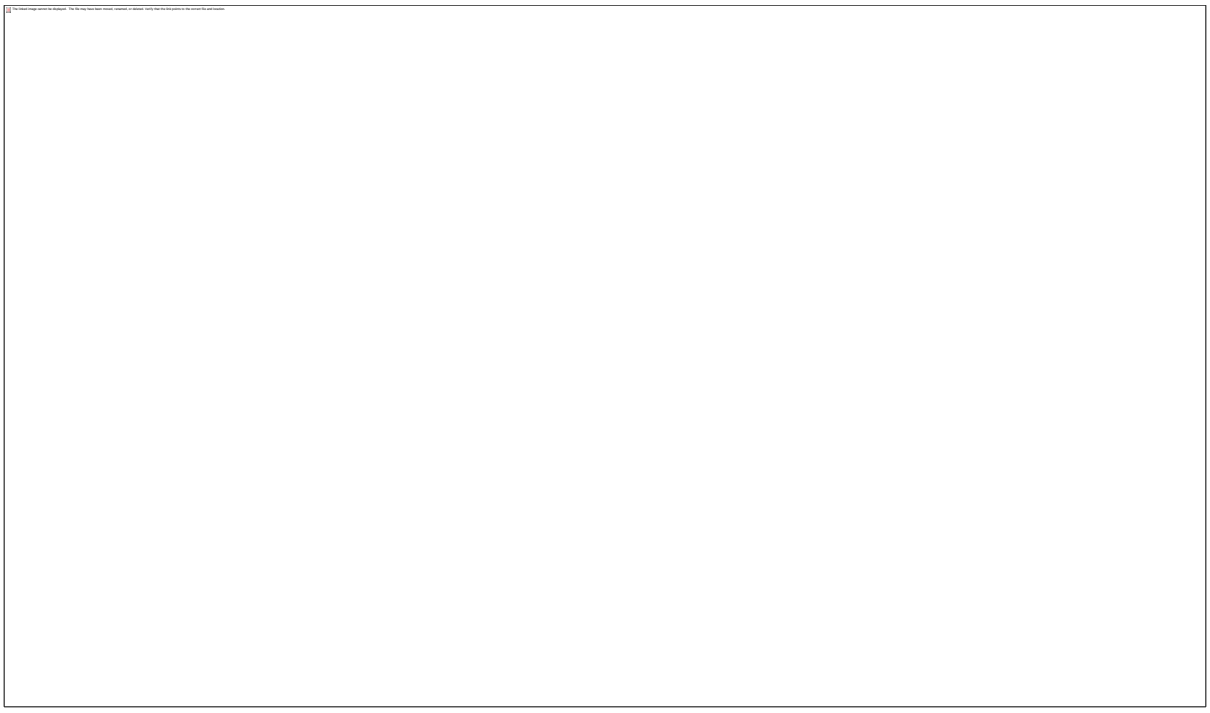


1.6.3.24 void diff2 (const ComplexChebyCoeff & *u*, ComplexChebyCoeff & *d2u*, ComplexChebyCoeff & *tmp*)

References diff2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1265                                                                {
1266   diff2(u.re, d2u.re, tmp.re);
1267   diff2(u.im, d2u.im, tmp.im);
1268 }
```

Here is the call graph for this function:



1.6.3.25 void diff2 (const ComplexChebyCoeff & *u*, ComplexChebyCoeff & *d2u*)

References diff2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1261                                                {
1262   diff2(u.re, d2u.re);
1263   diff2(u.im, d2u.im);
1264 }
```

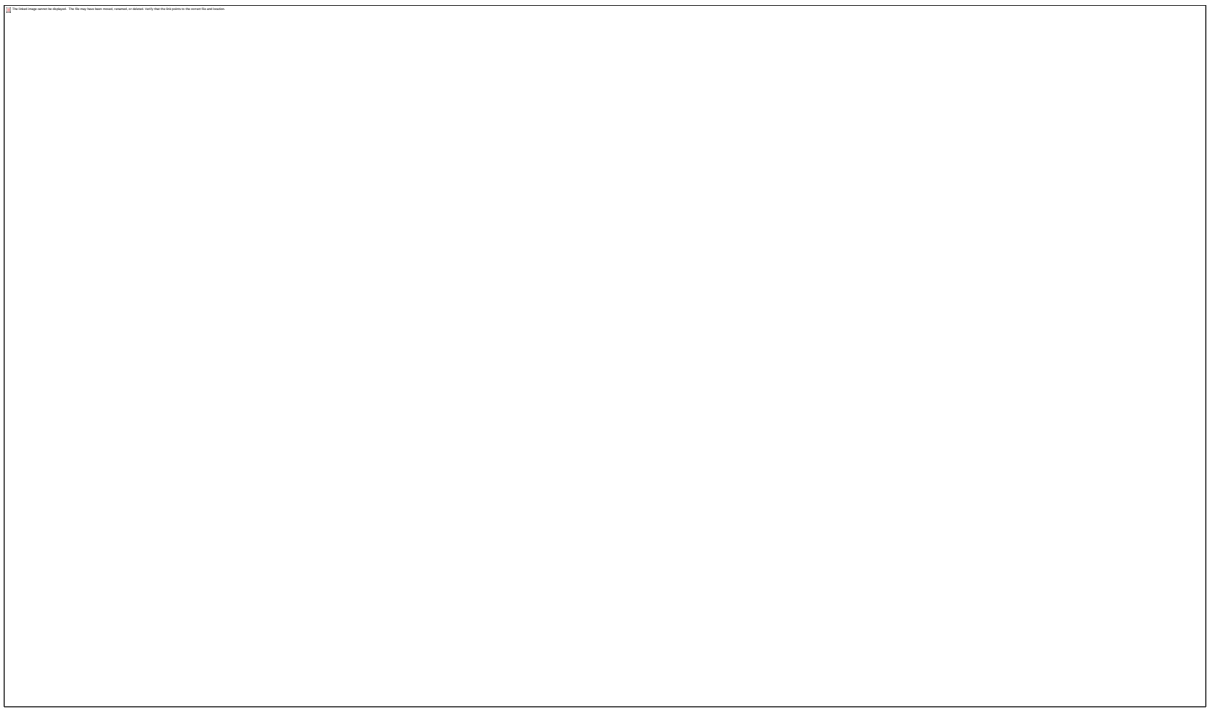
Here is the call graph for this function:

1.6.3.26 ChebyCoeff diff2 (const ChebyCoeff & *u*)

References ChebyCoeff::a(), ChebyCoeff::b(), diff2(), Vector::length(), and Spectral.

```
736                                 {
737   ChebyCoeff du2(u.length(), u.a(), u.b(), Spectral);
738   diff2(u, du2);
739   return du2;
740 }
```

Here is the call graph for this function:

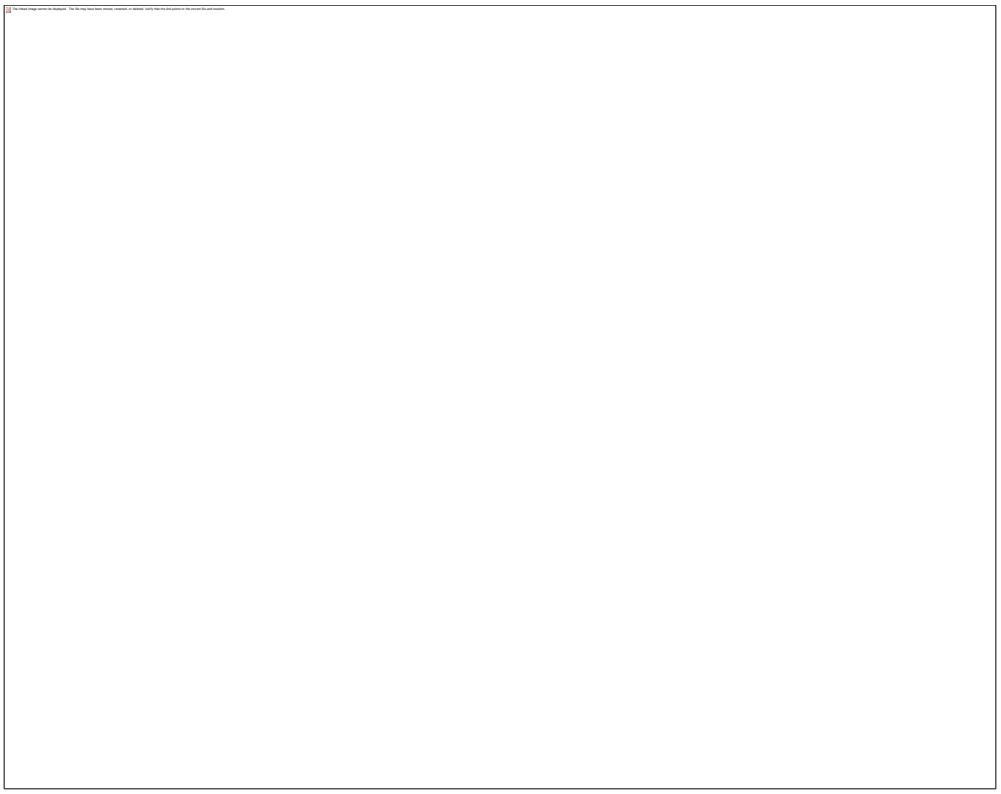


1.6.3.27 void diff2 (const ChebyCoeff & *u*, ChebyCoeff & *d2udy2*, ChebyCoeff & *tmp*)

References diff().

```
713                                                 {
714   diff(u, tmp);
715   diff(tmp, d2udy2);
716 }
```

Here is the call graph for this function:



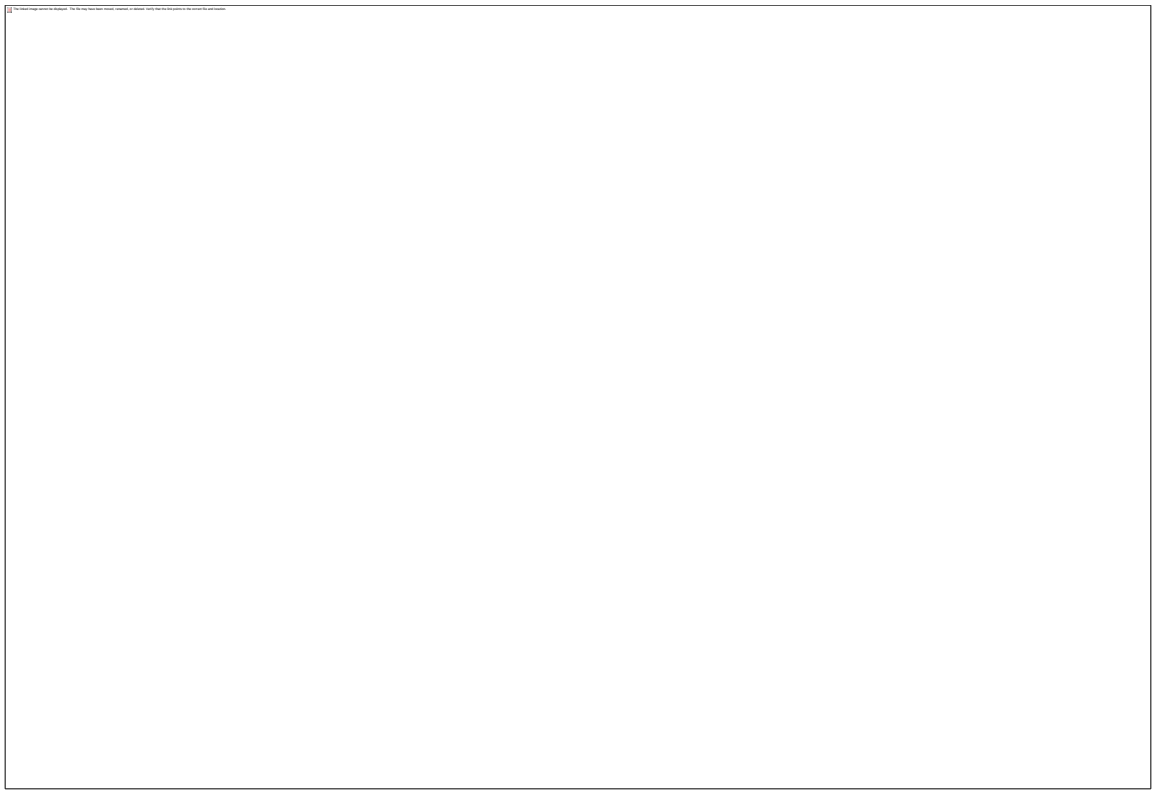
1.6.3.28 void diff2 (const ChebyCoeff & *u*, ChebyCoeff & *d2udy2*)

References ChebyCoeff::a(), ChebyCoeff::b(), diff(), Vector::length(), and Spectral.

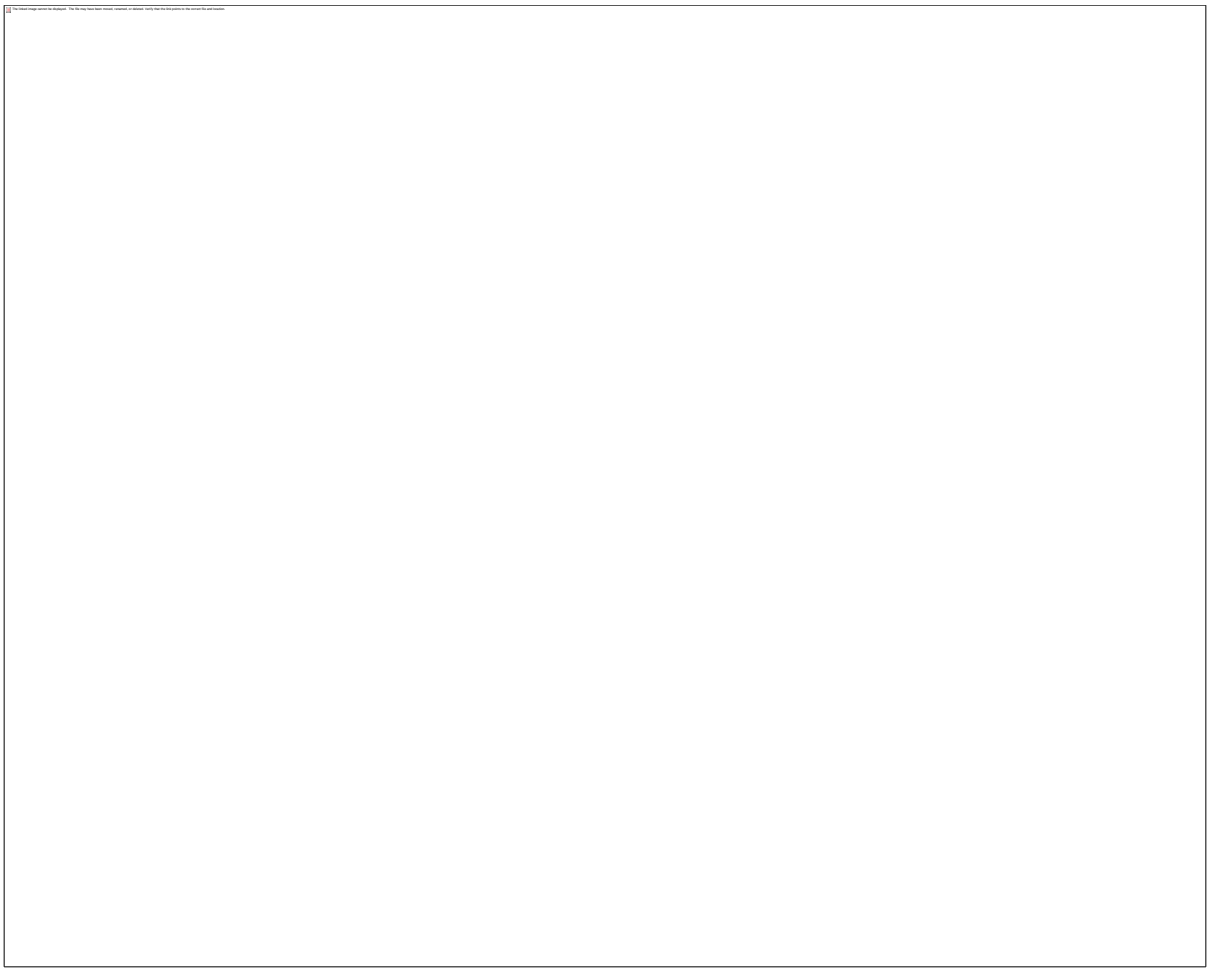
Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), diff2(), lapl(), laplacian(), HelmholtzSolver::residual(), PressureSolver::solve(), TauSolver::solve_P_and_v(), TauSolver::TauSolver(), TauSolver::verify(), and HelmholtzSolver::verify().

```
707                                     {
708   ChebyCoeff dudy(u.length(), u.a(), u.b(), Spectral);
709   diff(u, dudy);
710   diff(dudy, d2udy2);
711 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.6.3.29 Real dist (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, NormType *n*, bool *normalize*)

References chebyDist(), L2Dist(), and L2NORM.

```
1613                                    {
1614   return (n == L2NORM) ? L2Dist(u,v,normalize) : chebyDist(u,v,normalize);
1615 }
```

Here is the call graph for this function:

1.6.3.30 Real dist (const ChebyCoeff & *u*, const ChebyCoeff & *v*, NormType *n*, bool *normalize*)

References chebyDist(), L2Dist(), and L2NORM.

```
966                     {
967   return (n==L2NORM) ? L2Dist(u,v,normalize) : chebyDist(u,v,normalize);
968 }
```

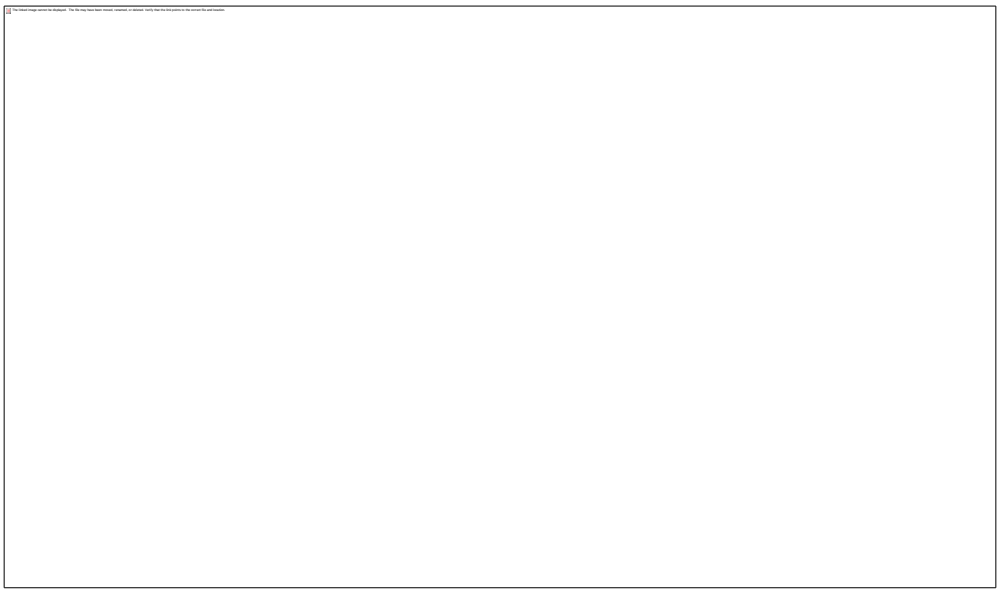
Here is the call graph for this function:

1.6.3.31 Real dist2 (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, NormType *n*, bool *normalize*)

References chebyDist2(), L2Dist2(), and L2NORM.

```
1604                                    {
1605   return (n == L2NORM) ? L2Dist2(u,v,normalize) : chebyDist2(u,v,normalize);
1606 }
```

Here is the call graph for this function:



1.6.3.32 Real dist2 (const ChebyCoeff & *u*, const ChebyCoeff & *v*, NormType *n*, bool *normalize*)

References chebyDist2(), L2Dist2(), and L2NORM.

```
958                         {
959   return (n==L2NORM) ? L2Dist2(u,v,normalize) : chebyDist2(u,v,normalize);
960 }
```

Here is the call graph for this function:

1.6.3.33 void fftw_loadwisdom (const char * *filename_*)

```
122                                     {
123
124 #ifdef HAVE_WORDEXP_H
125   //IG const char* filename = (filename_ ==0) ? "~/.fftw_wisdom" : filename_;
126   const char* filename = (filename_ ==0) ? ".fftw_wisdom" : filename_;
127
128   // This code converts filename to a full pathname.
129   wordexp_t w;
130   w.we_offs=0;
131   int flags = 0;
132   wordexp(filename, &w, flags);
133   if (w.we_wordc > 0)
134     filename = w.we_wordv[0];
135 #else
136   const char* filename = filename_;
137 #endif
138
139   // Good old byzantine C IO.
140   FILE* input_file = fopen(filename, "r");
141   if (input_file == NULL || !fftw_import_wisdom_from_file(input_file))
142     cerr << "Error reading fftw-wisdom file " << filename << ", proceding without fftw-
wisdom.\n";
143   else
144     fclose(input_file);
145 }
```

1.6.3.34 void fftw_savewisdom (const char * *filename_*)

```
146                                 {
147
148 #ifdef HAVE_WORDEXP_H
149   //IG const char* filename = (filename_ ==0) ? "~/.fftw_wisdom" : filename_;
150   const char* filename = (filename_ ==0) ? ".fftw_wisdom" : filename_;
151   wordexp_t w;
152   w.we_offs=0;
153   int flags = 0;
154   wordexp(filename, &w, flags);
155   if (w.we_wordc > 0)
156     filename = w.we_wordv[0];
157 #else
158   const char* filename = filename_;
159 #endif
160
161   FILE* output_file;
162   output_file = fopen(filename, "w");
163   if (output_file ==0)
164     cerr << "Can't write to fftw-wisdom file " << filename << "\n";
165
166   fftw_export_wisdom_to_file(output_file);
167   fclose(output_file);
168 }
```

1.6.3.35 void gaussLegendreQuadrature (int *N*, Real *a*, Real *b*, Vector & *x*, Vector & *w*)

References M_PI, and QUADRATURE_EPSILON.

```
89                                                  {
90   x = Vector(N);
91   w = Vector(N);
92   int M = (N+1)/2;
93   Real middle = 0.5*(b+a);
94   Real radius = 0.5*(b-a);
95   Real pp;
96   Real z,z1;
97
98   for (int m=0; m<M; ++m) {
99     z = cos(M_PI*(m+0.75)/(N+0.5));
100     Real p1,p2,p3;
101     int iterations=0;
102     do {
103       p1 = 1.0;
```

```
104       p2 = 0.0;
105       for (int n=0; n<N; ++n) {
106         p3 = p2;
107         p2 = p1;
108         p1 = ((2*n+1)*z*p2-n*p3)/(n+1);
109       }
110       pp = N*(z*p1-p2)/(z*z-1.0);
111       z1=z;
112       z=z1-p1/pp;
113     } while (fabs(z-z1) > QUADRATURE_EPSILON && ++iterations < 1024);
114
115     x(m) = middle - radius*z;
116     x(N-1-m) = middle + radius*z;
117     w(m) = 2.0*radius/((1.0-z*z)*pp*pp);
118     w(N-1-m) = w(m);
119   }
120 }
```

1.6.3.36 Complex innerProduct (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, NormType *n*, bool *normalize*)

References chebyInnerProduct(), L2InnerProduct(), and L2NORM.

```
1617                                        {
1618   return (n == L2NORM) ? L2InnerProduct(u,v,normalize)
: chebyInnerProduct(u,v,normalize);
1619 }
```

Here is the call graph for this function:

1.6.3.37 Real innerProduct (const ChebyCoeff & *u*, const ChebyCoeff & *v*, NormType *n*, bool *normalize*)

References chebyInnerProduct(), L2InnerProduct(), and L2NORM.

```
970                                    {
971   return (n==L2NORM)
? L2InnerProduct(u,v,normalize):chebyInnerProduct(u,v,normalize);
972 }
```

Here is the call graph for this function:

1.6.3.38 ComplexChebyCoeff integrate (const ComplexChebyCoeff & *du*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), ComplexChebyCoeff::im, integrate(), ComplexChebyCoeff::numModes(), ComplexChebyCoeff::re, and Spectral.

```
1298                                          {
1299   ComplexChebyCoeff u(du.numModes(), du.a(), du.b(), Spectral);
1300   integrate(du.re, u.re);
1301   integrate(du.im, u.im);
1302   return u;
1303 }
```

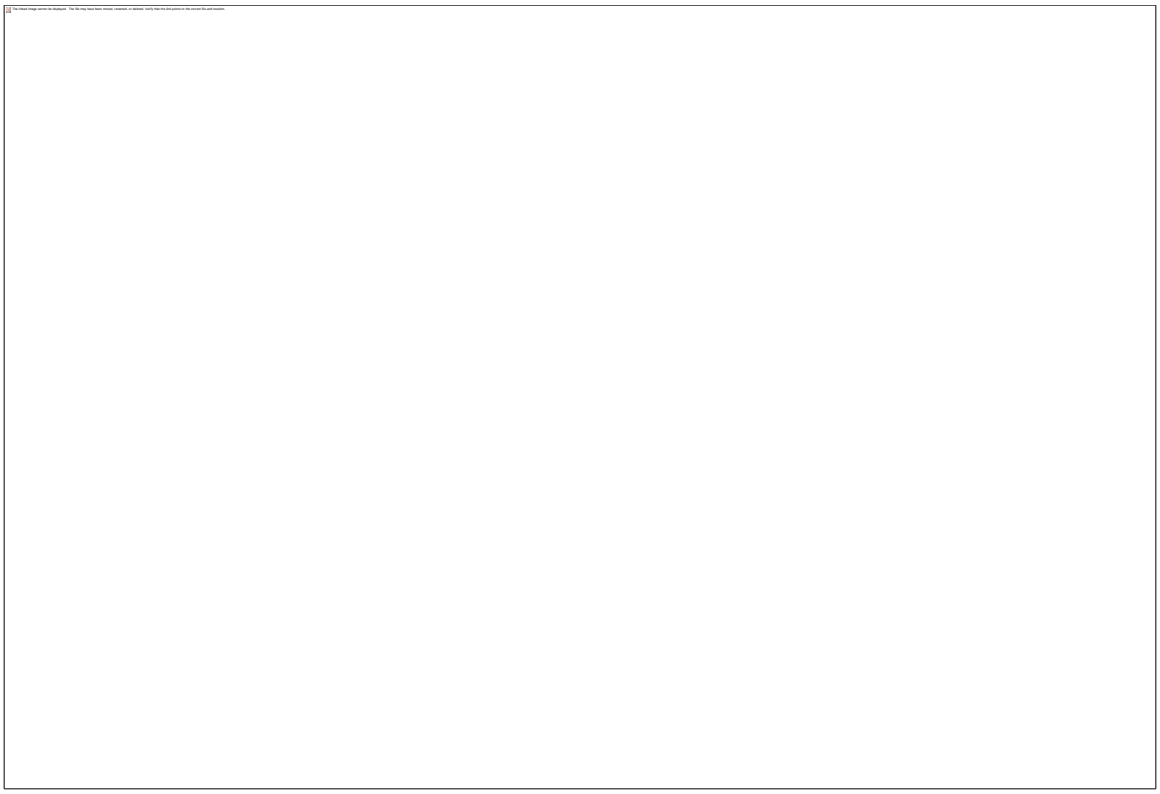
Here is the call graph for this function:

1.6.3.39 void integrate (const ComplexChebyCoeff & *du*, ComplexChebyCoeff & *u*)

References ComplexChebyCoeff::im, integrate(), and ComplexChebyCoeff::re.

```
1293                                                    {
1294   integrate(du.re, u.re);
1295   integrate(du.im, u.im);
1296 }
```

Here is the call graph for this function:

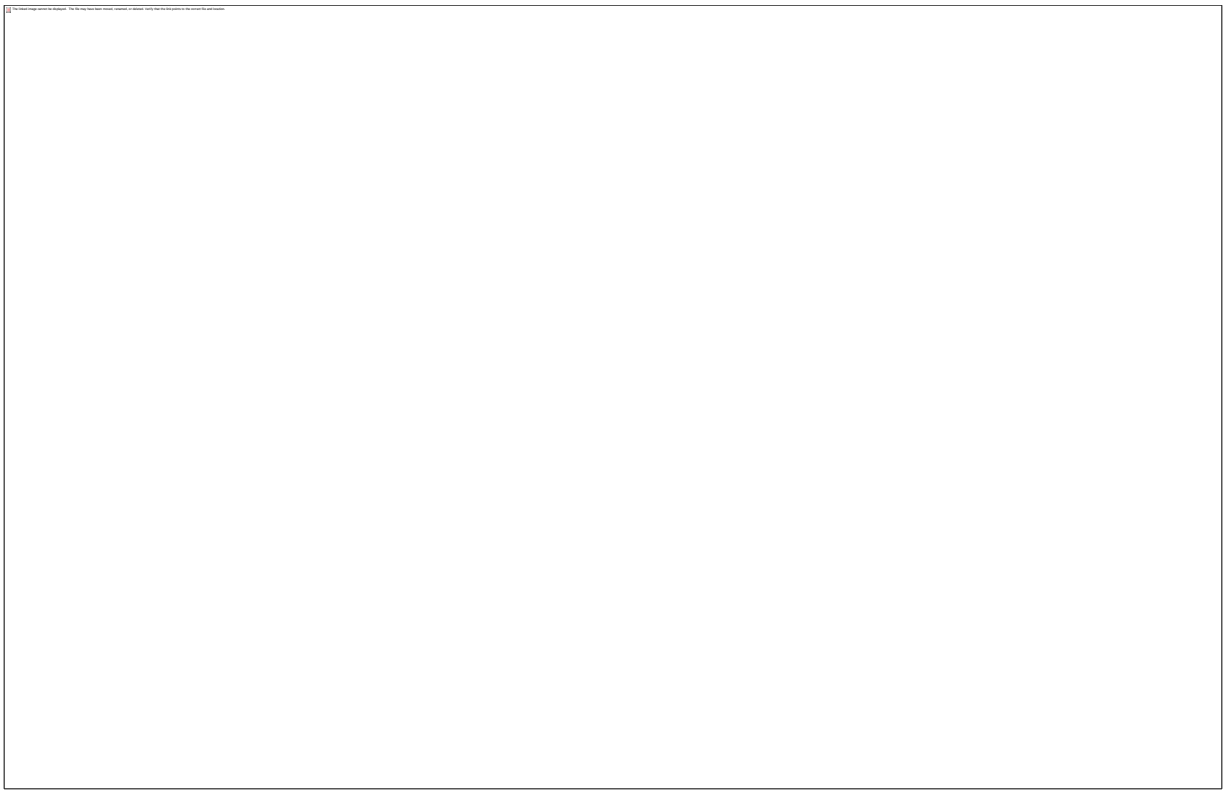


1.6.3.40 ChebyCoeff integrate (const ChebyCoeff & *dudy*)

References ChebyCoeff::a(), ChebyCoeff::b(), integrate(), Vector::length(), and Spectral.

```
676                                         {
677   ChebyCoeff u(dudy.length(), dudy.a(), dudy.b(), Spectral);
678   integrate(dudy, u);
679   return u;
680 }
```

Here is the call graph for this function:



1.6.3.41 void integrate (const ChebyCoeff & *dudy*, ChebyCoeff & *u*)

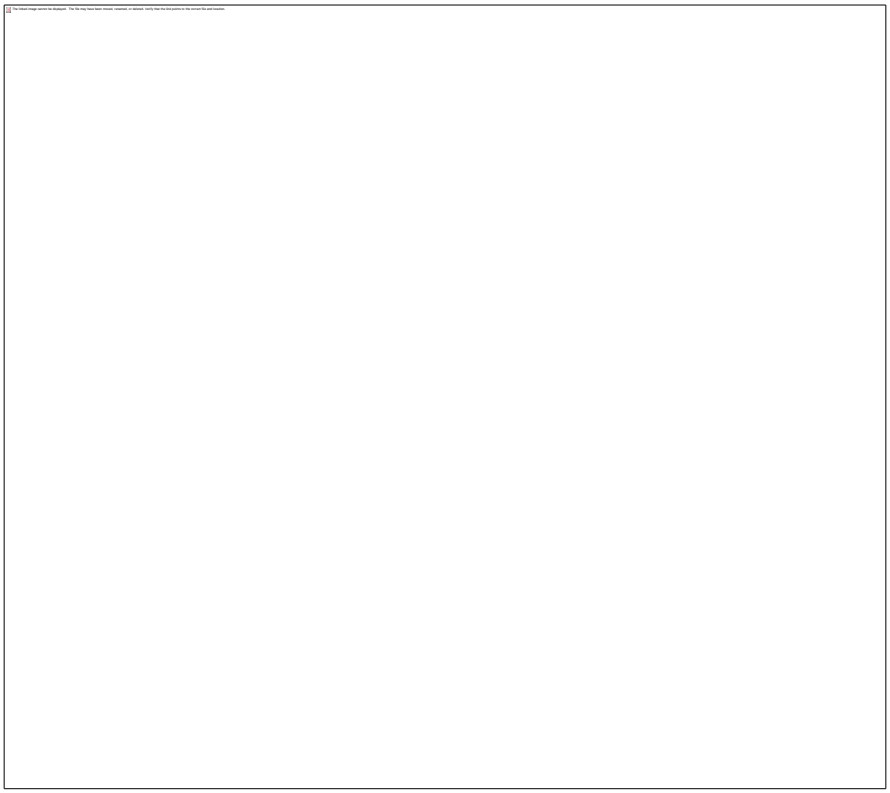
References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::mean(), ChebyCoeff::numModes(), Vector::resize(), ChebyCoeff::setBounds(), ChebyCoeff::setState(), Spectral, and ChebyCoeff::state().

Referenced by integrate(), L1Dist(), and L1Norm().

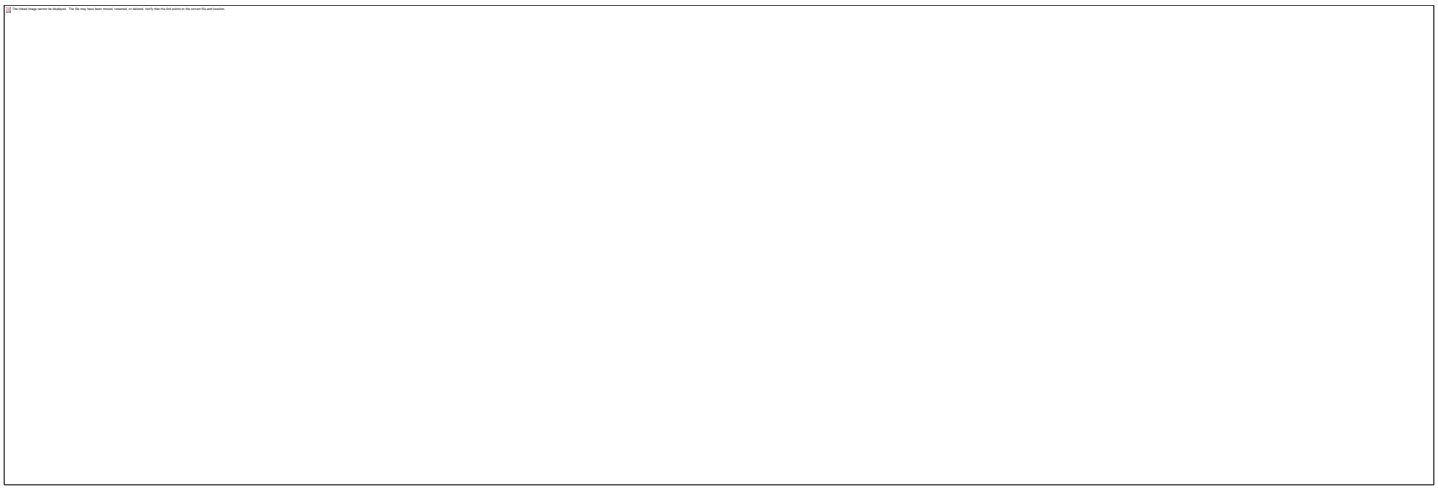
```
646                                        {
647   assert(dudy.state() == Spectral);
648   int N=dudy.numModes();
649   if (u.numModes() != N)
650     u.resize(N);
651   u.setBounds(u.a(), u.b());
652   u.setState(Spectral);
653
654   Real h2 = (dudy.b()-dudy.a())/2;
655   switch (N) {
656   case 0:
657     break;
658   case 1:
659     u[0] = 0.0;
660     break;
661   case 2:
```

```
662     u[0] = 0;
663     u[1] = h2*dudy[0];
664     break;
665   default:
666     u[1] = h2*(dudy[0] - dudy[2]/2);
667     for (int n=2; n<N-1; ++n)
668       u[n] = h2*(dudy[n-1] - dudy[n+1])/(2*n);
669     u[N-1] = h2*dudy[N-2]/(2*(N-1));
670     u[0] -= u.mean();  // const term is arbitrary, set to zero.
671   }
672   return;
673 
674 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

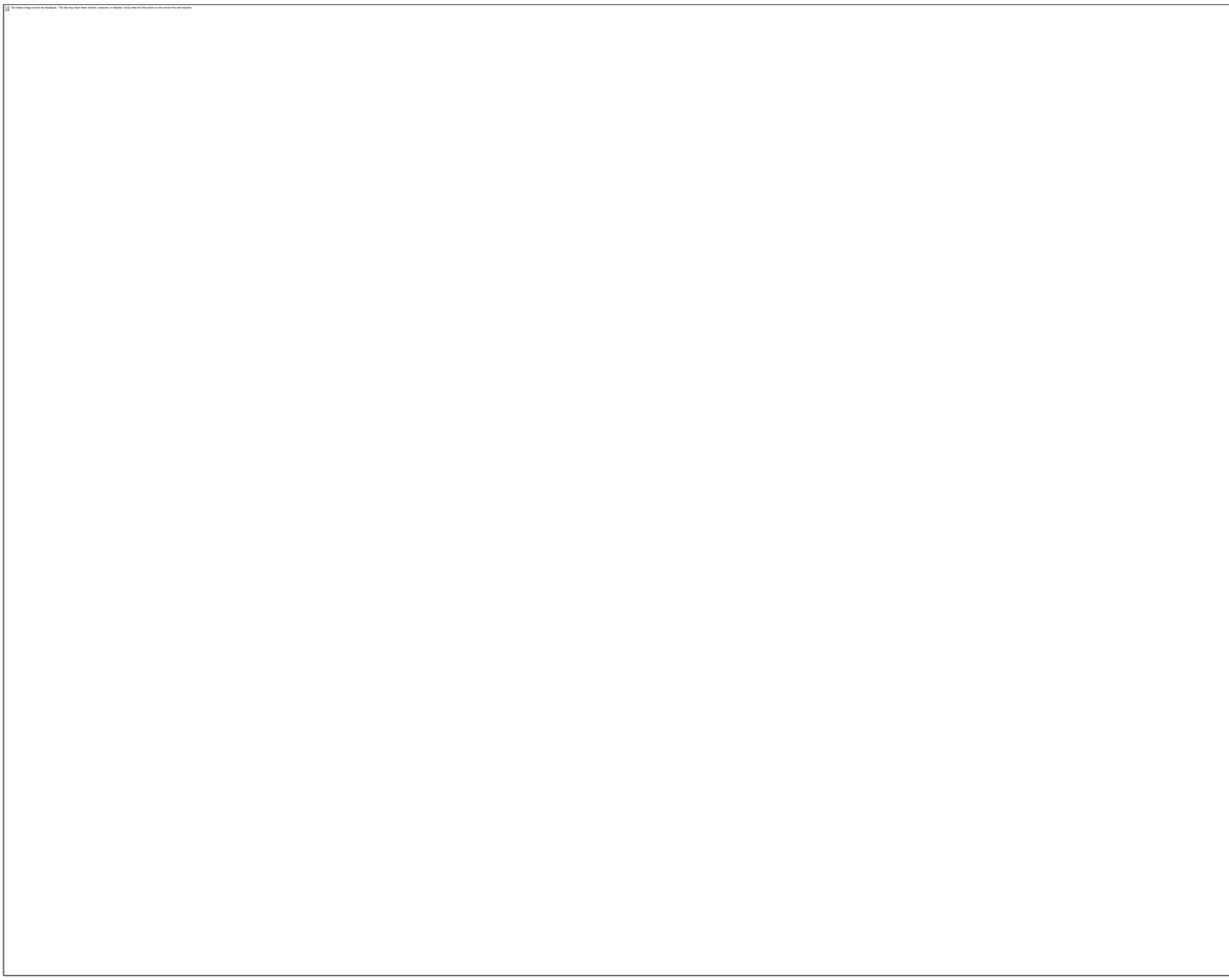


1.6.3.42 Real L1Dist (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, bool *normalize*)

References ComplexChebyCoeff::im, L1Dist(), and ComplexChebyCoeff::re.

```
1536                        {
1537   return L1Dist(u.re, v.re, normalize) + L1Dist(u.im, v.im, normalize);
1538 }
```

Here is the call graph for this function:



1.6.3.43 Real L1Dist (const ChebyCoeff & *f*, const ChebyCoeff & *g*, bool *normalize*)

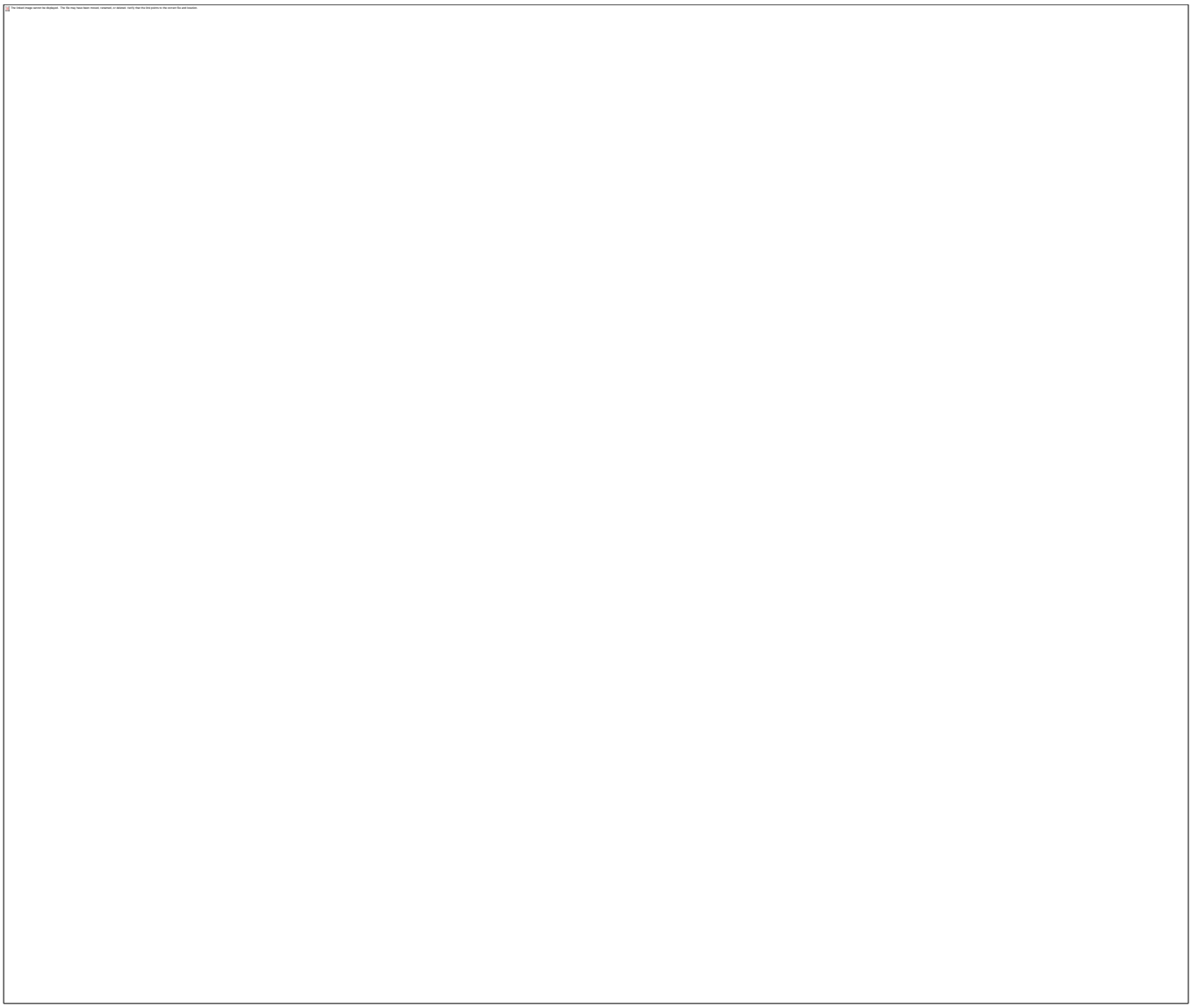
References ChebyCoeff::a(), abs(), ChebyCoeff::b(), ChebyCoeff::congruent(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), integrate(), ChebyCoeff::makeSpectral(), ChebyCoeff::N(), and ChebyCoeff::state().

Referenced by L1Dist(), and HelmholtzSolver::residual().

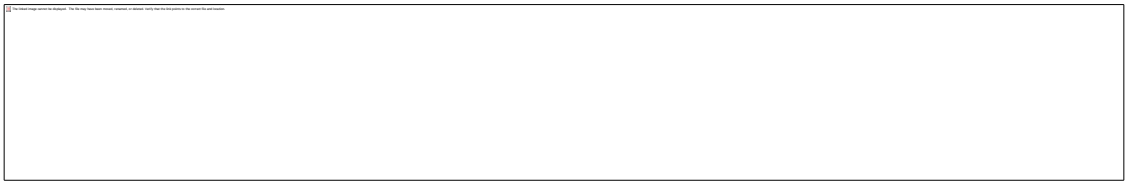
```
925                                                    {
926   assert(f.congruent(g));
927   fieldstate fs = f.state();
928   fieldstate gs = f.state();
929   ((ChebyCoeff&) f).makePhysical();
930   ((ChebyCoeff&) g).makePhysical();
```

```
931
932   ChebyCoeff d(f);
933   for (int n=0; n<g.N(); ++n)
934     d[n] = abs(f[n]-g[n]);
935   d.makeSpectral();
936
937   ChebyCoeff D(g);
938   integrate(d,D);
939
940   Real rtn = D.eval_b() - D.eval_a();
941
942   if (normalize)
943     rtn /= d.b() - d.a();
944
945   ((ChebyCoeff&) f).makeState(fs);
946   ((ChebyCoeff&) g).makeState(gs);
947
948   return rtn;
949 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



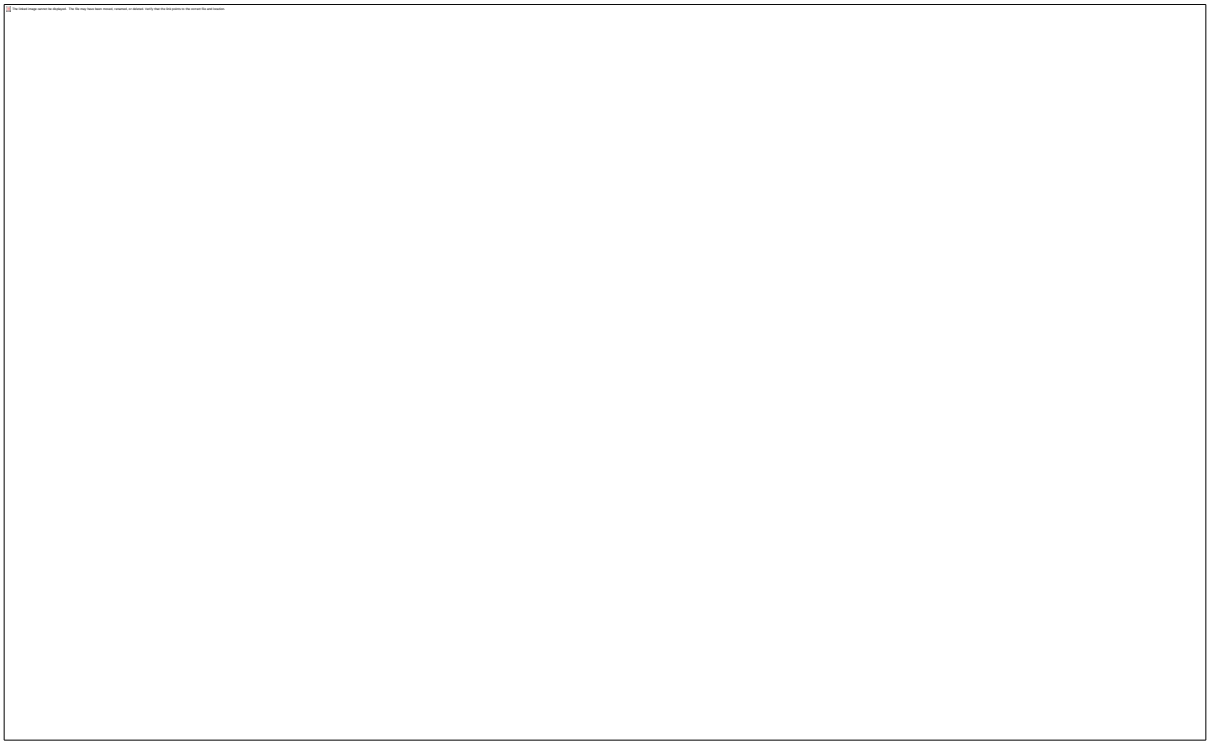
1.6.3.44 Real L1Norm (const ComplexChebyCoeff & *u*, bool *normalize*)

References ComplexChebyCoeff::im, L1Norm(), and ComplexChebyCoeff::re.

```
1531                                                {
1532   return L1Norm(u.re, normalize) + L1Norm(u.im, normalize);
```

```
1533 }
```

Here is the call graph for this function:



1.6.3.45 Real L1Norm (const ChebyCoeff & *f*, bool *normalize*)

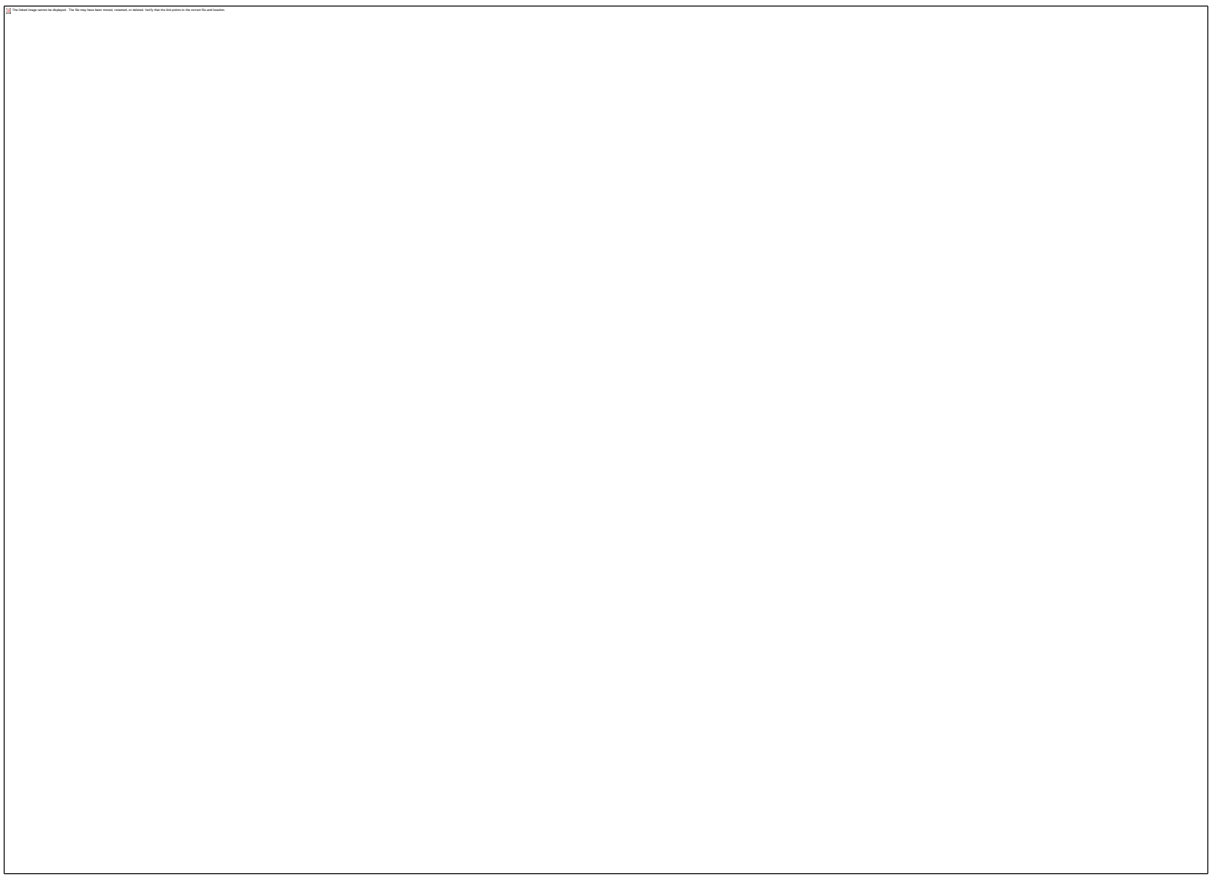
References ChebyCoeff::a(), abs(), ChebyCoeff::b(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), integrate(), ChebyCoeff::makePhysical(), ChebyCoeff::makeSpectral(), and ChebyCoeff::N().

Referenced by L1Dist(), L1Norm(), and HelmholtzSolver::verify().

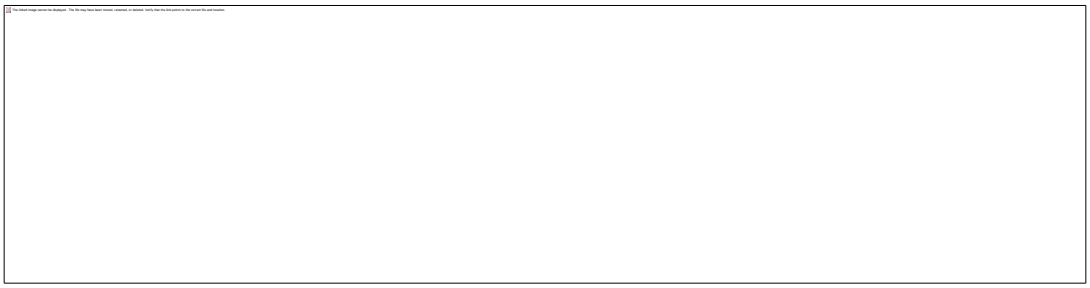
```
907                                     {
908   ChebyCoeff g(f);
909   g.makePhysical();
910   for (int n=0; n<g.N(); ++n)
911     g[n] = abs(g[n]);
912   g.makeSpectral();
913
914   ChebyCoeff G(g);
915   integrate(g,G);
916
917   Real rtn = G.eval_b() - G.eval_a();
918
919   if (normalize)
920     rtn /= g.b() - g.a();
```

```
921
922   return rtn;
923 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

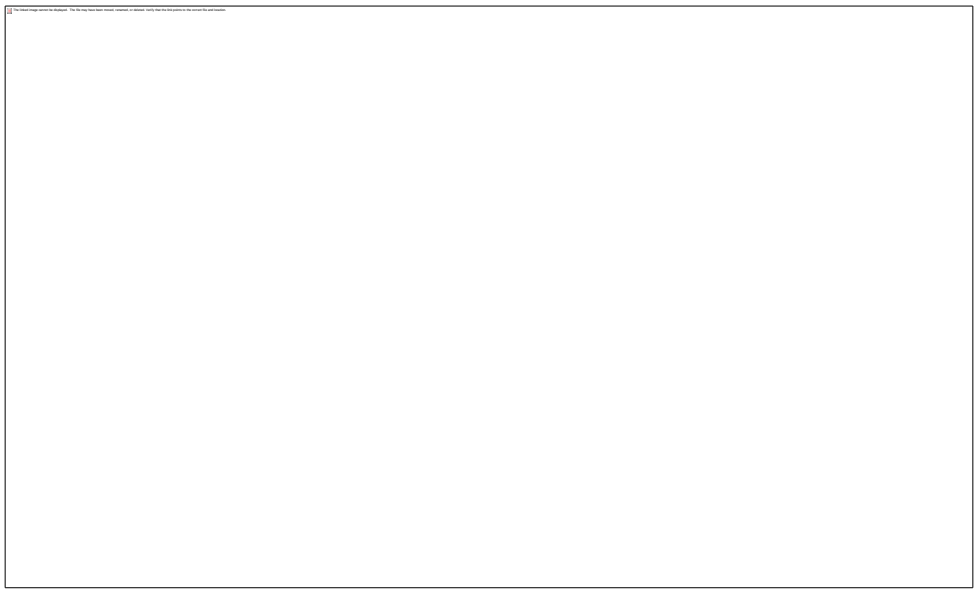


1.6.3.46 Real L2Dist (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, bool *normalize*)

References ComplexChebyCoeff::im, L2Dist2(), and ComplexChebyCoeff::re.

```
1484                         {
1485   return sqrt(L2Dist2(u.re, v.re, normalize) + L2Dist2(u.im, v.im, normalize));
1486 }
```

Here is the call graph for this function:



1.6.3.47 Real L2Dist (const ChebyCoeff & *u*, const ChebyCoeff & *v*, bool *normalize*)

References L2Dist2().

```
800                                                 {
801   return sqrt(L2Dist2(u,v, normalize));
802 }
```

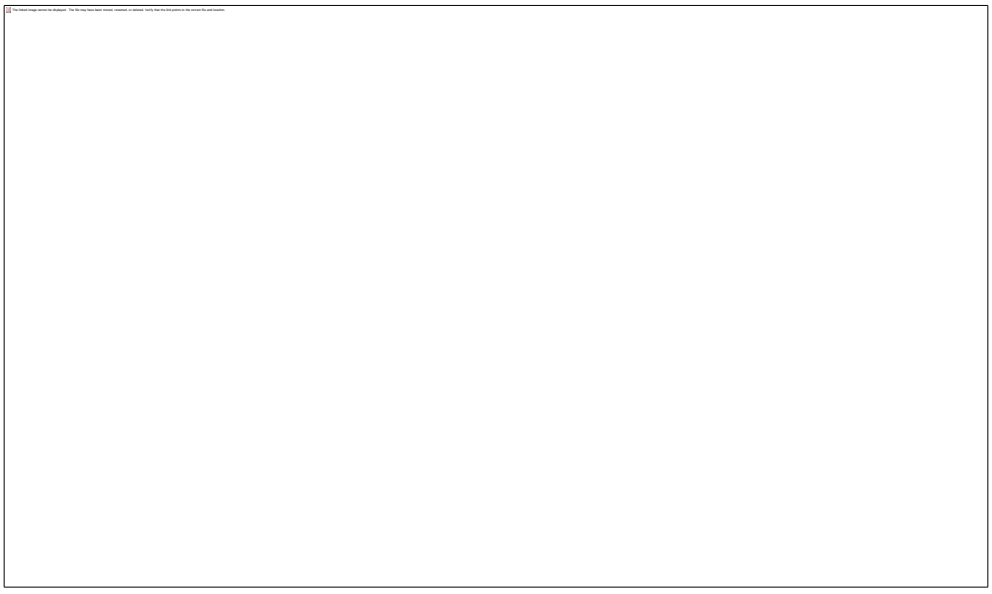
Here is the call graph for this function:

1.6.3.48 Real L2Dist2 (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, bool *normalize*)

References ComplexChebyCoeff::im, L2Dist2(), and ComplexChebyCoeff::re.

```
1475                                {
1476   return L2Dist2(u.re, v.re, normalize) + L2Dist2(u.im, v.im, normalize);
1477 }
```

Here is the call graph for this function:

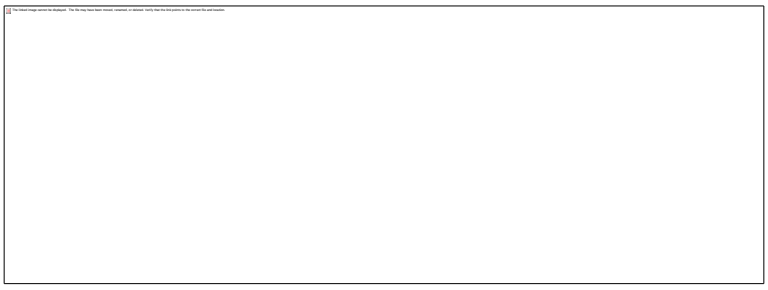


1.6.3.49 Real L2Dist2 (const ChebyCoeff & *u*, const ChebyCoeff & *v*, bool *normalize*)

References L2Norm2().

```
790                                                 {
791   ChebyCoeff tmp(u);
792   tmp -= v;
793   return L2Norm2(tmp, normalize);
794 }
```

Here is the call graph for this function:



1.6.3.50 Complex L2InnerProduct (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, bool *normalize*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), ComplexChebyCoeff::im, ComplexChebyCoeff::numModes(), ComplexChebyCoeff::re, Spectral, and ComplexChebyCoeff::state().

```
1489                                 {
1490   assert(u.state()==Spectral && v.state()==Spectral);
1491   assert(u.numModes() == v.numModes());
1492   int N = u.numModes();
1493
1494   // Unroll the complex operations, for efficiency.
1495   // Profiling an L2InnerProduct-intensive program reveals that
1496   // complex * double is expensive
1497   Real e=1.0;
1498   Real sum_r=0.0;
1499   Real sum_i=0.0;
1500   Real psum_r;
1501   Real psum_i;
1502   Real um_r;
1503   Real um_i;
1504   Real vn_r;
1505   Real vn_i;
1506   Real k;
1507   for (int m=N-1; m>=0; --m) {
```

```
1508     um_r = u.re[m];
1509     um_i = u.im[m];
1510     psum_r = 0.0;
1511     psum_i = 0.0;
1512     for (int n=m%2; n<N; n+=2) {
1513       k = (e-m*m-n*n)/((e+m-n)*(e-m+n)*(e+m+n)*(e-m-n));
1514       vn_r = v.re[n];
1515       vn_i = v.im[n];
1516 
1517       psum_r += k*(um_r*vn_r + um_i*vn_i);
1518       psum_i += k*(um_i*vn_r - um_r*vn_i);
1519     }
1520     sum_r += psum_r;
1521     sum_i += psum_i;
1522   }
1523   if (!normalize) {
1524     sum_r *= u.b()-u.a();
1525     sum_i *= u.b()-u.a();
1526   }
1527   return Complex(sum_r, sum_i);
1528 }
```

Here is the call graph for this function:

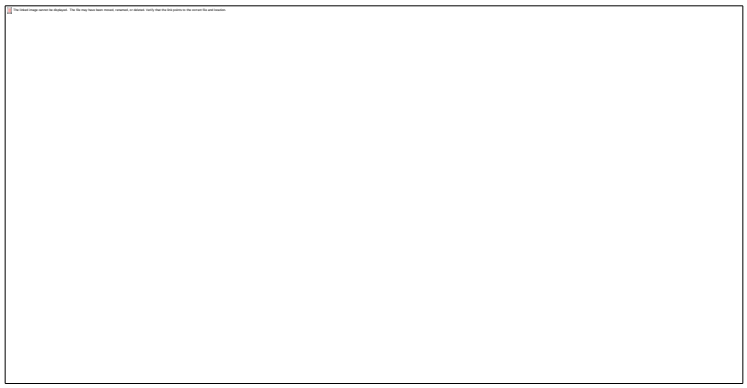
1.6.3.51 Real L2InnerProduct (const ChebyCoeff & *u*, const ChebyCoeff & *v*, bool *normalize*)

References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::numModes(), Spectral, and ChebyCoeff::state().

```
804                                                    {
805   assert(u.state()==Spectral && v.state()==Spectral);
806   assert(u.a()==v.a() && u.b()==v.b());
807   assert(u.numModes() ==  v.numModes());
```

```
808   int N = u.numModes();
809   Real sum=0.0;
810   Real e=1.0;
811   for (int m=N-1; m>=0; --m) {
812     Real um = u[m];
813     Real psum = 0.0;
814     for (int n=m%2; n<N; n+=2)
815       psum += um*v[n]*(e-m*m-n*n)/((e+m-n)*(e-m+n)*(e+m+n)*(e-m-n));
816     sum += psum;
817   }
818   if (!normalize)
819     sum *= u.b()-u.a();
820   return sum;
821 }
```

Here is the call graph for this function:

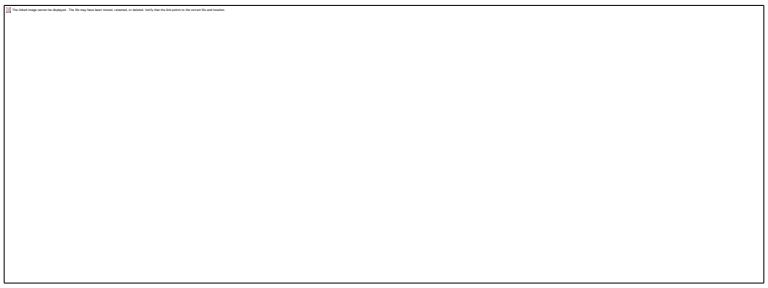


1.6.3.52 Real L2Norm (const ComplexChebyCoeff & *u*, bool *normalize*)

References ComplexChebyCoeff::im, L2Norm2(), and ComplexChebyCoeff::re.

```
1479                                                {
1480   return sqrt(L2Norm2(u.re, normalize) + L2Norm2(u.im, normalize));
1481 }
```

Here is the call graph for this function:

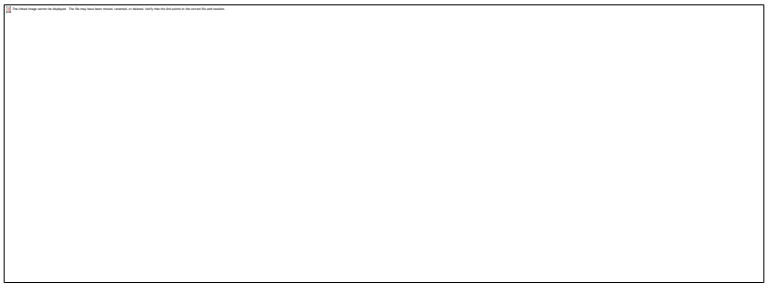


1.6.3.53 Real L2Norm (const ChebyCoeff & *u*, bool *normalize*)

References L2Norm2().

```
796                                         {
797   return sqrt(L2Norm2(u, normalize));
798 }
```

Here is the call graph for this function:

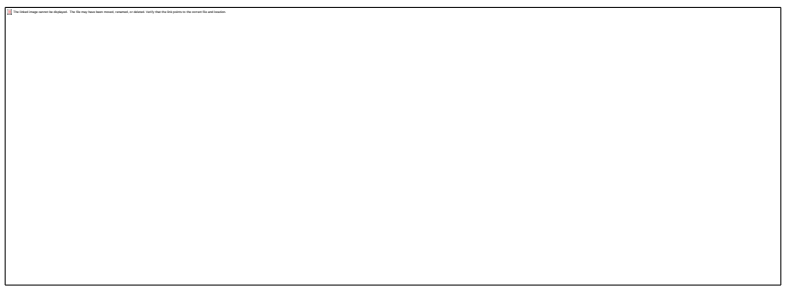


1.6.3.54 Real L2Norm2 (const ComplexChebyCoeff & *u*, bool *normalize*)

References ComplexChebyCoeff::im, L2Norm2(), and ComplexChebyCoeff::re.

```
1470                                                {
1471   return L2Norm2(u.re, normalize) + L2Norm2(u.im, normalize);
1472 }
```

Here is the call graph for this function:



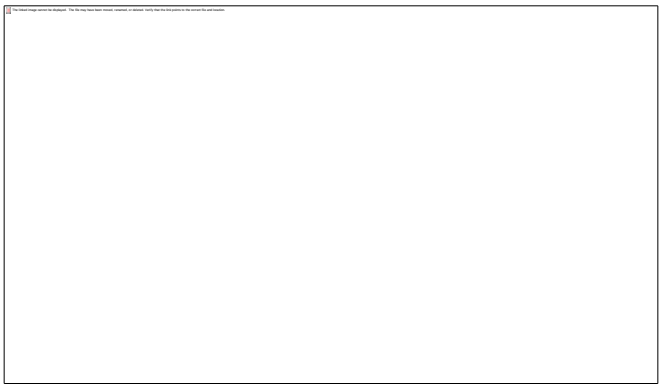
1.6.3.55 Real L2Norm2 (const ChebyCoeff & *u*, bool *normalize*)

References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::numModes(), Spectral, and ChebyCoeff::state().

```
773                                         {
774   assert(u.state() == Spectral);
```

```
775   int N=u.numModes();
776   Real sum=0.0;
777   Real e=1.0;
778   for (int m=N-1; m>=0; --m) {
779     Real um = u[m];
780     Real psum = 0.0;
781     for (int n=m%2; n<N; n+=2)
782       psum += um*u[n]*(e-m*m-n*n)/((e+m-n)*(e-m+n)*(e+m+n)*(e-m-n));
783     sum += psum;
784   }
785   if (!normalize)
786     sum *= u.b()-u.a();
787   return sum;
788 }
```

Here is the call graph for this function:

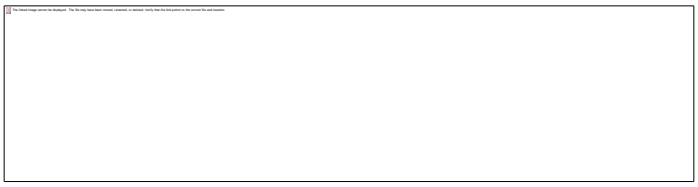


1.6.3.56 ChebyCoeff legendre (int *N*, int *n*, bool *normalize*)

References legendre(), ChebyCoeff::makeSpectral(), Physical, and pi.

```
66                                          {
67   assert(n<N);
68   ChebyCoeff Pn(N, -1 , 1, Physical);
69   Real piN=pi/(N-1);
70   for (int q=0; q<N; ++q)
71     Pn[q] = legendre(n, cos(q*piN));
72   ChebyTransform trans(N);
73   Pn.makeSpectral(trans);
74   if (normalize)
75     Pn *= sqrt((double) (2*n+1));
76   return Pn;
77 }
```

Here is the call graph for this function:



1.6.3.57 Real legendre (int *n*, Real *x*)

Referenced by legendre().

```
51                          {
52   Real p=1.0;
53   Real q=0.0;
54   for (int m=0; m<n; ++m) {
55     Real r = q;
56     q = p;
57     p = ((2*m+1)*x*q-m*r)/(m+1);
58   }
59   return p;
60 }
```

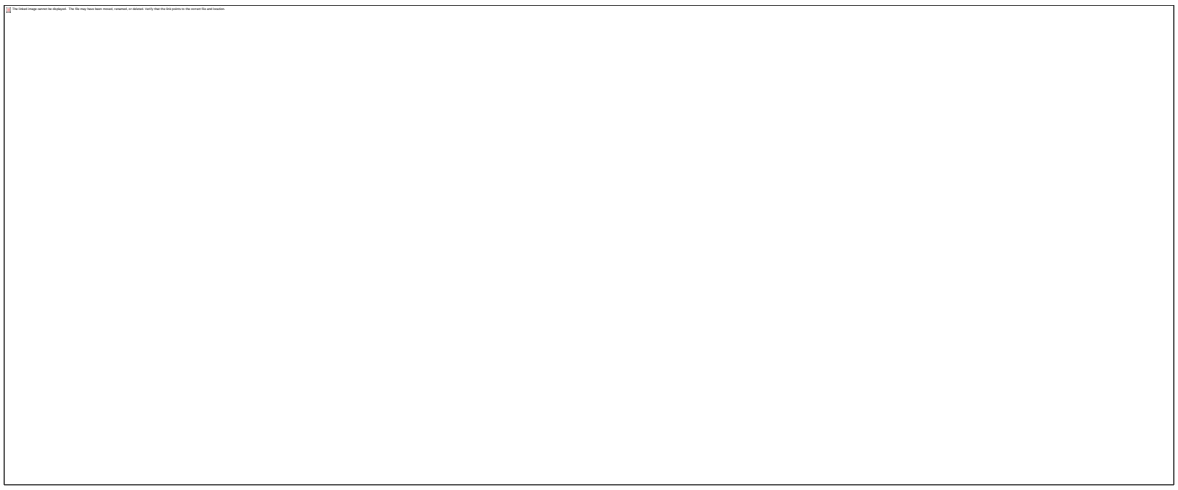
Here is the caller graph for this function:

1.6.3.58 Real LinfDist (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*)

References ComplexChebyCoeff::im, LinfDist(), and ComplexChebyCoeff::re.

```
1544                                                    {
1545   return LinfDist(u.re, v.re) + LinfDist(u.im, v.im);
1546 }
```

Here is the call graph for this function:



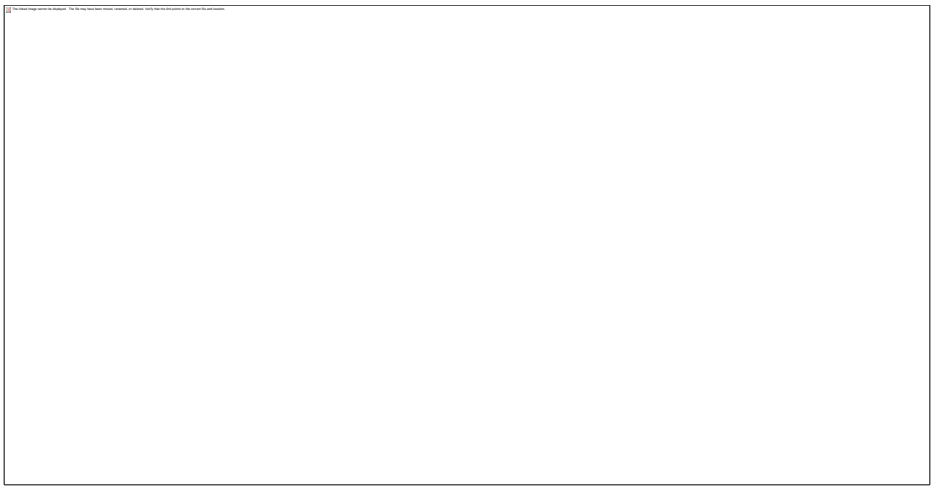
1.6.3.59 Real LinfDist (const ChebyCoeff & *f*, const ChebyCoeff & *g*)

References abs(), ChebyCoeff::congruent(), Greater(), ChebyCoeff::N(), and ChebyCoeff::state().

Referenced by LinfDist().

```
890                                             {
891   assert(f.congruent(g));
892   fieldstate fs = f.state();
893   fieldstate gs = f.state();
894   ((ChebyCoeff&) f).makePhysical();
895   ((ChebyCoeff&) g).makePhysical();
896
897   Real rtn=0.0;
898   for (int m=0; m<f.N(); ++m)
899     rtn = Greater(rtn, abs(f[m]-g[m]));
900
901   ((ChebyCoeff&) f).makeState(fs);
902   ((ChebyCoeff&) g).makeState(gs);
903   return rtn;
904 }
```

Here is the call graph for this function:



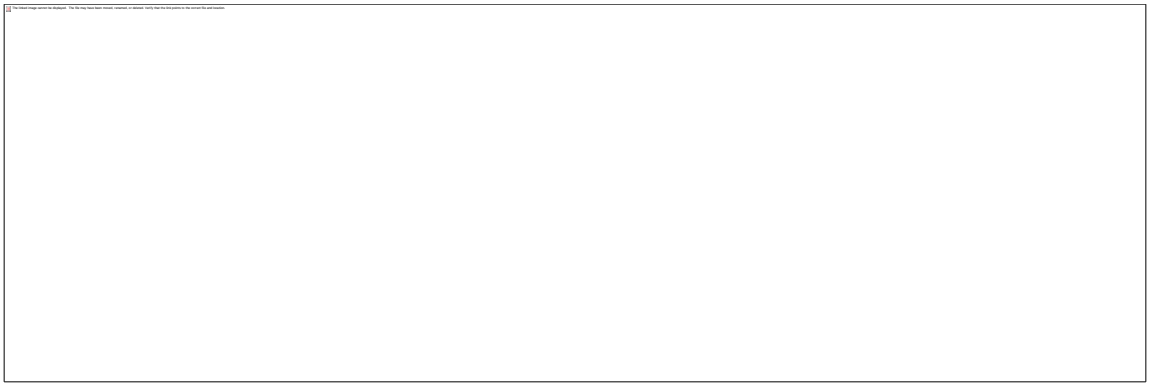
Here is the caller graph for this function:

1.6.3.60 Real LinfNorm (const ComplexChebyCoeff & *u*)

References ComplexChebyCoeff::im, LinfNorm(), and ComplexChebyCoeff::re.

```
1540                                    {
1541   return LinfNorm(u.re) + LinfNorm(u.im);
1542 }
```

Here is the call graph for this function:



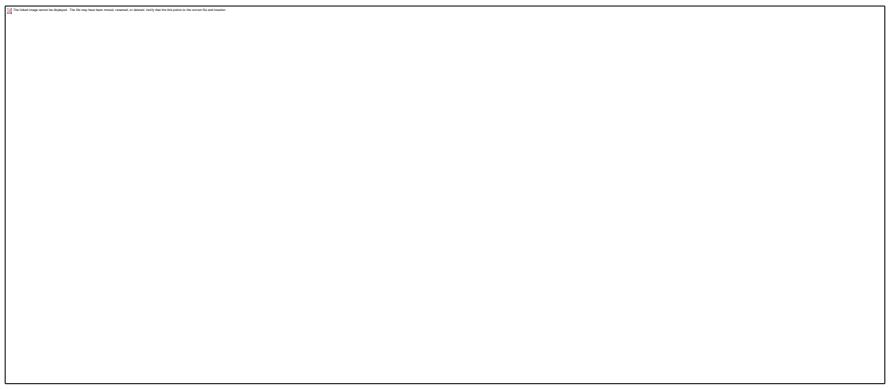
1.6.3.61 Real LinfNorm (const ChebyCoeff & *f*)

References abs(), Greater(), ChebyCoeff::N(), and ChebyCoeff::state().

Referenced by LinfDist(), and LinfNorm().

```
877                                 {
878   fieldstate s = f.state();
879   ((ChebyCoeff&) f).makePhysical();
880
881   Real rtn=0.0;
882   for (int m=0; m<f.N(); ++m)
883     rtn = Greater(abs(f[m]), rtn);
884
885   ((ChebyCoeff&) f).makeState(s);
886
887   return rtn;
888 }
```

Here is the call graph for this function:



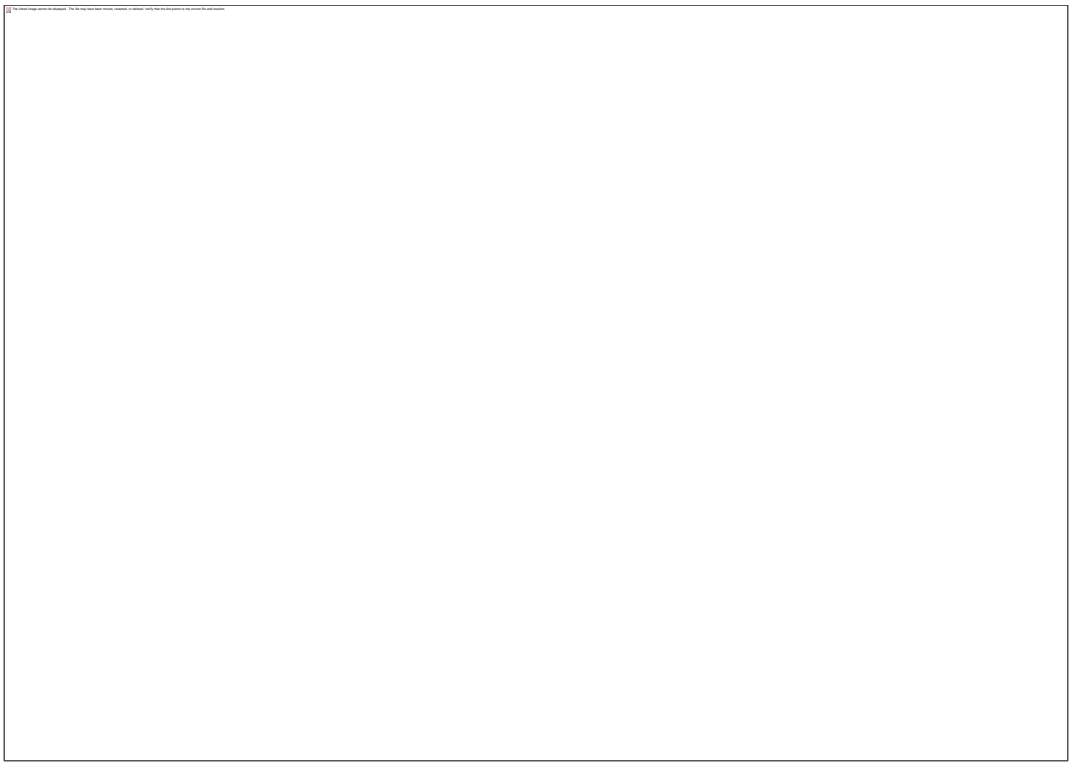
Here is the caller graph for this function:

1.6.3.62 Real norm (const ComplexChebyCoeff & *u*, NormType *n*, bool *normalize*)

References chebyNorm(), L2Norm(), and L2NORM.

```
1608                                                 {
1609   return (n == L2NORM) ? L2Norm(u,normalize) : chebyNorm(u,normalize);
1610 }
```

Here is the call graph for this function:

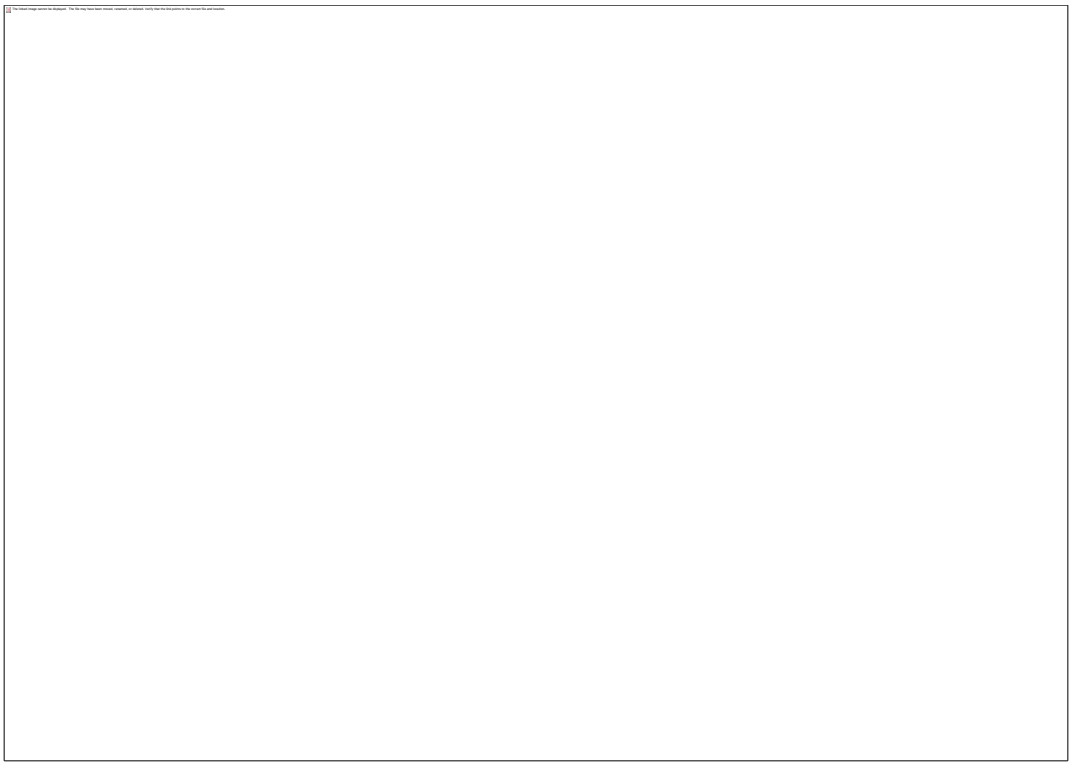


1.6.3.63 Real norm (const ChebyCoeff & *u*, NormType *n*, bool *normalize*)

References chebyNorm(), L2Norm(), and L2NORM.

```
962                                                 {
963   return (n==L2NORM) ? L2Norm(u,normalize) : chebyNorm(u,normalize);
964 }
```

Here is the call graph for this function:

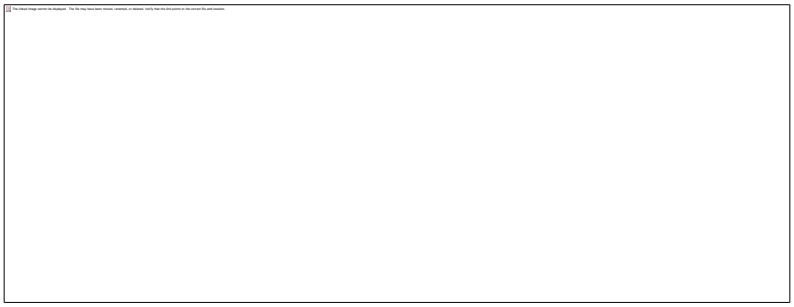


1.6.3.64 Real norm2 (const ComplexChebyCoeff & *u*, NormType *n*, bool *normalize*)

References chebyNorm2(), L2NORM, and L2Norm2().

```
1599                                                {
1600   return (n == L2NORM) ? L2Norm2(u,normalize) : chebyNorm2(u,normalize);
1601 }
```

Here is the call graph for this function:

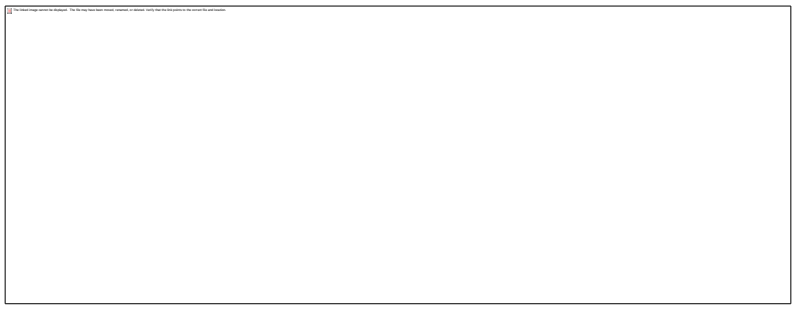


1.6.3.65 Real norm2 (const ChebyCoeff & *u*, NormType *n*, bool *normalize*)

References chebyNorm2(), L2NORM, and L2Norm2().

```
953                                                 {
954   return (n==L2NORM) ? L2Norm2(u,normalize) : chebyNorm2(u,normalize);
955 }
```

Here is the call graph for this function:



1.6.3.66 bool operator!= (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*)

```
1228                                                    {
1229   return !(u==v);
1230 }
```

1.6.3.67 bool operator!= (const ChebyCoeff & *u*, const ChebyCoeff & *v*)

```
620                                                 {
621   return !(u==v);
622 }
```

1.6.3.68 ComplexChebyCoeff operator* (Real *c*, const ComplexChebyCoeff & *v*)

```
1238                                                    {
1239   ComplexChebyCoeff rtn(v);
1240   rtn *= c;
1241   return rtn;
1242 }
```

1.6.3.69 ChebyCoeff operator* (Real *c*, const ChebyCoeff & *v*)

```
580                                                 {
581   ChebyCoeff rtn(v);
582   rtn *= c;
583   return rtn;
```

```
584 }
```

1.6.3.70 ComplexChebyCoeff operator+ (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*)

```
1244                                                    {
1245   ComplexChebyCoeff rtn(u);
1246   rtn += v;
1247   return rtn;
1248 }
```

1.6.3.71 ChebyCoeff operator+ (const ChebyCoeff & *u*, const ChebyCoeff & *v*)

```
599                                                {
600   ChebyCoeff rtn(u);
601   rtn += v;
602   return rtn;
603 }
```

1.6.3.72 ComplexChebyCoeff operator- (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*)

```
1250                                                    {
1251   ComplexChebyCoeff rtn(u);
1252   rtn -= v;
1253   return rtn;
1254 }
```

1.6.3.73 ChebyCoeff operator- (const ChebyCoeff & *u*, const ChebyCoeff & *v*)

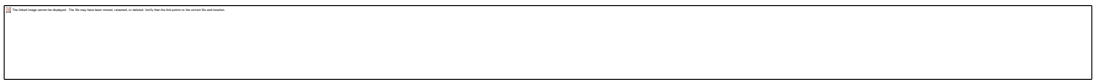
```
605                                                {
606   ChebyCoeff rtn(u);
607   rtn -= v;
608   return rtn;
609 }
```

1.6.3.74 ostream& operator<< (ostream & *os*, const ComplexChebyCoeff & *u*)

References ComplexChebyCoeff::im, ComplexChebyCoeff::length(), and ComplexChebyCoeff::re.

```
1462                                                {
1463   for (int i=0; i<u.length(); ++i)
1464     os << '(' << u.re[i] << ", " << u.im[i] << ")\n";
1465   return os;
1466 }
```

Here is the call graph for this function:



1.6.3.75 bool operator== (const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*)

References ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1225                                                {
1226   return u.re == v.re && u.im == v.im;
1227 }
```

1.6.3.76 bool operator== (const ChebyCoeff & *u*, const ChebyCoeff & *v*)

References ChebyCoeff::congruent(), and ChebyCoeff::numModes().

```
611                                                {
612   if (!u.congruent(v))
613     return false;
614   for (int i=0; i<u.numModes(); ++i)
615     if (u[i] != v[i])
616       return false;
617   return true;
618 }
```

Here is the call graph for this function:

1.6.3.77 Vector periodicpoints (int *N*, Real *L*)

```
763                                {
764   Vector x(N+1);
```

```
765   Real dx=L/N;
766   for (int n=0; n<=N; ++n)
767     x[n] = n*dx;
768   return x;
769 }
```

1.6.3.78 void swap (ComplexChebyCoeff & *f*, ComplexChebyCoeff & *g*)

References ComplexChebyCoeff::im, ComplexChebyCoeff::re, and swap().

```
1232                                        {
1233   swap(f.re,g.re);
1234   swap(f.im,g.im);
1235 }
```

Here is the call graph for this function:

1.6.3.79 void swap (ChebyCoeff & *f*, ChebyCoeff & *g*)

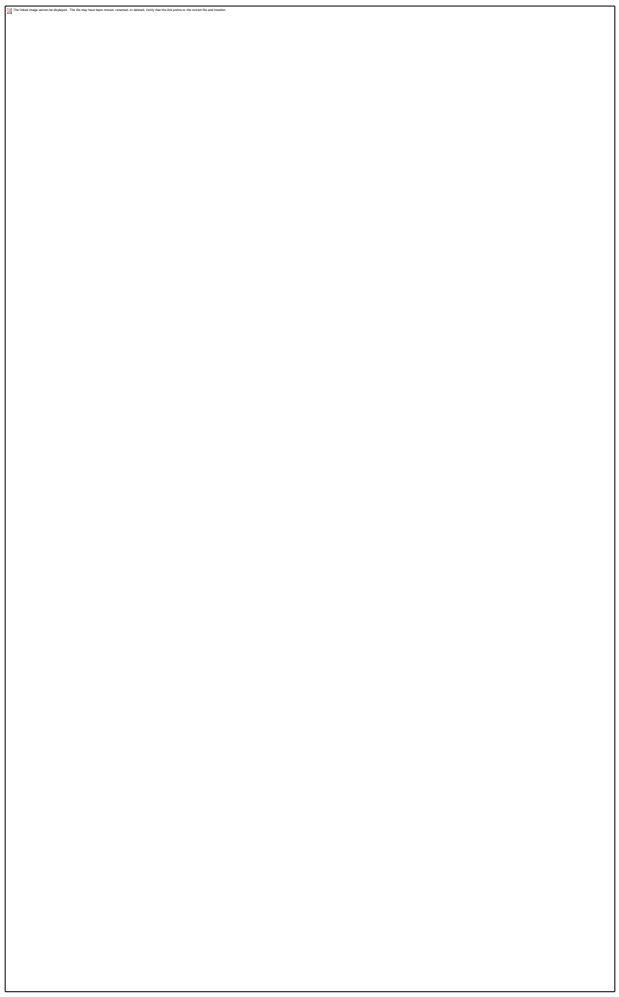
References ChebyCoeff::a_, ChebyCoeff::b_, Vector::data_, Vector::N_, and ChebyCoeff::state_.

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), MultistepDNS::advance(), diff(), FieldSeries::push(), PPEDNS::push(), CNABstyleDNS::push(), MultistepDNS::push(), swap(), and ydiff().

```
624                                     {
625   Real rtmp = f.a_;
626   f.a_ = g.a_;
627   g.a_ = rtmp;
628   rtmp = f.b_;
629   f.b_ = g.b_;
630   g.b_ = rtmp;
631
632   fieldstate stmp = f.state_;
633   f.state_ = g.state_;
634   g.state_ = stmp;
635
636   int itmp = f.N_;
637   f.N_ = g.N_;
638   g.N_ = itmp;
639
640   Real* dtmp = f.data_;
```

```
641   f.data_ = g.data_;
642   g.data_ = dtmp;
643 }
```

Here is the caller graph for this function:



1.6.4 Variable Documentation

1.6.4.1 const Real QUADRATURE_EPSILON = 1e-17

Referenced by gaussLegendreQuadrature().

1.6.4.2

1.7 /_cuda/channelflow-0.9.22/channelflow/chebyshev.h File Reference

#include <fftw3.h>
#include "channelflow/mathdefs.h"
#include "channelflow/vector.h"
Include dependency graph for chebyshev.h:

This graph shows which files directly or indirectly include this file:

1.7.1 Classes

- class ChebyCoeff
- class ComplexChebyCoeff
- class ChebyTransform

1.7.2 Enumerations

- enum NormType { L2NORM, CHEBYNORM }

1.7.3 Functions

- void fftw_loadwisdom (const char *filename=0)
- void fftw_savewisdom (const char *filename=0)
- Vector chebypoints (int N, Real a, Real b)
- Vector periodicpoints (int N, Real L)
- Real chebyIP (int m, int n)
- int cheby_c (int n)

- Real legendre (int n, Real x)
- Real chebyshev (int n, Real x)
- void gaussLegendreQuadrature (int N, Real a, Real b, Vector &x, Vector &w)
- ChebyCoeff operator* (Real c, const ChebyCoeff &g)
- ChebyCoeff operator+ (const ChebyCoeff &f, const ChebyCoeff &g)
- ChebyCoeff operator- (const ChebyCoeff &f, const ChebyCoeff &g)
- bool operator== (const ChebyCoeff &f, const ChebyCoeff &g)
- bool operator!= (const ChebyCoeff &f, const ChebyCoeff &g)
- void diff (const ChebyCoeff &f, ChebyCoeff &df)
- void diff2 (const ChebyCoeff &f, ChebyCoeff &d2f)
- void diff2 (const ChebyCoeff &f, ChebyCoeff &d2f, ChebyCoeff &tmp)
- void diff (const ChebyCoeff &f, ChebyCoeff &df, int n)
- ChebyCoeff diff (const ChebyCoeff &f)
- ChebyCoeff diff2 (const ChebyCoeff &f)
- ChebyCoeff diff (const ChebyCoeff &f, int n)
- void integrate (const ChebyCoeff &df, ChebyCoeff &f)
- ChebyCoeff integrate (const ChebyCoeff &df)
- ChebyCoeff legendre (int N, int n, bool normalize=false)
- ChebyCoeff chebyshev (int N, int n, bool normalize=false)
- Real L2Norm2 (const ChebyCoeff &f, bool normalize=true)
- Real L2Dist2 (const ChebyCoeff &f, const ChebyCoeff &g, bool normalize=true)
- Real L2Norm (const ChebyCoeff &f, bool normalize=true)
- Real L2Dist (const ChebyCoeff &f, const ChebyCoeff &g, bool normalize=true)
- Real L2InnerProduct (const ChebyCoeff &f, const ChebyCoeff &g, bool normalize=true)
- Real chebyNorm2 (const ChebyCoeff &f, bool normalize=true)
- Real chebyDist2 (const ChebyCoeff &f, const ChebyCoeff &g, bool normalize=true)
- Real chebyNorm (const ChebyCoeff &f, bool normalize=true)
- Real chebyDist (const ChebyCoeff &f, const ChebyCoeff &g, bool normalize=true)
- Real chebyInnerProduct (const ChebyCoeff &f, const ChebyCoeff &g, bool normalize=true)
- Real norm2 (const ChebyCoeff &f, NormType n, bool normalize=true)
- Real dist2 (const ChebyCoeff &f, NormType n, const ChebyCoeff &g, bool normalize=true)
- Real norm (const ChebyCoeff &f, NormType n, bool normalize=true)
- Real dist (const ChebyCoeff &f, const ChebyCoeff &g, NormType n, bool normalize=true)
- Real innerProduct (const ChebyCoeff &f, const ChebyCoeff &g, NormType n, bool normalize=true)
- Real L1Norm (const ChebyCoeff &f, bool normalize=true)
- Real L1Dist (const ChebyCoeff &f, const ChebyCoeff &g, bool normalize=true)
- Real LinfNorm (const ChebyCoeff &f)
- Real LinfDist (const ChebyCoeff &f, const ChebyCoeff &g)
- ComplexChebyCoeff operator* (Real c, const ComplexChebyCoeff &g)
- ComplexChebyCoeff operator+ (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g)
- ComplexChebyCoeff operator- (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g)

- bool operator== (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g)
- bool operator!= (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g)
- void diff (const ComplexChebyCoeff &f, ComplexChebyCoeff &df)
- void diff2 (const ComplexChebyCoeff &f, ComplexChebyCoeff &d2f)
- void diff2 (const ComplexChebyCoeff &f, ComplexChebyCoeff &d2f, ComplexChebyCoeff &tmp)
- void diff (const ComplexChebyCoeff &f, ComplexChebyCoeff &d2f, int n)
- ComplexChebyCoeff diff (const ComplexChebyCoeff &f)
- ComplexChebyCoeff diff2 (const ComplexChebyCoeff &f)
- ComplexChebyCoeff diff (const ComplexChebyCoeff &f, int n)
- void integrate (const ComplexChebyCoeff &df, ComplexChebyCoeff &f)
- ComplexChebyCoeff integrate (const ComplexChebyCoeff &df)
- std::ostream & operator<< (std::ostream &os, const ComplexChebyCoeff &f)
- ChebyCoeff & Re (ComplexChebyCoeff &f)
- ChebyCoeff & Im (ComplexChebyCoeff &f)
- const ChebyCoeff & Re (const ComplexChebyCoeff &f)
- const ChebyCoeff & Im (const ComplexChebyCoeff &f)
- Real L2Norm2 (const ComplexChebyCoeff &f, bool normalize=true)
- Real L2Dist2 (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g, bool normalize=true)
- Real L2Norm (const ComplexChebyCoeff &f, bool normalize=true)
- Real L2Dist (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g, bool normalize=true)
- Complex L2InnerProduct (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g, bool normalize=true)
- Real chebyNorm2 (const ComplexChebyCoeff &f, bool normalize=true)
- Real chebyDist2 (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g, bool normalize=true)
- Real chebyNorm (const ComplexChebyCoeff &f, bool normalize=true)
- Real chebyDist (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g, bool normalize=true)
- Complex chebyInnerProduct (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g, bool normalize=true)
- Real norm2 (const ComplexChebyCoeff &f, NormType n, bool normalize=true)
- Real dist2 (const ComplexChebyCoeff &f, NormType n, const ChebyCoeff &g, bool normalize=true)
- Real norm (const ComplexChebyCoeff &f, NormType n, bool normalize=true)
- Real dist (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g, NormType n, bool normalize=true)
- Complex innerProduct (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g, NormType n, bool normalize=true)
- Real L1Norm (const ComplexChebyCoeff &f, bool normalize=true)
- Real L1Dist (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g, bool normalize=true)

- Real LinfNorm (const ComplexChebyCoeff &f)
- Real LinfDist (const ComplexChebyCoeff &f, const ComplexChebyCoeff &g)
- ComplexChebyCoeff chebyEval (const ComplexChebyCoeff &c, const Vector &x)

1.7.4 Enumeration Type Documentation

1.7.4.1 enum NormType

Enumerator:

L2NORM
CHEBYNORM

```
35 {L2NORM, CHEBYNORM};
```

1.7.5 Function Documentation

1.7.5.1 int cheby_c (int *n*) [inline]

Referenced by chebyshev().

```
44 {return (n>0) ? 1 : ((n==0) ? 2 : 0);}
```

Here is the caller graph for this function:

1.7.5.2 Real chebyDist (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*, bool *normalize* = true)

References chebyDist2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1565                        {
1566   return sqrt(chebyDist2(u.re, v.re, normalize) + chebyDist2(u.im, v.im, normalize));
1567 }
```

Here is the call graph for this function:

1.7.5.3 Real chebyDist (const ChebyCoeff & *f*, const ChebyCoeff & *g*, bool *normalize* = true)

References chebyDist2().

```
858                                                 {
859   return sqrt(chebyDist2(u,v, normalize));
860 }
```

Here is the call graph for this function:

1.7.5.4 Real chebyDist2 (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*, bool *normalize* = true)

References chebyDist2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1556                                {
1557   return chebyDist2(u.re, v.re, normalize) + chebyDist2(u.im, v.im, normalize);
1558 }
```

Here is the call graph for this function:

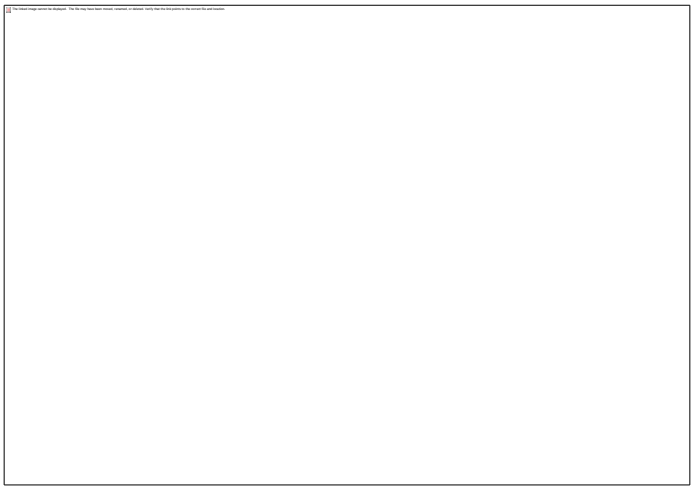
1.7.5.5 Real chebyDist2 (const ChebyCoeff & *f*, const ChebyCoeff & *g*, bool *normalize* = true)

References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::numModes(), pi, Spectral, square(), and ChebyCoeff::state().

```
839                                                {
840   assert(u.state()==Spectral && v.state()==Spectral);
841   assert(u.a()==v.a() && u.b()==v.b());
842   assert(u.numModes() ==  v.numModes());
843   int N=u.numModes();
844   Real sum=0.0;
845   for (int m=N-1; m>0; --m)
846     sum += square(u[m]-v[m]);
847   if (N>0)
848     sum += 2*square(u[0]-v[0]);  // coeff of T_0(x) has prefactor of 2 in norm
849   if (!normalize)
850     sum *= u.b()-u.a();
```

```
851   return sum*pi/2;
852 }
```

Here is the call graph for this function:



1.7.5.6 ComplexChebyCoeff chebyEval (const ComplexChebyCoeff & *c*, const Vector & *x*)

1.7.5.7 Complex chebyInnerProduct (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*, bool *normalize* = true)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), ComplexChebyCoeff::im, ComplexChebyCoeff::numModes(), pi, ComplexChebyCoeff::re, Spectral, and ComplexChebyCoeff::state().

```
1571                                    {
1572   assert(u.state()==Spectral && v.state()==Spectral);
1573   assert(u.numModes() == v.numModes());
1574   int N = u.numModes();
1575
1576   // Unroll the complex operations, for efficiency.
1577   // Profiling an L2InnerProduct-intensive program reveals that
1578   // complex * double is expensive
1579   Real sum_r = 0.0;
1580   Real sum_i = 0.0;
1581   for (int m=N-1; m>0; --m) {
1582     sum_r += u.re[m]*v.re[m] + u.im[m]*v.im[m];
1583     sum_i += u.im[m]*v.re[m] - u.re[m]*v.im[m];
1584   }
1585   if (N>0) {
1586     sum_r += 2*(u.re[0]*v.re[0] + u.im[0]*v.im[0]);
1587     sum_i += 2*(u.im[0]*v.re[0] - u.re[0]*v.im[0]);
```

```
1588   }
1589   if (!normalize) {
1590     sum_r *= u.b()-u.a();
1591     sum_i *= u.b()-u.a();
1592   }
1593   return Complex(sum_r*pi/2, sum_i*pi/2);
1594 }
```

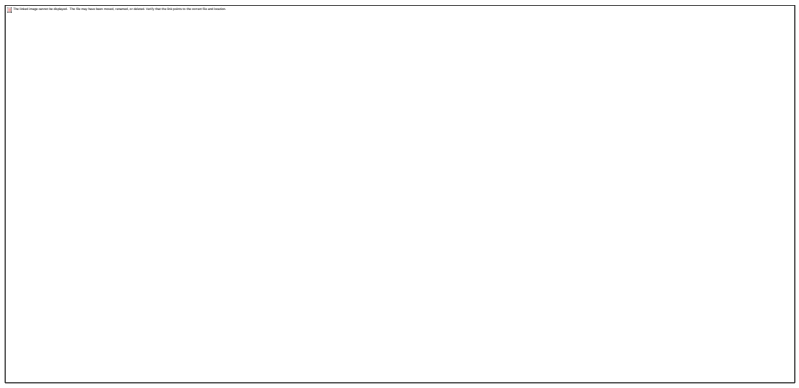
Here is the call graph for this function:

1.7.5.8 Real chebyInnerProduct (const ChebyCoeff & *f*, const ChebyCoeff & *g*, bool *normalize* = true)

References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::numModes(), pi, Spectral, and ChebyCoeff::state().

```
862                                                    {
863   assert(u.state()==Spectral && v.state()==Spectral);
864   assert(u.a()==v.a() && u.b()==v.b());
865   assert(u.numModes() ==  v.numModes());
866   int N=u.numModes();
867   Real sum=0.0;
868   for (int m=N-1; m>0; --m)
869     sum += u[m]*v[m];
870   if (N>0)
871     sum += 2*u[0]*v[0];  // coeff of T_0(x) has prefactor of 2 in norm
872   if (!normalize)
873     sum *= u.b()-u.a();
874   return sum*pi/2;
875 }
```

Here is the call graph for this function:



1.7.5.9 Real chebyIP (int *m*, int *n*)

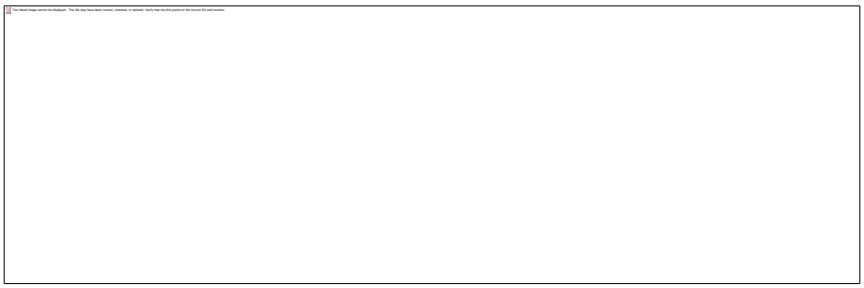
```
46                          {
47   return ((m+n) % 2 == 1) ? 0 :
48     (1.0-m*m-n*n)/((1.0+m-n)*(1.0-m+n)*(1.0+m+n)*(1.0-m-n));
49 }
```

1.7.5.10 Real chebyNorm (const ComplexChebyCoeff & *f*, bool *normalize* = true)

References chebyNorm2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1560                                                {
1561   return sqrt(chebyNorm2(u.re, normalize) + chebyNorm2(u.im, normalize));
1562 }
```

Here is the call graph for this function:

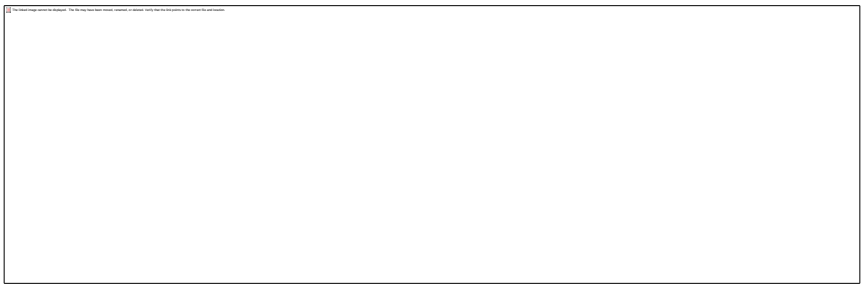


1.7.5.11 Real chebyNorm (const ChebyCoeff & *f*, bool *normalize* = true)

References chebyNorm2().

```
854                                         {
855   return sqrt(chebyNorm2(u, normalize));
856 }
```

Here is the call graph for this function:



1.7.5.12 Real chebyNorm2 (const ComplexChebyCoeff & *f*, bool *normalize* = true)

References chebyNorm2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1551                                                {
1552   return chebyNorm2(u.re, normalize) + chebyNorm2(u.im, normalize);
1553 }
```

Here is the call graph for this function:

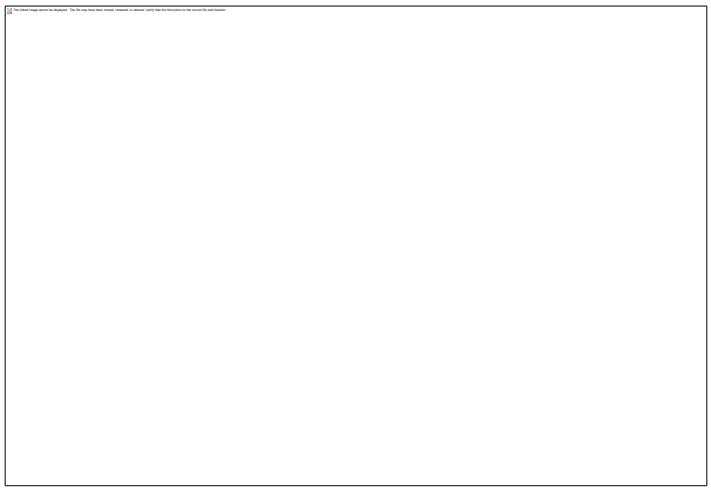
1.7.5.13 Real chebyNorm2 (const ChebyCoeff & *f*, bool *normalize* = true)

References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::numModes(), pi, Spectral, square(), and ChebyCoeff::state().

```
825                                                {
826   assert(u.state() == Spectral);
827   int N=u.numModes();
828   Real sum=0.0;
829   for (int m=N-1; m>0; --m)
830     sum += square(u[m]);
831   if (N>0)
832     sum += 2*square(u[0]);  // coeff of T_0(x) has prefactor of 2 in norm
833   if (!normalize)
834     sum *= u.b()-u.a();
```

```
835   sum *= pi/2;
836   return sum;
837 }
```

Here is the call graph for this function:



1.7.5.14 Vector chebypoints (int *N*, Real *a*, Real *b*)

References pi.

Referenced by FlowField::CFLfactor(), makemeen(), makeprtb(), makeroll(), makewave(), makewobl(), and TurbStats::yplus().

```
754                                 {
755   Vector xcheb(N);
756   Real piN = pi/(N-1);
757   Real radius = (b-a)/2;
758   Real center = (b+a)/2;
759   for (int j=0; j<N; ++j)
760     xcheb[j] = center + radius*cos(j*piN);
761   return xcheb;
762 }
```

Here is the caller graph for this function:

1.7.5.15 ChebyCoeff chebyshev (int *N*, int *n*, bool *normalize* = false)

References cheby_c(), pi, and Spectral.

```
79                                          {
80   assert(n<N);
81   ChebyCoeff Tn(N, -1 , 1, Spectral);
82   if (!normalize)
83     Tn[n] = 1;
84   else
85     Tn[n] = sqrt(2/(pi*cheby_c(n)));
86   return Tn;
87 }
```

Here is the call graph for this function:

1.7.5.16 Real chebyshev (int *n*, Real *x*)

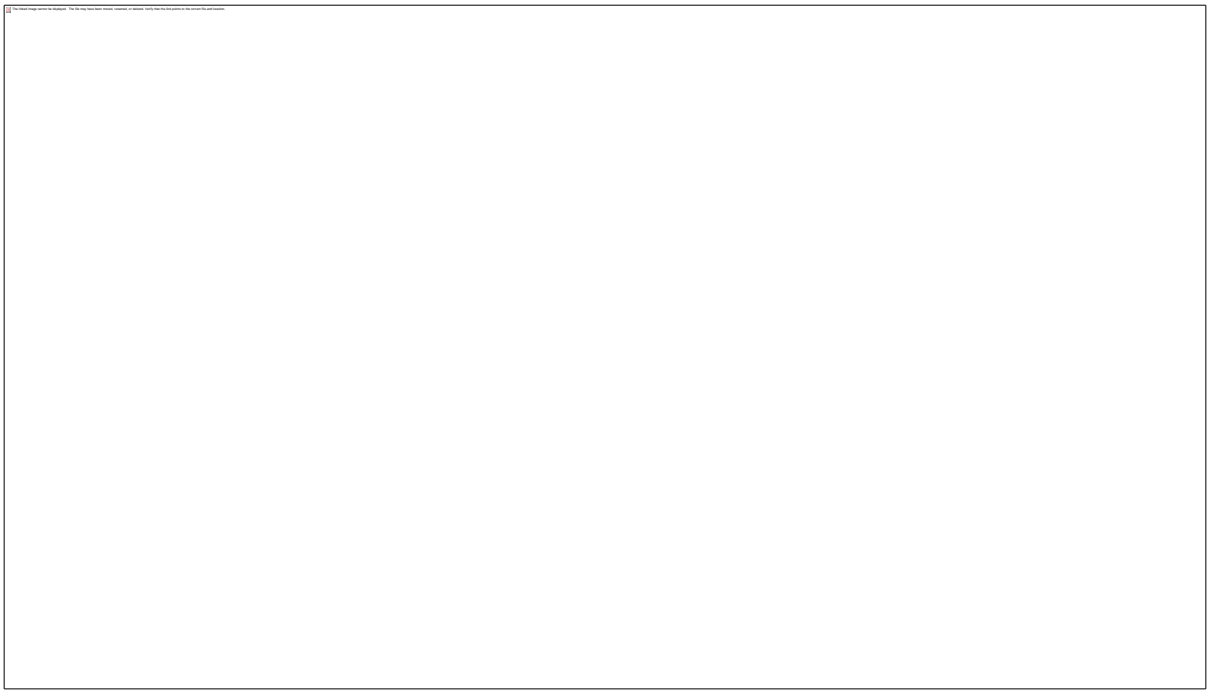
```
62                          {
63   return cos(n*acos(x));
64 }
```

1.7.5.17 ComplexChebyCoeff diff (const ComplexChebyCoeff & *f*, int *n*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff(), ComplexChebyCoeff::im, ComplexChebyCoeff::numModes(), ComplexChebyCoeff::re, and Spectral.

```
1286                                                {
1287   ComplexChebyCoeff du(u.numModes(), u.a(), u.b(), Spectral);
1288   diff(u.re, du.re, n);
1289   diff(u.im, du.im, n);
1290   return du;
1291 }
```

Here is the call graph for this function:



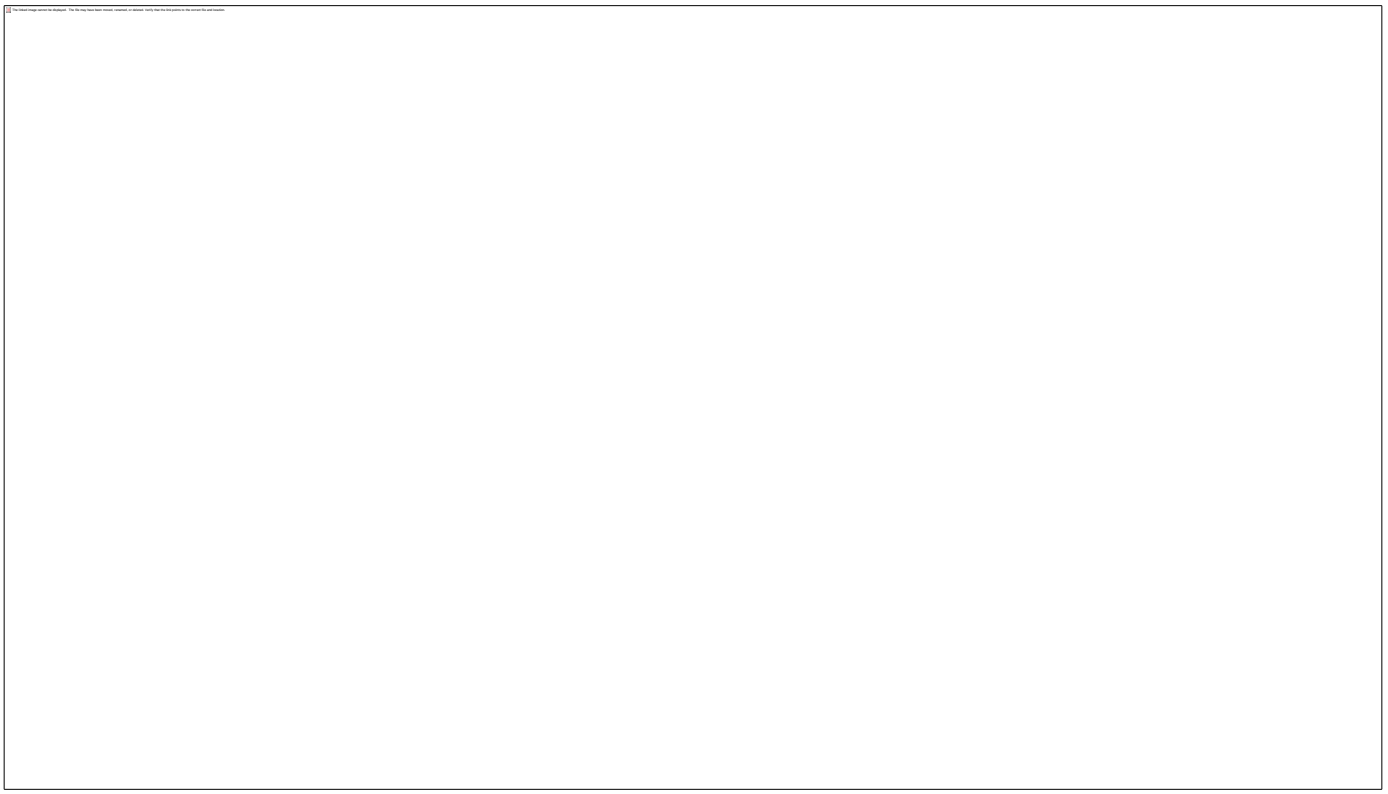
1.7.5.18 ComplexChebyCoeff diff (const ComplexChebyCoeff & *f*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff(), ComplexChebyCoeff::im, ComplexChebyCoeff::numModes(), ComplexChebyCoeff::re, and Spectral.

```
1274                                {
```

```
1275   ComplexChebyCoeff du(u.numModes(), u.a(), u.b(), Spectral);
1276   diff(u.re, du.re);
1277   diff(u.im, du.im);
1278   return du;
1279 }
```

Here is the call graph for this function:

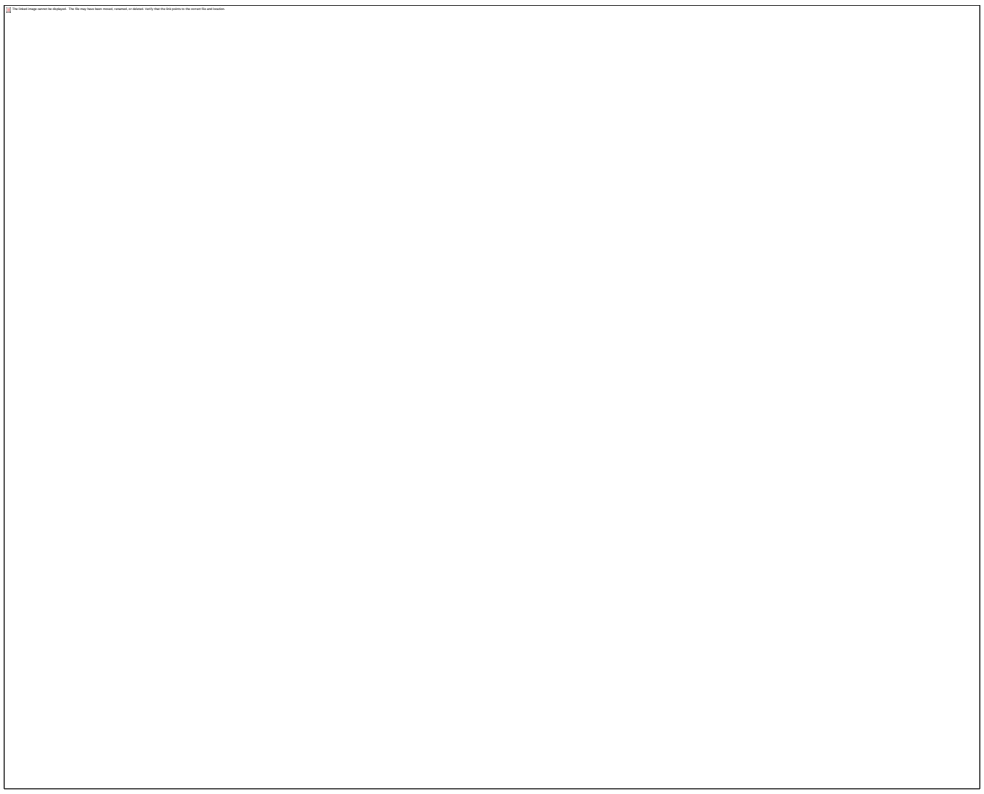


1.7.5.19 void diff (const ComplexChebyCoeff & *f*, ComplexChebyCoeff & *d2f*, int *n*)

References diff(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1270                                                    {
1271   diff(u.re, du.re, n);
1272   diff(u.im, du.im, n);
1273 }
```

Here is the call graph for this function:

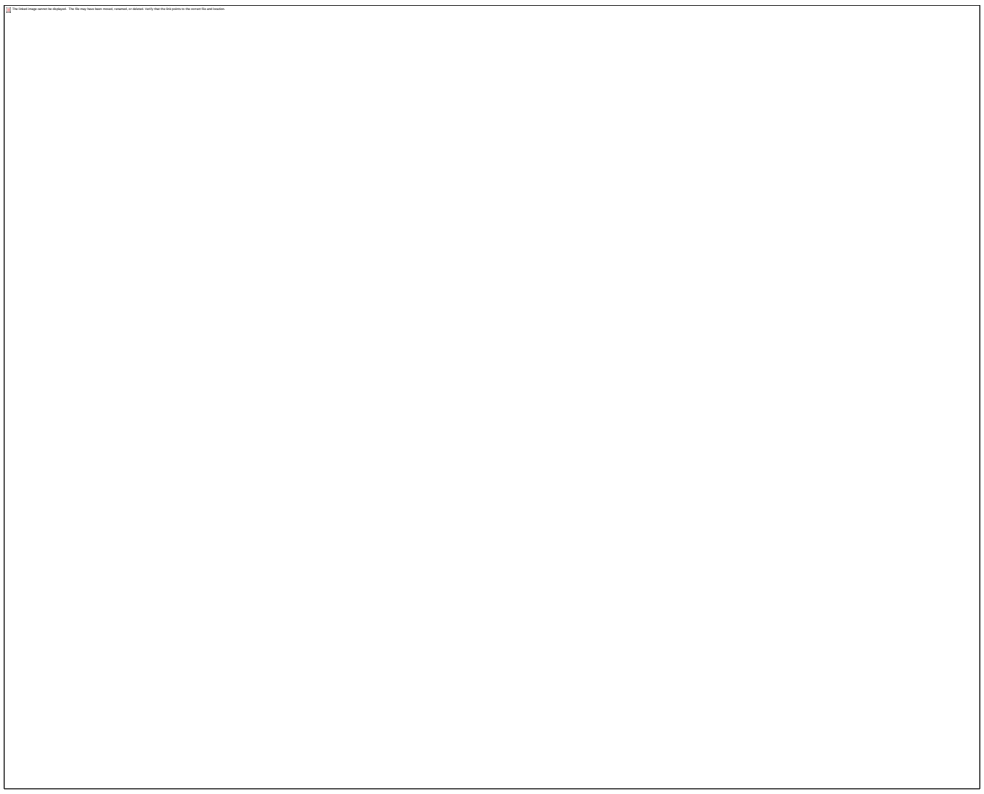


1.7.5.20 void diff (const ComplexChebyCoeff & *f*, ComplexChebyCoeff & *df*)

References diff(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1256                                                {
1257   diff(u.re, du.re);
1258   diff(u.im, du.im);
1259 }
```

Here is the call graph for this function:

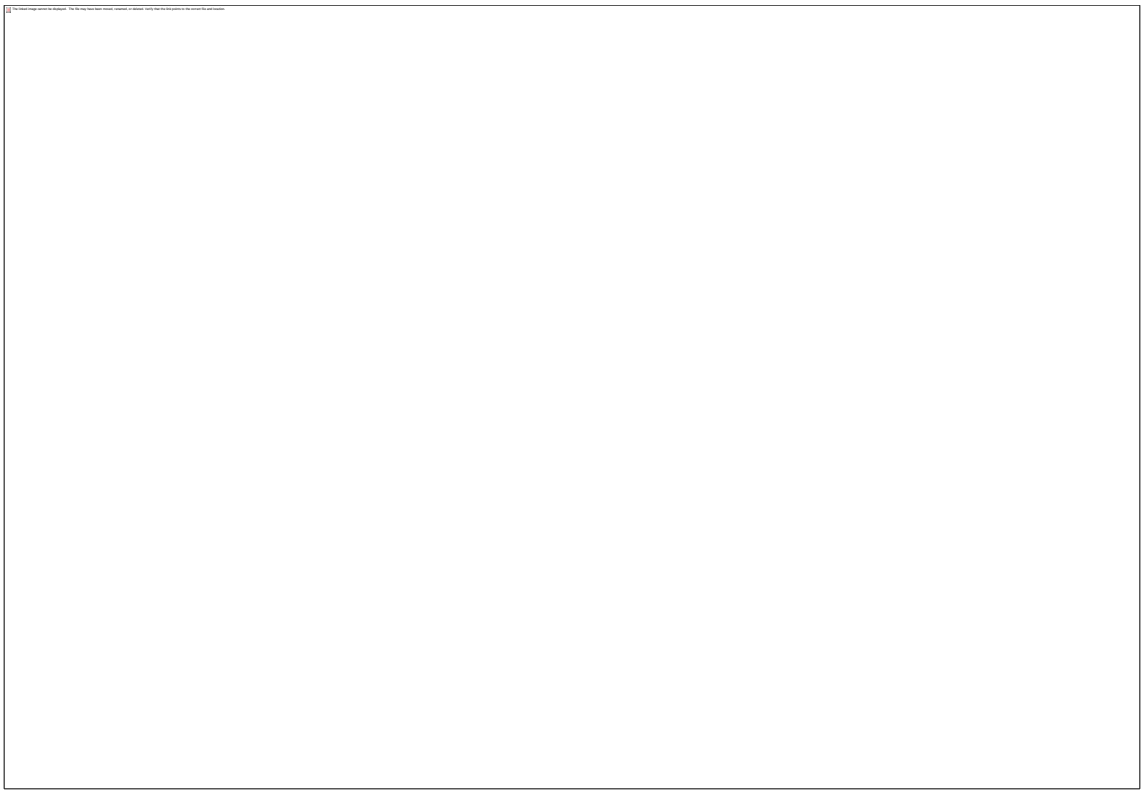


1.7.5.21 ChebyCoeff diff (const ChebyCoeff & *f*, int *n*)

References ChebyCoeff::a(), ChebyCoeff::b(), diff(), ChebyCoeff::N(), Spectral, and swap().

```
742                                     {
743   assert(n>=0);
744   ChebyCoeff  g = f;
745   ChebyCoeff gy(f.N(), f.a(), f.b(), Spectral);
746
747   for (int n_=0; n_<n; ++n_) {
748     diff(g,gy);
749     swap(g,gy);
750   }
751   return g;
752 }
```

Here is the call graph for this function:

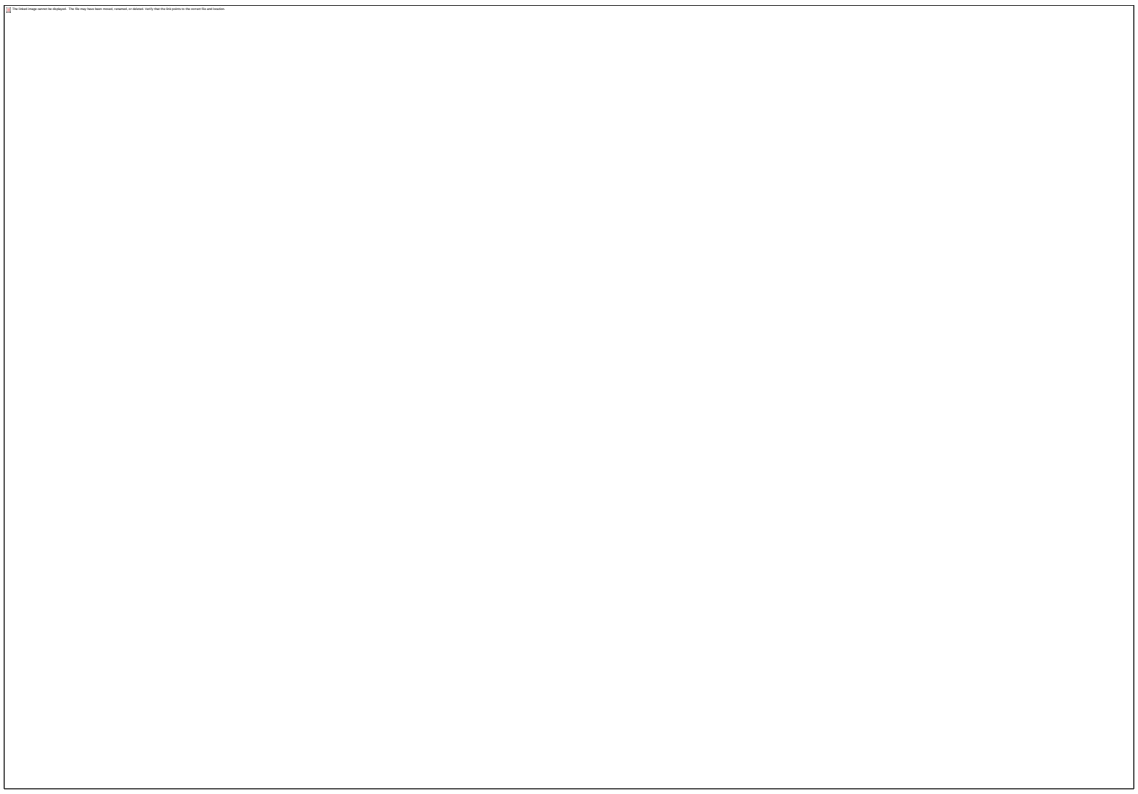


1.7.5.22 ChebyCoeff diff (const ChebyCoeff & *f*)

References ChebyCoeff::a(), ChebyCoeff::b(), diff(), Vector::length(), and Spectral.

```
730                                 {
731   ChebyCoeff du(u.length(), u.a(), u.b(), Spectral);
732   diff(u, du);
733   return du;
734 }
```

Here is the call graph for this function:

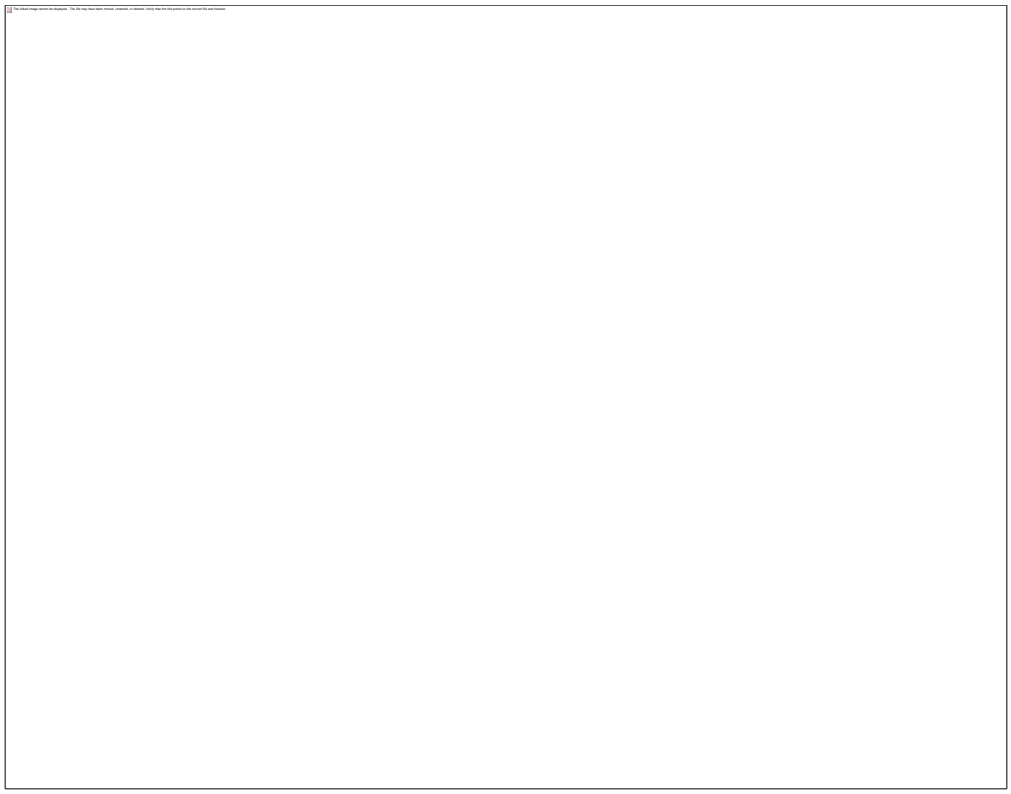


1.7.5.23 void diff (const ChebyCoeff & *f*, ChebyCoeff & *df*, int *n*)

References diff(), Spectral, ChebyCoeff::state(), and swap().

```
718                                          {
719   assert(n>=0);
720   assert(f.state() == Spectral);
721   df = f;
722   ChebyCoeff tmp;
723   for (int k=0; k<n; ++k) {
724     diff(df,tmp);
725     swap(df,tmp);
726   }
727   return;
728 }
```

Here is the call graph for this function:



1.7.5.24 void diff (const ChebyCoeff & *f*, ChebyCoeff & *df*)

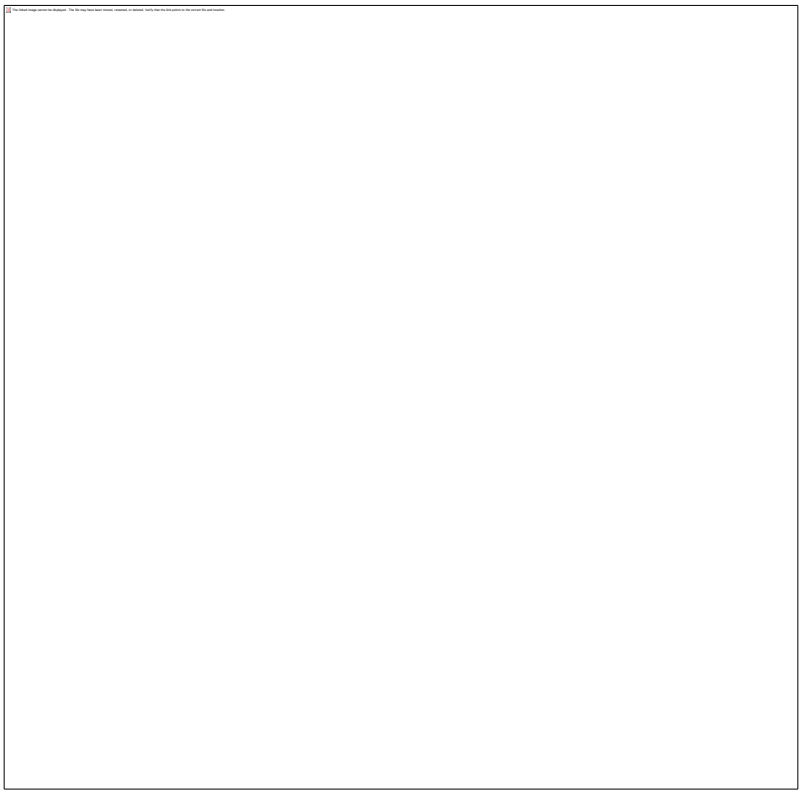
References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::L(), ChebyCoeff::numModes(), Vector::resize(), ChebyCoeff::setBounds(), ChebyCoeff::setState(), Spectral, and ChebyCoeff::state().

Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), bcDist2(), chebyProfile(), chebyVprofile(), curl(), diff(), diff2(), div(), divcheck(), divDist2(), divergence(), divNorm2(), DNSAlgorithm::DNSAlgorithm(), FlowField::dudy_a(), FlowField::dudy_b(), grad(), gradient(), TauSolver::influenceCorrection(), linearizedNL(), makeprtb(), makeroll(), makewave(), makewobl(), BasisFunc::randomize(), randomProfile(), randomVprofile(), rotationalNL(), skewsymmetricNL_task4::Run(), FlowField::saveDivSpectrum(), skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), TauSolver::solve(), PressureSolver::solve(), TauSolver::solve_P_and_v(), TauSolver::TauSolver(), TurbStats::ustar(), vbcFix(), TauSolver::verify(), PoissonSolver::verify(), TauSolver::verify_P_and_v(), and ydiff().

```
683                                     {
684   assert(u.state() == Spectral);
685   if (dudy.numModes() != u.numModes())
686     dudy.resize(u.numModes());
687   dudy.setBounds(u.a(), u.b());
688   dudy.setState(Spectral);
689
```

```
690   int Nb = u.numModes() - 1;
691   if (Nb==-1)
692     return;
693   if (Nb==0) {
694     dudy[0] = 0.0;
695     return;
696   }
697   Real scale=4.0/u.L();
698   dudy[Nb] = 0.0;
699   dudy[Nb-1] = scale*Nb*u[Nb];
700   for (int n=Nb-2; n>=0; --n)
701     dudy[n] = dudy[n+2] + scale*(n+1)*u[n+1];
702   dudy[0] /= 2.0;
703
704   //dudy *= 2.0/u.L();
705 }
```

Here is the call graph for this function:



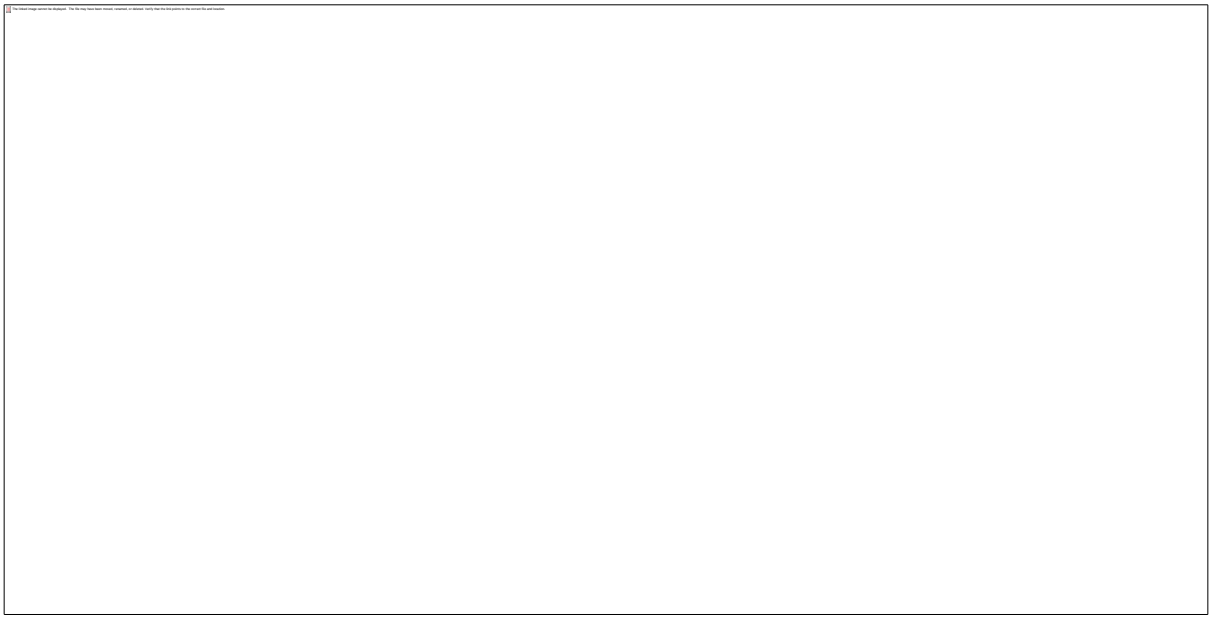
Here is the caller graph for this function:

1.7.5.25 ComplexChebyCoeff diff2 (const ComplexChebyCoeff & *f*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff2(), ComplexChebyCoeff::im, ComplexChebyCoeff::numModes(), ComplexChebyCoeff::re, and Spectral.

```
1280                                                {
1281   ComplexChebyCoeff du2(u.numModes(), u.a(), u.b(), Spectral);
1282   diff2(u.re, du2.re);
1283   diff2(u.im, du2.im);
1284   return du2;
1285 }
```

Here is the call graph for this function:



1.7.5.26 void diff2 (const ComplexChebyCoeff & *f*, ComplexChebyCoeff & *d2f*, ComplexChebyCoeff & *tmp*)

References diff2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1265                                                                {
1266   diff2(u.re, d2u.re, tmp.re);
1267   diff2(u.im, d2u.im, tmp.im);
1268 }
```

Here is the call graph for this function:

1.7.5.27 void diff2 (const ComplexChebyCoeff & *f*, ComplexChebyCoeff & *d2f*)

References diff2(), ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

```
1261                                                    {
1262   diff2(u.re, d2u.re);
1263   diff2(u.im, d2u.im);
1264 }
```

Here is the call graph for this function:

1.7.5.28 ChebyCoeff diff2 (const ChebyCoeff & *f*)

References ChebyCoeff::a(), ChebyCoeff::b(), diff2(), Vector::length(), and Spectral.

```
736                                     {
737   ChebyCoeff du2(u.length(), u.a(), u.b(), Spectral);
738   diff2(u, du2);
739   return du2;
740 }
```

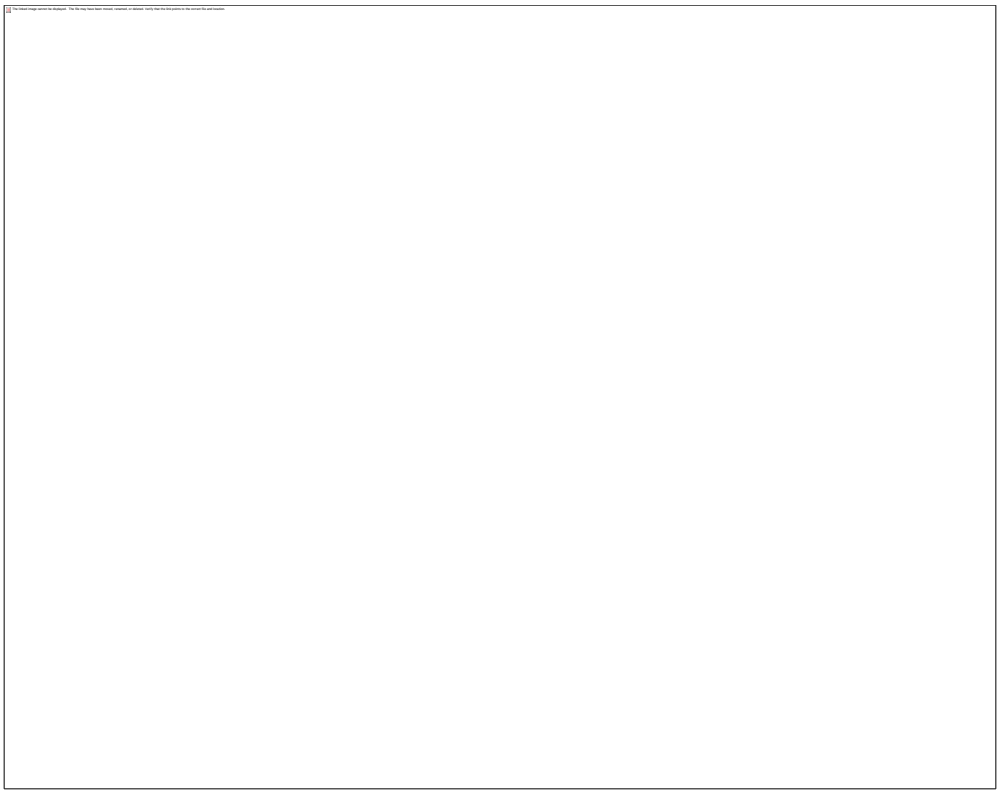
Here is the call graph for this function:

1.7.5.29 void diff2 (const ChebyCoeff & *f*, ChebyCoeff & *d2f*, ChebyCoeff & *tmp*)

References diff().

```
713                                                 {
714   diff(u, tmp);
715   diff(tmp, d2udy2);
716 }
```

Here is the call graph for this function:



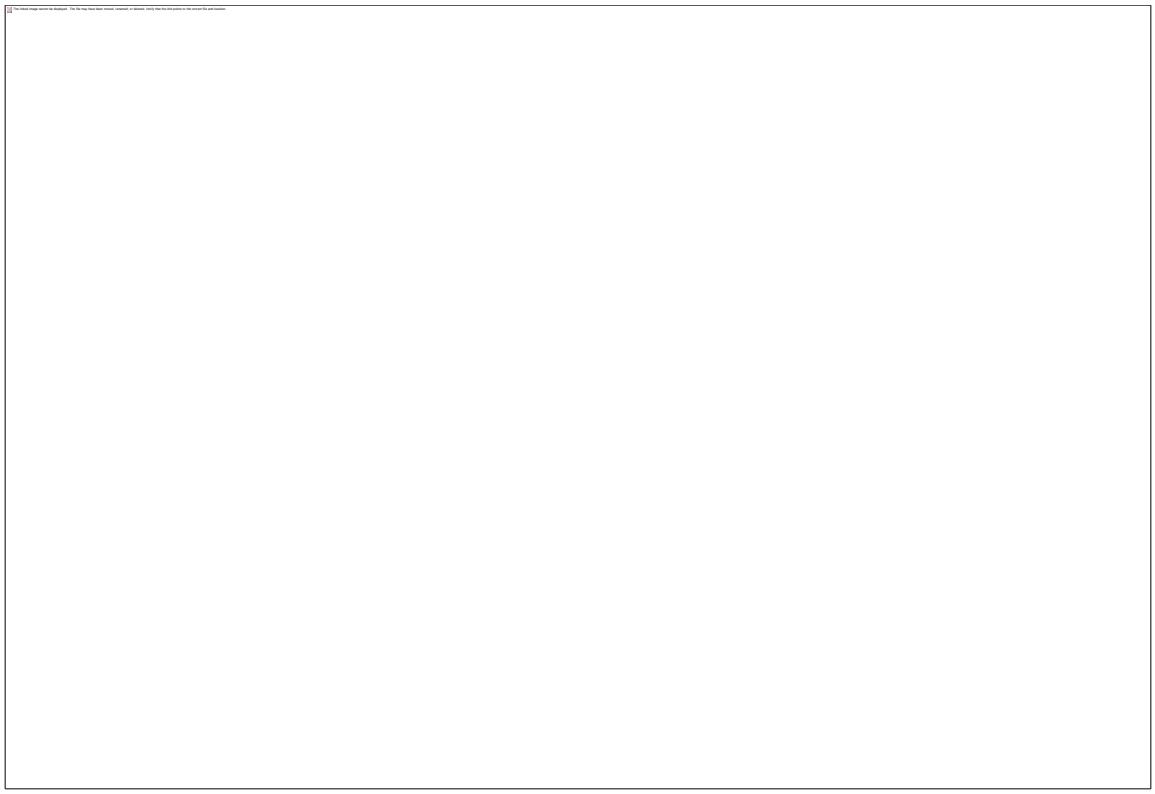
1.7.5.30 void diff2 (const ChebyCoeff & *f*, ChebyCoeff & *d2f*)

References ChebyCoeff::a(), ChebyCoeff::b(), diff(), Vector::length(), and Spectral.

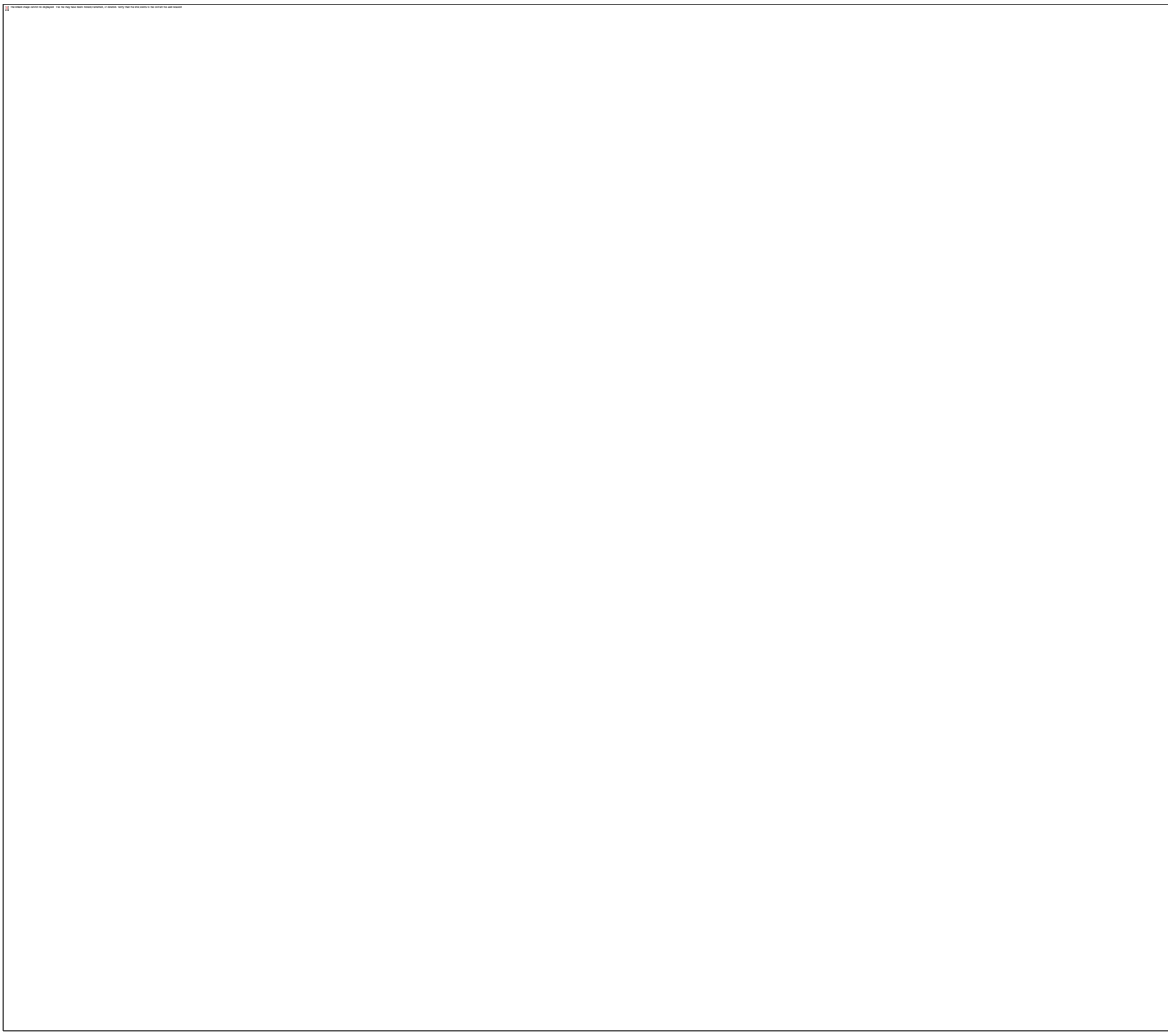
Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), diff2(), lapl(), laplacian(), HelmholtzSolver::residual(), PressureSolver::solve(), TauSolver::solve_P_and_v(), TauSolver::TauSolver(), TauSolver::verify(), and HelmholtzSolver::verify().

```
707                                         {
708   ChebyCoeff dudy(u.length(), u.a(), u.b(), Spectral);
709   diff(u, dudy);
710   diff(dudy, d2udy2);
711 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.7.5.31 Real dist (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*, NormType *n*, bool *normalize* = true)

References chebyDist(), L2Dist(), and L2NORM.

```
1613                                {
1614   return (n == L2NORM) ? L2Dist(u,v,normalize) : chebyDist(u,v,normalize);
1615 }
```

Here is the call graph for this function:

1.7.5.32 Real dist (const ChebyCoeff & *f*, const ChebyCoeff & *g*, NormType *n*, bool *normalize* = true)

References chebyDist(), L2Dist(), and L2NORM.

```
966                     {
967   return (n==L2NORM) ? L2Dist(u,v,normalize) : chebyDist(u,v,normalize);
968 }
```

Here is the call graph for this function:

1.7.5.33 Real dist2 (const ComplexChebyCoeff & *f*, NormType *n*, const ChebyCoeff & *g*, bool *normalize* = true)

1.7.5.34 Real dist2 (const ChebyCoeff & *f*, NormType *n*, const ChebyCoeff & *g*, bool *normalize* = true)

1.7.5.35 void fftw_loadwisdom (const char * *filename* = 0)

```
122                                     {
123
124 #ifdef HAVE_WORDEXP_H
125   //IG const char* filename = (filename_ ==0) ? "~/.fftw_wisdom" : filename_;
126   const char* filename = (filename_ ==0) ? ".fftw_wisdom" : filename_;
127
128   // This code converts filename to a full pathname.
129   wordexp_t w;
130   w.we_offs=0;
131   int flags = 0;
132   wordexp(filename, &w, flags);
133   if (w.we_wordc > 0)
134     filename = w.we_wordv[0];
135 #else
136   const char* filename = filename_;
137 #endif
138
139   // Good old byzantine C IO.
140   FILE* input_file = fopen(filename, "r");
141   if (input_file == NULL || !fftw_import_wisdom_from_file(input_file))
142     cerr << "Error reading fftw-wisdom file " << filename << ", proceding without fftw-
wisdom.\n";
143   else
144     fclose(input_file);
145 }
```

1.7.5.36 void fftw_savewisdom (const char * *filename* = 0)

```
146                                     {
147
148 #ifdef HAVE_WORDEXP_H
149   //IG const char* filename = (filename_ ==0) ? "~/.fftw_wisdom" : filename_;
150   const char* filename = (filename_ ==0) ? ".fftw_wisdom" : filename_;
151   wordexp_t w;
152   w.we_offs=0;
153   int flags = 0;
154   wordexp(filename, &w, flags);
```

```
155   if (w.we_wordc > 0)
156     filename = w.we_wordv[0];
157 #else
158   const char* filename = filename_;
159 #endif
160
161   FILE* output_file;
162   output_file = fopen(filename, "w");
163   if (output_file ==0)
164     cerr << "Can't write to fftw-wisdom file " << filename << "\n";
165
166   fftw_export_wisdom_to_file(output_file);
167   fclose(output_file);
168 }
```

1.7.5.37 void gaussLegendreQuadrature (int *N*, Real *a*, Real *b*, Vector & *x*, Vector & *w*)

References M_PI, and QUADRATURE_EPSILON.

```
89                                                  {
90   x = Vector(N);
91   w = Vector(N);
92   int M = (N+1)/2;
93   Real middle = 0.5*(b+a);
94   Real radius = 0.5*(b-a);
95   Real pp;
96   Real z,z1;
97
98   for (int m=0; m<M; ++m) {
99     z = cos(M_PI*(m+0.75)/(N+0.5));
100     Real p1,p2,p3;
101     int iterations=0;
102     do {
103       p1 = 1.0;
104       p2 = 0.0;
105       for (int n=0; n<N; ++n) {
106         p3 = p2;
107         p2 = p1;
108         p1 = ((2*n+1)*z*p2-n*p3)/(n+1);
109       }
110       pp = N*(z*p1-p2)/(z*z-1.0);
111       z1=z;
112       z=z1-p1/pp;
113     } while (fabs(z-z1) > QUADRATURE_EPSILON && ++iterations < 1024);
114
```

```
115     x(m) = middle - radius*z;
116     x(N-1-m) = middle + radius*z;
117     w(m) = 2.0*radius/((1.0-z*z)*pp*pp);
118     w(N-1-m) = w(m);
119   }
120 }
```

1.7.5.38 const ChebyCoeff & Im (const ComplexChebyCoeff & *f*) [inline]

References ComplexChebyCoeff::im.

```
359 {return f.im;}
```

1.7.5.39 ChebyCoeff & Im (ComplexChebyCoeff & *f*) [inline]

References ComplexChebyCoeff::im.

Referenced by ComplexChebyCoeff::add(), FlowField::asciiSave(), exp(), FieldSeries::interpolate(), ComplexChebyCoeff::operator*=(), operator<<(), save(), BasisFunc::save(), FlowField::saveProfile(), FlowField::saveSpectrum(), ComplexChebyCoeff::set(), ComplexChebyCoeff::sub(), and write().

```
357 {return f.im;}
```

Here is the caller graph for this function:

1.7.5.40 Complex innerProduct (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*, NormType *n*, bool *normalize* = true)

References chebyInnerProduct(), L2InnerProduct(), and L2NORM.

```
1617                                        {
1618   return (n == L2NORM) ? L2InnerProduct(u,v,normalize)
: chebyInnerProduct(u,v,normalize);
1619 }
```

Here is the call graph for this function:

1.7.5.41 Real innerProduct (const ChebyCoeff & *f*, const ChebyCoeff & *g*, NormType *n*, bool *normalize* = true)

References chebyInnerProduct(), L2InnerProduct(), and L2NORM.

```
970                                        {
971   return (n==L2NORM)
? L2InnerProduct(u,v,normalize):chebyInnerProduct(u,v,normalize);
972 }
```

Here is the call graph for this function:

1.7.5.42 ComplexChebyCoeff integrate (const ComplexChebyCoeff & *df*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), ComplexChebyCoeff::im, integrate(), ComplexChebyCoeff::numModes(), ComplexChebyCoeff::re, and Spectral.

```
1298                                                {
1299   ComplexChebyCoeff u(du.numModes(), du.a(), du.b(), Spectral);
1300   integrate(du.re, u.re);
1301   integrate(du.im, u.im);
1302   return u;
1303 }
```

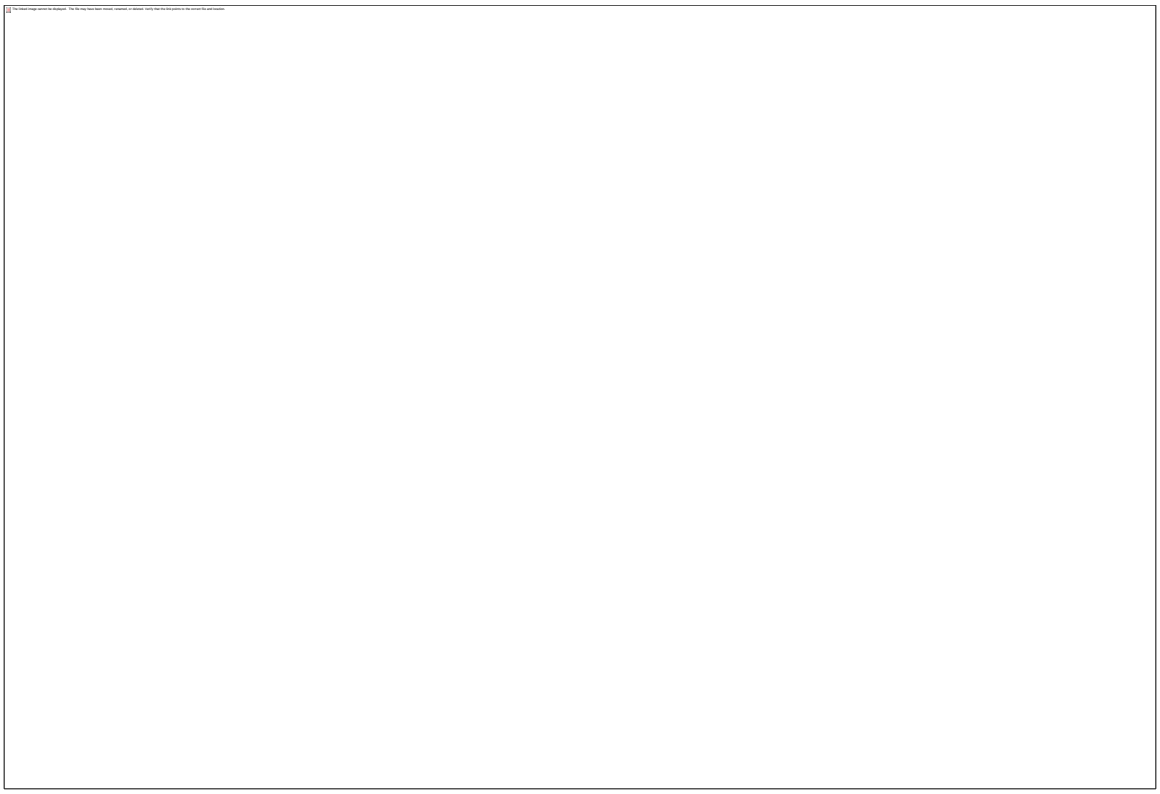
Here is the call graph for this function:

1.7.5.43 void integrate (const ComplexChebyCoeff & *df*, ComplexChebyCoeff & *f*)

References ComplexChebyCoeff::im, integrate(), and ComplexChebyCoeff::re.

```
1293                                                    {
1294   integrate(du.re, u.re);
1295   integrate(du.im, u.im);
1296 }
```

Here is the call graph for this function:

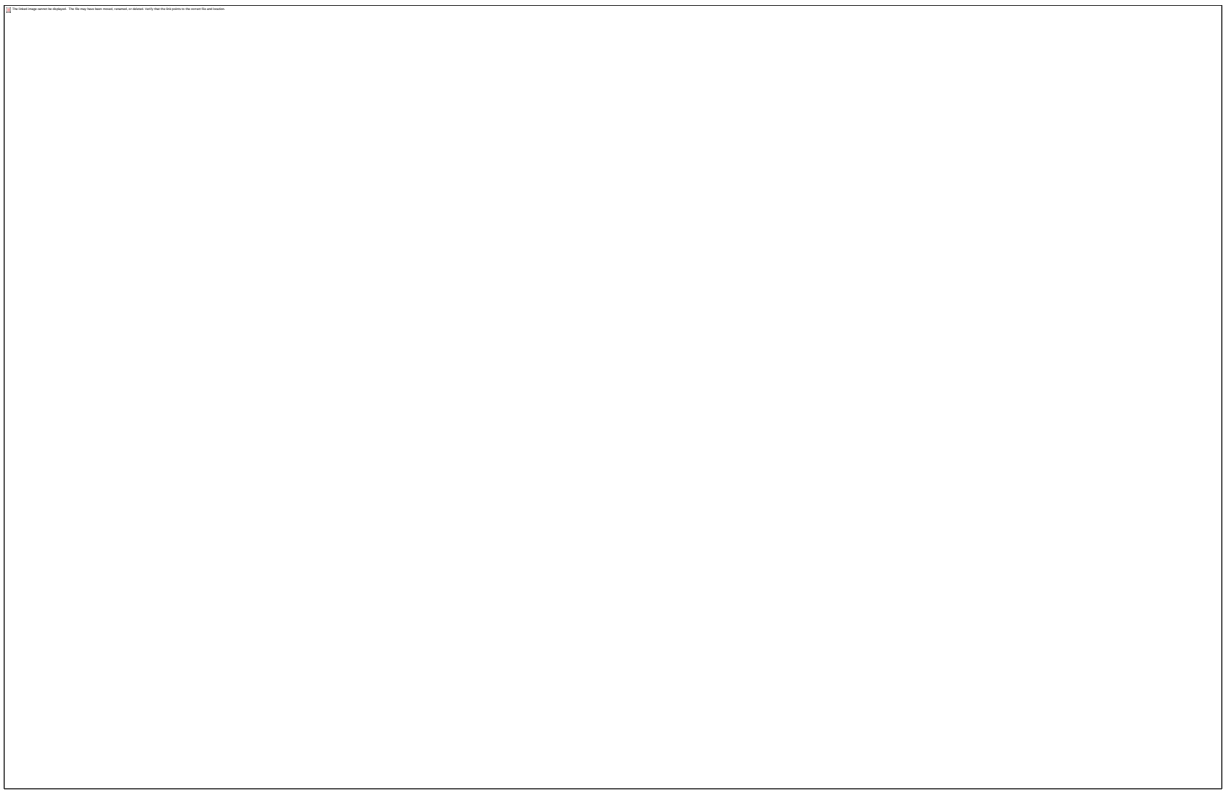


1.7.5.44 ChebyCoeff integrate (const ChebyCoeff & *df*)

References ChebyCoeff::a(), ChebyCoeff::b(), integrate(), Vector::length(), and Spectral.

```
676                                     {
677   ChebyCoeff u(dudy.length(), dudy.a(), dudy.b(), Spectral);
678   integrate(dudy, u);
679   return u;
680 }
```

Here is the call graph for this function:



1.7.5.45 void integrate (const ChebyCoeff & *df*, ChebyCoeff & *f*)

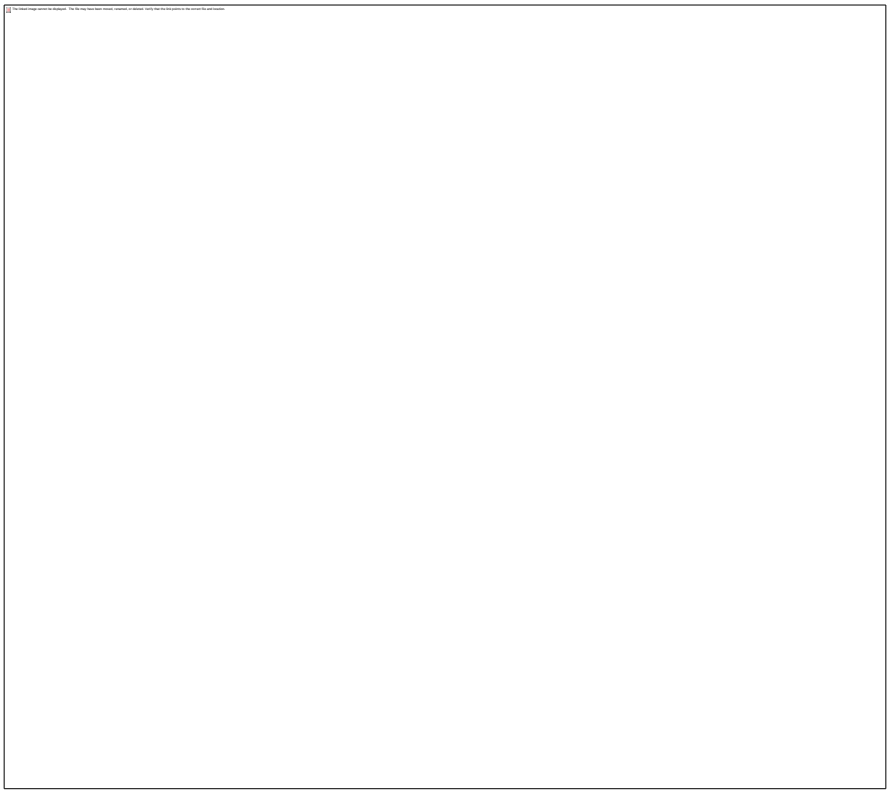
References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::mean(), ChebyCoeff::numModes(), Vector::resize(), ChebyCoeff::setBounds(), ChebyCoeff::setState(), Spectral, and ChebyCoeff::state().

Referenced by integrate(), L1Dist(), and L1Norm().

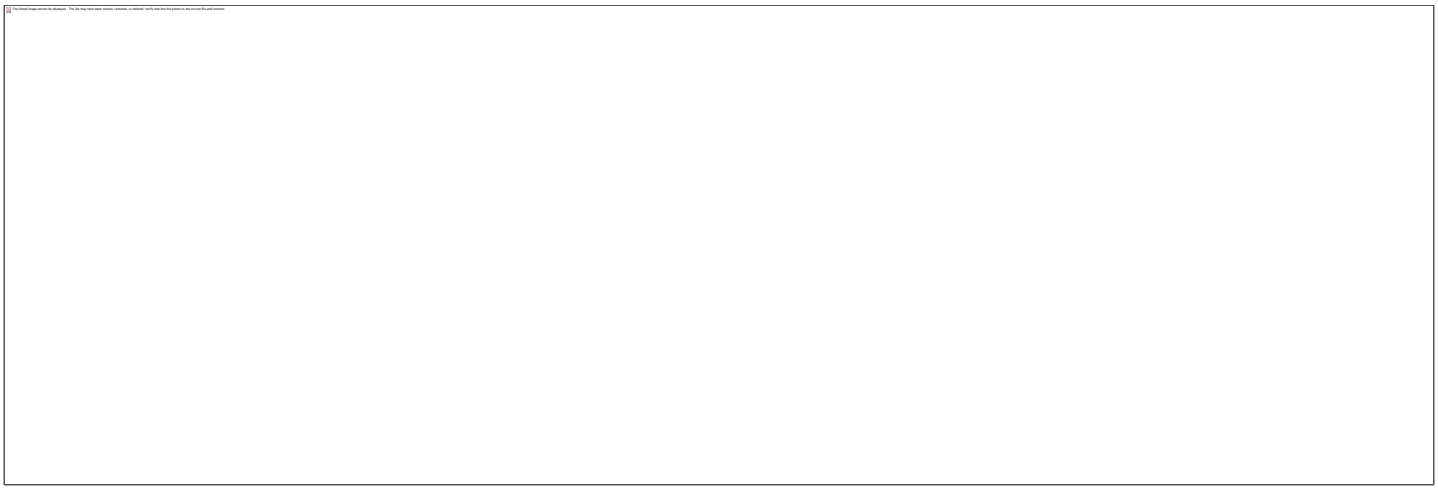
```
646                                         {
647   assert(dudy.state() == Spectral);
648   int N=dudy.numModes();
649   if (u.numModes() != N)
650     u.resize(N);
651   u.setBounds(u.a(), u.b());
652   u.setState(Spectral);
653 
654   Real h2 = (dudy.b()-dudy.a())/2;
655   switch (N) {
656   case 0:
657     break;
658   case 1:
659     u[0] = 0.0;
660     break;
661   case 2:
```

```
662     u[0] = 0;
663     u[1] = h2*dudy[0];
664     break;
665   default:
666     u[1] = h2*(dudy[0] - dudy[2]/2);
667     for (int n=2; n<N-1; ++n)
668       u[n] = h2*(dudy[n-1] - dudy[n+1])/(2*n);
669     u[N-1] = h2*dudy[N-2]/(2*(N-1));
670     u[0] -= u.mean();  // const term is arbitrary, set to zero.
671   }
672   return;
673 
674 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

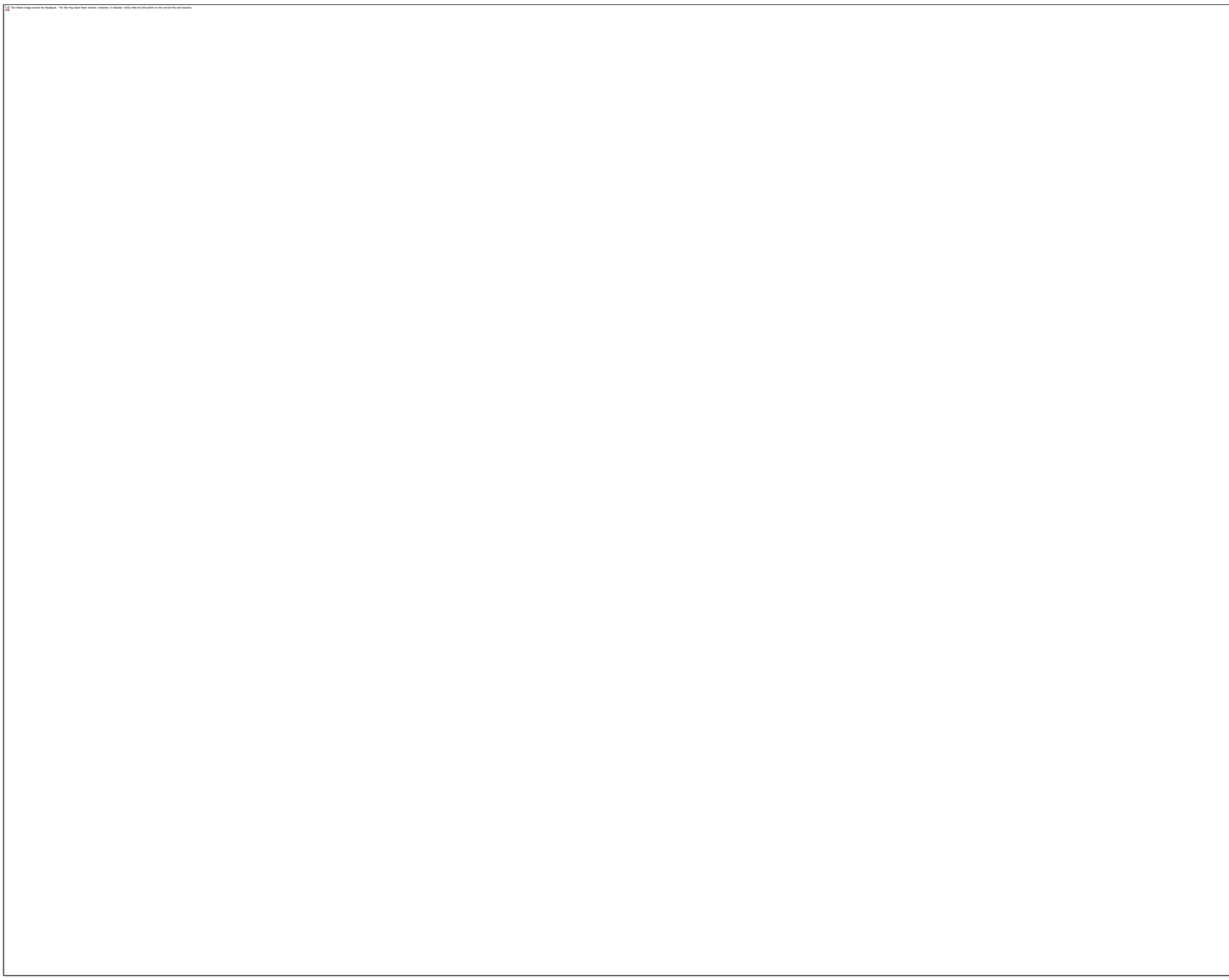


1.7.5.46 Real L1Dist (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*, bool *normalize* = true)

References ComplexChebyCoeff::im, L1Dist(), and ComplexChebyCoeff::re.

```
1536                        {
1537   return L1Dist(u.re, v.re, normalize) + L1Dist(u.im, v.im, normalize);
1538 }
```

Here is the call graph for this function:



1.7.5.47 Real L1Dist (const ChebyCoeff & *f*, const ChebyCoeff & *g*, bool *normalize* = true)

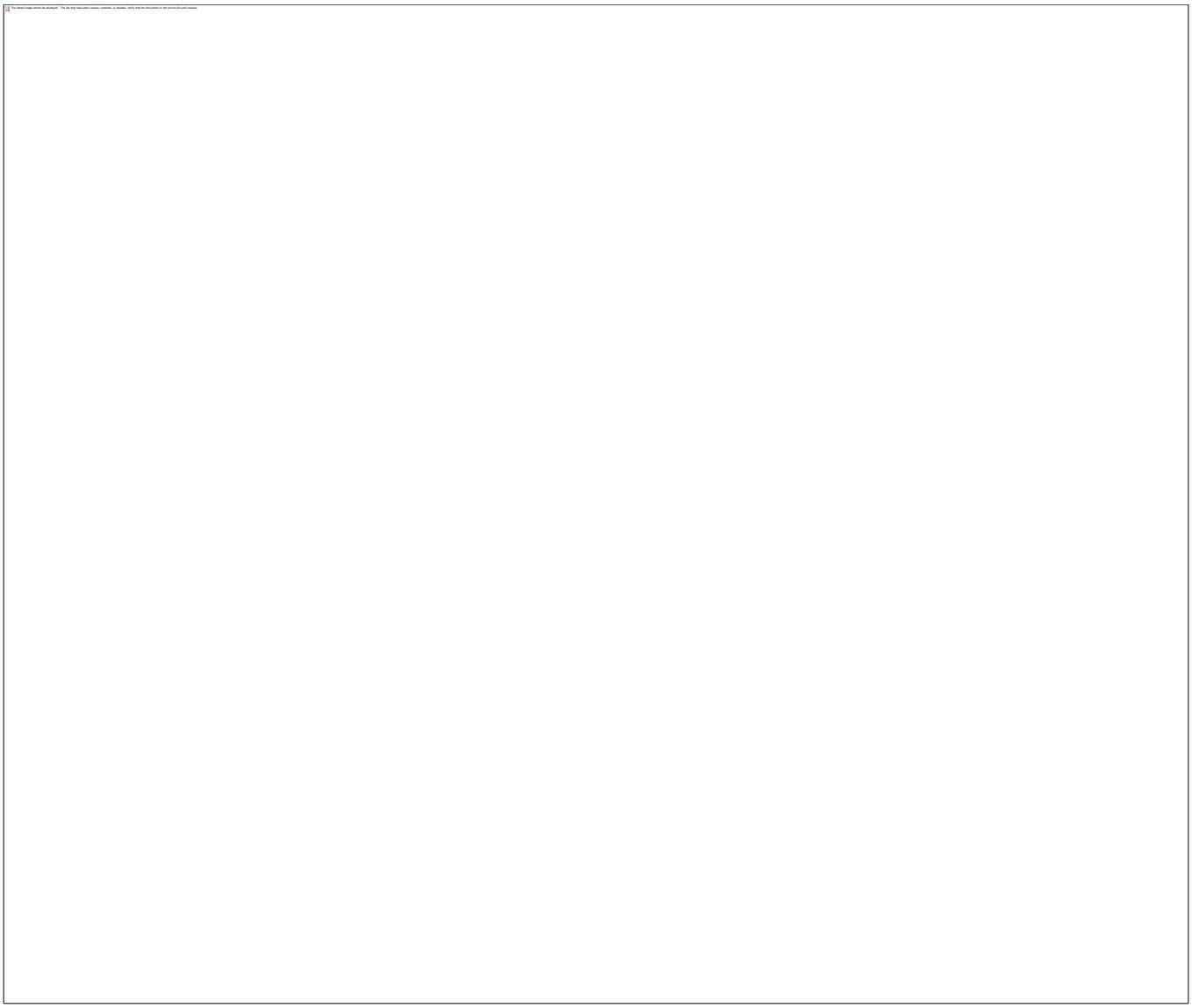
References ChebyCoeff::a(), abs(), ChebyCoeff::b(), ChebyCoeff::congruent(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), integrate(), ChebyCoeff::makeSpectral(), ChebyCoeff::N(), and ChebyCoeff::state().

Referenced by L1Dist(), and HelmholtzSolver::residual().

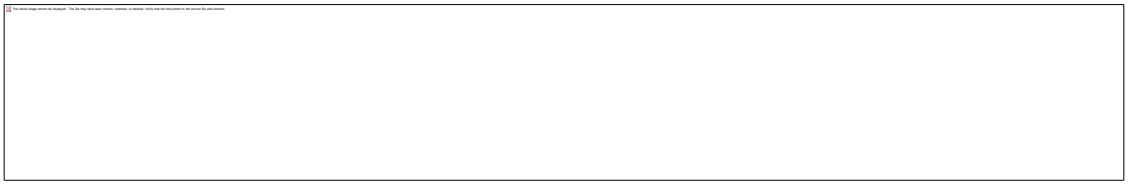
```
925                                                 {
926   assert(f.congruent(g));
927   fieldstate fs = f.state();
928   fieldstate gs = f.state();
929   ((ChebyCoeff&) f).makePhysical();
```

```
930   ((ChebyCoeff&) g).makePhysical();
931
932   ChebyCoeff d(f);
933   for (int n=0; n<g.N(); ++n)
934     d[n] = abs(f[n]-g[n]);
935   d.makeSpectral();
936
937   ChebyCoeff D(g);
938   integrate(d,D);
939
940   Real rtn = D.eval_b() - D.eval_a();
941
942   if (normalize)
943     rtn /= d.b() - d.a();
944
945   ((ChebyCoeff&) f).makeState(fs);
946   ((ChebyCoeff&) g).makeState(gs);
947
948   return rtn;
949 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



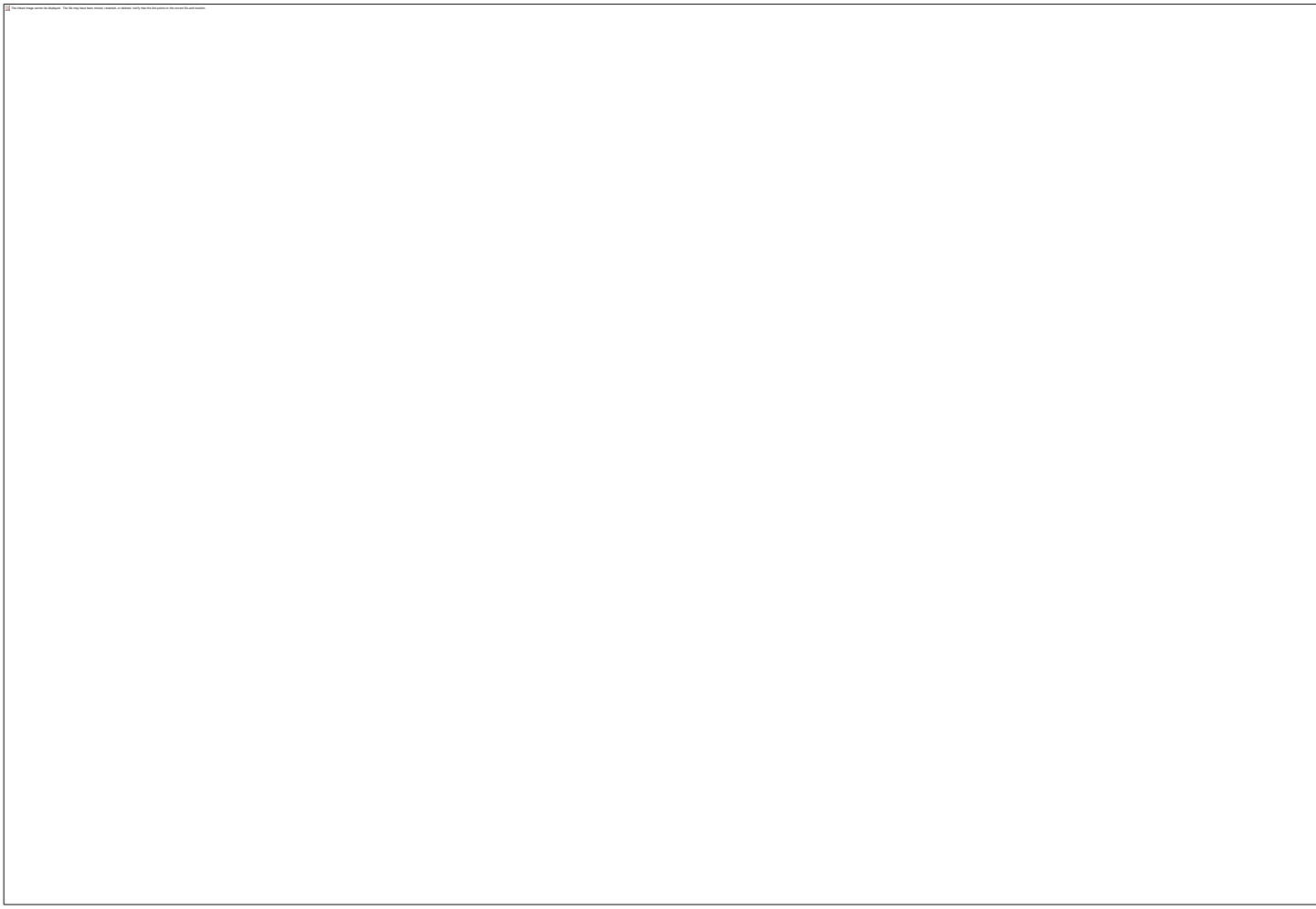
1.7.5.48 Real L1Norm (const ComplexChebyCoeff & *f*, bool *normalize* = true)

References ComplexChebyCoeff::im, L1Norm(), and ComplexChebyCoeff::re.

```
1531                                                {
1532   return L1Norm(u.re, normalize) + L1Norm(u.im, normalize);
```

```
1533 }
```

Here is the call graph for this function:



1.7.5.49 Real L1Norm (const ChebyCoeff & *f*, bool *normalize* = true)

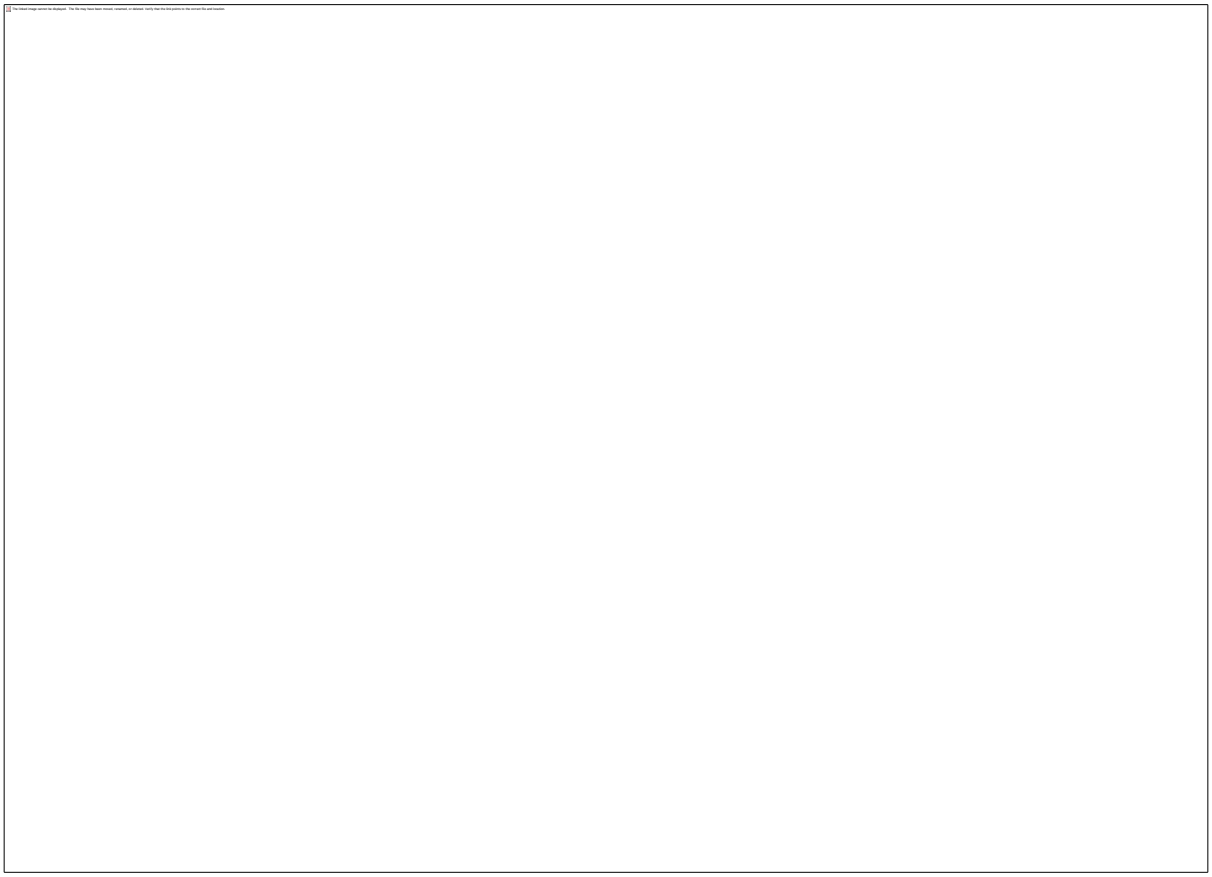
References ChebyCoeff::a(), abs(), ChebyCoeff::b(), ChebyCoeff::eval_a(), ChebyCoeff::eval_b(), integrate(), ChebyCoeff::makePhysical(), ChebyCoeff::makeSpectral(), and ChebyCoeff::N().

Referenced by L1Dist(), L1Norm(), and HelmholtzSolver::verify().

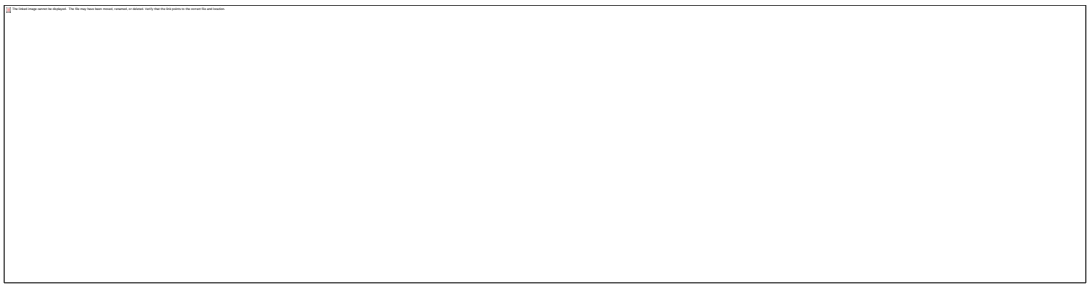
```
907                                     {
908   ChebyCoeff g(f);
909   g.makePhysical();
910   for (int n=0; n<g.N(); ++n)
911     g[n] = abs(g[n]);
912   g.makeSpectral();
913
```

```
914   ChebyCoeff G(g);
915   integrate(g,G);
916
917   Real rtn = G.eval_b() - G.eval_a();
918
919   if (normalize)
920     rtn /= g.b() - g.a();
921
922   return rtn;
923 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.7.5.50 Real L2Dist (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*, bool *normalize* = true)

References ComplexChebyCoeff::im, L2Dist2(), and ComplexChebyCoeff::re.

```
1484                                {
1485   return sqrt(L2Dist2(u.re, v.re, normalize) + L2Dist2(u.im, v.im, normalize));
1486 }
```

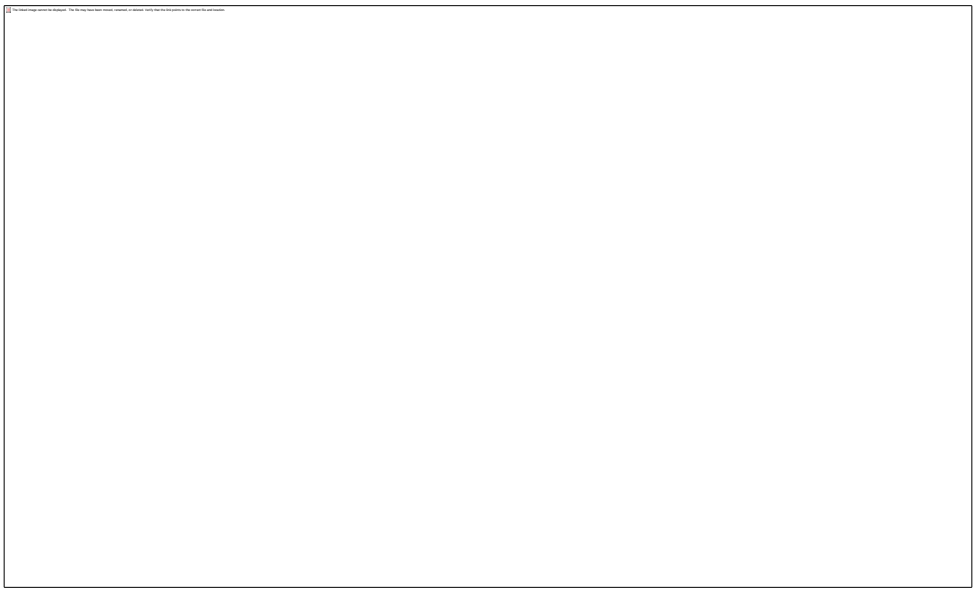
Here is the call graph for this function:

1.7.5.51 Real L2Dist (const ChebyCoeff & *f*, const ChebyCoeff & *g*, bool *normalize* = true)

References L2Dist2().

```
800                                         {
801   return sqrt(L2Dist2(u,v, normalize));
802 }
```

Here is the call graph for this function:



1.7.5.52 Real L2Dist2 (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*, bool *normalize* = true)

References ComplexChebyCoeff::im, L2Dist2(), and ComplexChebyCoeff::re.

```
1475                                {
1476   return L2Dist2(u.re, v.re, normalize) + L2Dist2(u.im, v.im, normalize);
1477 }
```

Here is the call graph for this function:

1.7.5.53 Real L2Dist2 (const ChebyCoeff & *f*, const ChebyCoeff & *g*, bool *normalize* = true)

References L2Norm2().

```
790                                                 {
791   ChebyCoeff tmp(u);
792   tmp -= v;
793   return L2Norm2(tmp, normalize);
794 }
```

Here is the call graph for this function:

1.7.5.54 Complex L2InnerProduct (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*, bool *normalize* = true)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), ComplexChebyCoeff::im, ComplexChebyCoeff::numModes(), ComplexChebyCoeff::re, Spectral, and ComplexChebyCoeff::state().

```
1489                                {
1490   assert(u.state()==Spectral && v.state()==Spectral);
1491   assert(u.numModes() == v.numModes());
1492   int N = u.numModes();
1493
1494   // Unroll the complex operations, for efficiency.
1495   // Profiling an L2InnerProduct-intensive program reveals that
1496   // complex * double is expensive
1497   Real e=1.0;
1498   Real sum_r=0.0;
1499   Real sum_i=0.0;
1500   Real psum_r;
1501   Real psum_i;
1502   Real um_r;
1503   Real um_i;
1504   Real vn_r;
1505   Real vn_i;
1506   Real k;
1507   for (int m=N-1; m>=0; --m) {
1508     um_r = u.re[m];
1509     um_i = u.im[m];
1510     psum_r = 0.0;
1511     psum_i = 0.0;
1512     for (int n=m%2; n<N; n+=2) {
1513       k = (e-m*m-n*n)/((e+m-n)*(e-m+n)*(e+m+n)*(e-m-n));
1514       vn_r = v.re[n];
1515       vn_i = v.im[n];
1516
1517       psum_r += k*(um_r*vn_r + um_i*vn_i);
1518       psum_i += k*(um_i*vn_r - um_r*vn_i);
1519     }
1520     sum_r += psum_r;
1521     sum_i += psum_i;
1522   }
1523   if (!normalize) {
1524     sum_r *= u.b()-u.a();
1525     sum_i *= u.b()-u.a();
1526   }
1527   return Complex(sum_r, sum_i);
1528 }
```

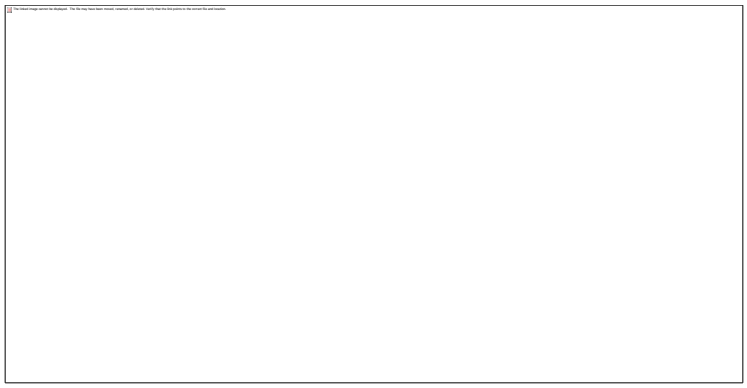
Here is the call graph for this function:

1.7.5.55 Real L2InnerProduct (const ChebyCoeff & *f*, const ChebyCoeff & *g*, bool *normalize* = true)

References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::numModes(), Spectral, and ChebyCoeff::state().

```
804                                                    {
805   assert(u.state()==Spectral && v.state()==Spectral);
806   assert(u.a()==v.a() && u.b()==v.b());
807   assert(u.numModes() ==  v.numModes());
808   int N = u.numModes();
809   Real sum=0.0;
810   Real e=1.0;
811   for (int m=N-1; m>=0; --m) {
812     Real um = u[m];
813     Real psum = 0.0;
814     for (int n=m%2; n<N; n+=2)
815       psum += um*v[n]*(e-m*m-n*n)/((e+m-n)*(e-m+n)*(e+m+n)*(e-m-n));
816     sum += psum;
817   }
818   if (!normalize)
819     sum *= u.b()-u.a();
820   return sum;
821 }
```

Here is the call graph for this function:

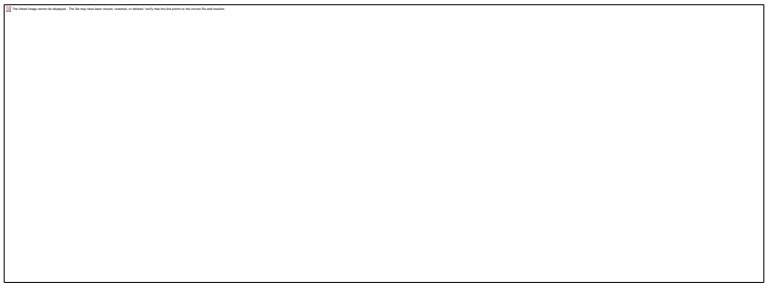


1.7.5.56 Real L2Norm (const ComplexChebyCoeff & *f*, bool *normalize* = true)

References ComplexChebyCoeff::im, L2Norm2(), and ComplexChebyCoeff::re.

```
1479                                                {
1480   return sqrt(L2Norm2(u.re, normalize) + L2Norm2(u.im, normalize));
1481 }
```

Here is the call graph for this function:

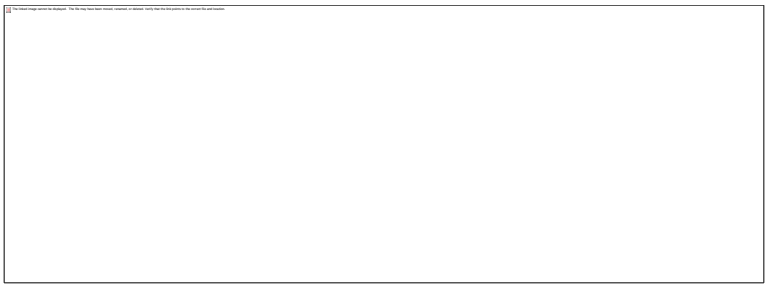


1.7.5.57 Real L2Norm (const ChebyCoeff & *f*, bool *normalize* = true)

References L2Norm2().

```
796                                             {
797   return sqrt(L2Norm2(u, normalize));
798 }
```

Here is the call graph for this function:

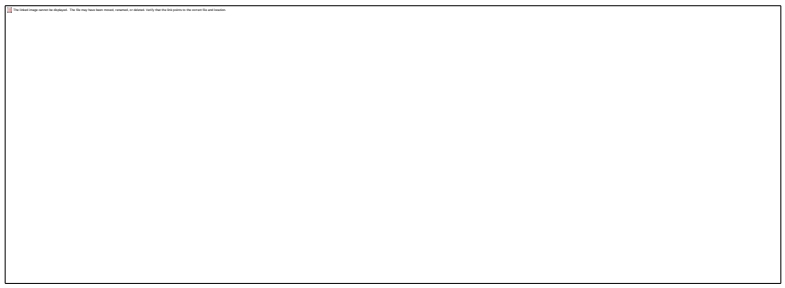


1.7.5.58 Real L2Norm2 (const ComplexChebyCoeff & *f*, bool *normalize* = true)

References ComplexChebyCoeff::im, L2Norm2(), and ComplexChebyCoeff::re.

```
1470                                                {
1471   return L2Norm2(u.re, normalize) + L2Norm2(u.im, normalize);
1472 }
```

Here is the call graph for this function:



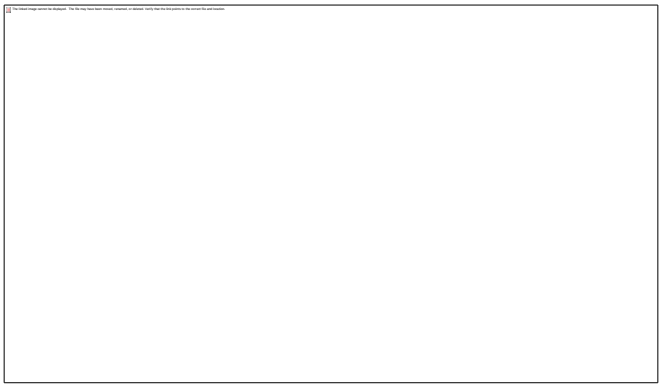
1.7.5.59 Real L2Norm2 (const ChebyCoeff & *f*, bool *normalize* = true)

References ChebyCoeff::a(), ChebyCoeff::b(), ChebyCoeff::numModes(), Spectral, and ChebyCoeff::state().

```
773                                             {
774   assert(u.state() == Spectral);
775   int N=u.numModes();
776   Real sum=0.0;
777   Real e=1.0;
778   for (int m=N-1; m>=0; --m) {
779     Real um = u[m];
780     Real psum = 0.0;
781     for (int n=m%2; n<N; n+=2)
782       psum += um*u[n]*(e-m*m-n*n)/((e+m-n)*(e-m+n)*(e+m+n)*(e-m-n));
783     sum += psum;
784   }
785   if (!normalize)
```

```
786     sum *= u.b()-u.a();
787   return sum;
788 }
```

Here is the call graph for this function:

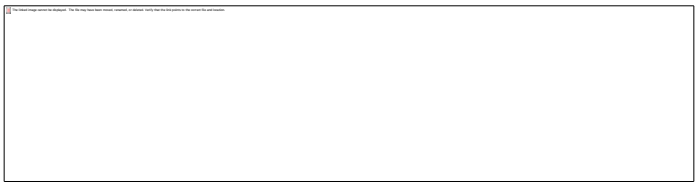


1.7.5.60 ChebyCoeff legendre (int *N*, int *n*, bool *normalize* = false)

References legendre(), ChebyCoeff::makeSpectral(), Physical, and pi.

```
66                                          {
67   assert(n<N);
68   ChebyCoeff Pn(N, -1 , 1, Physical);
69   Real piN=pi/(N-1);
70   for (int q=0; q<N; ++q)
71     Pn[q] = legendre(n, cos(q*piN));
72   ChebyTransform trans(N);
73   Pn.makeSpectral(trans);
74   if (normalize)
75     Pn *= sqrt((double) (2*n+1));
76   return Pn;
77 }
```

Here is the call graph for this function:



1.7.5.61 Real legendre (int *n*, Real *x*)

Referenced by legendre().

```
51                          {
52   Real p=1.0;
53   Real q=0.0;
54   for (int m=0; m<n; ++m) {
55     Real r = q;
56     q = p;
57     p = ((2*m+1)*x*q-m*r)/(m+1);
58   }
59   return p;
60 }
```

Here is the caller graph for this function:

1.7.5.62 Real LinfDist (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*)

References ComplexChebyCoeff::im, LinfDist(), and ComplexChebyCoeff::re.

```
1544                                                    {
1545   return LinfDist(u.re, v.re) + LinfDist(u.im, v.im);
1546 }
```

Here is the call graph for this function:

1.7.5.63 Real LinfDist (const ChebyCoeff & *f*, const ChebyCoeff & *g*)

References abs(), ChebyCoeff::congruent(), Greater(), ChebyCoeff::N(), and ChebyCoeff::state().

Referenced by LinfDist().

```
890                                             {
891   assert(f.congruent(g));
892   fieldstate fs = f.state();
893   fieldstate gs = f.state();
894   ((ChebyCoeff&) f).makePhysical();
895   ((ChebyCoeff&) g).makePhysical();
896
897   Real rtn=0.0;
898   for (int m=0; m<f.N(); ++m)
899     rtn = Greater(rtn, abs(f[m]-g[m]));
900
901   ((ChebyCoeff&) f).makeState(fs);
902   ((ChebyCoeff&) g).makeState(gs);
903   return rtn;
904 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.7.5.64 Real LinfNorm (const ComplexChebyCoeff & *f*)

References ComplexChebyCoeff::im, LinfNorm(), and ComplexChebyCoeff::re.

```
1540                                    {
```

```
1541   return LinfNorm(u.re) + LinfNorm(u.im);
1542 }
```

Here is the call graph for this function:

1.7.5.65 Real LinfNorm (const ChebyCoeff & *f*)

References abs(), Greater(), ChebyCoeff::N(), and ChebyCoeff::state().

Referenced by LinfDist(), and LinfNorm().

```
877                                {
878   fieldstate s = f.state();
879   ((ChebyCoeff&) f).makePhysical();
880
881   Real rtn=0.0;
882   for (int m=0; m<f.N(); ++m)
883     rtn = Greater(abs(f[m]), rtn);
884
885   ((ChebyCoeff&) f).makeState(s);
886
887   return rtn;
888 }
```

Here is the call graph for this function:

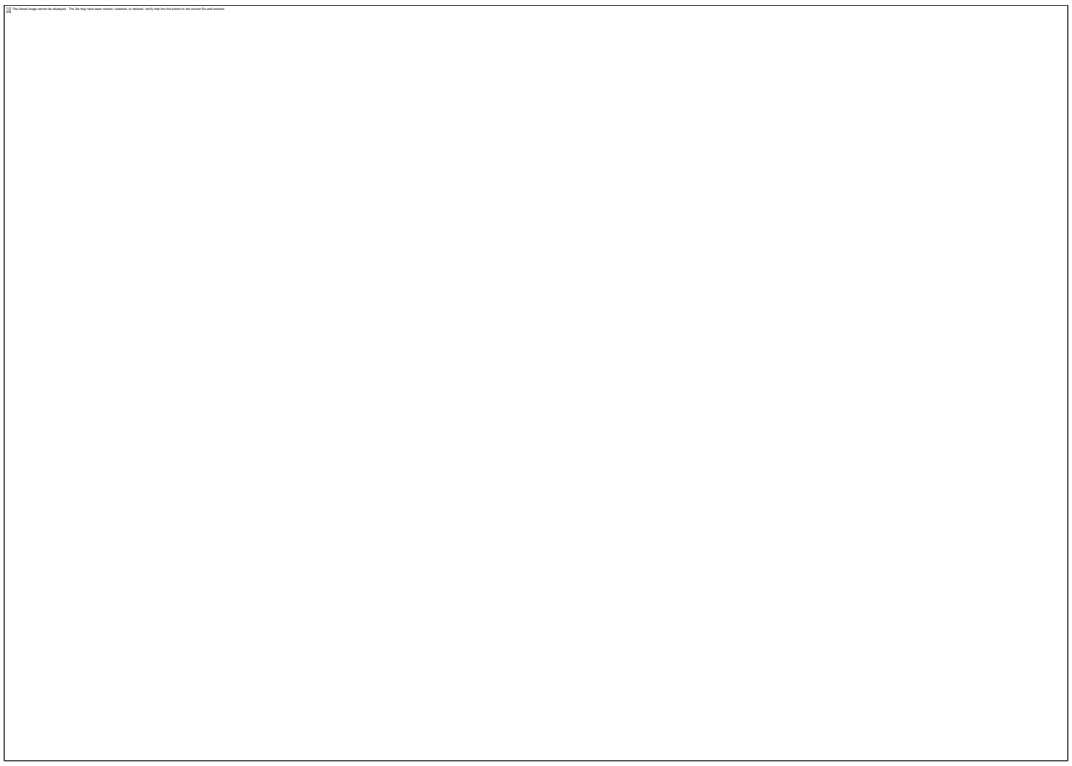
Here is the caller graph for this function:

1.7.5.66 Real norm (const ComplexChebyCoeff & *f*, NormType *n*, bool *normalize* = true)

References chebyNorm(), L2Norm(), and L2NORM.

```
1608                                                    {
1609   return (n == L2NORM) ? L2Norm(u,normalize) : chebyNorm(u,normalize);
1610 }
```

Here is the call graph for this function:

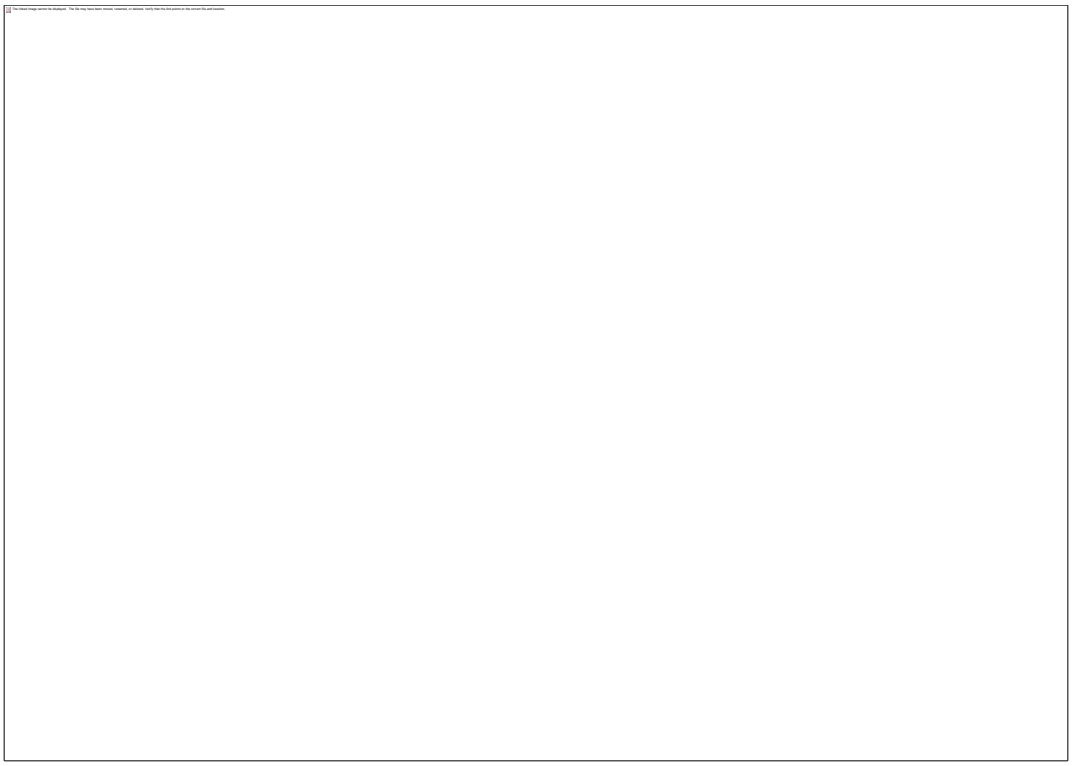


1.7.5.67 Real norm (const ChebyCoeff & *f*, NormType *n*, bool *normalize* = true)

References chebyNorm(), L2Norm(), and L2NORM.

```
962                                                 {
963   return (n==L2NORM) ? L2Norm(u,normalize) : chebyNorm(u,normalize);
964 }
```

Here is the call graph for this function:

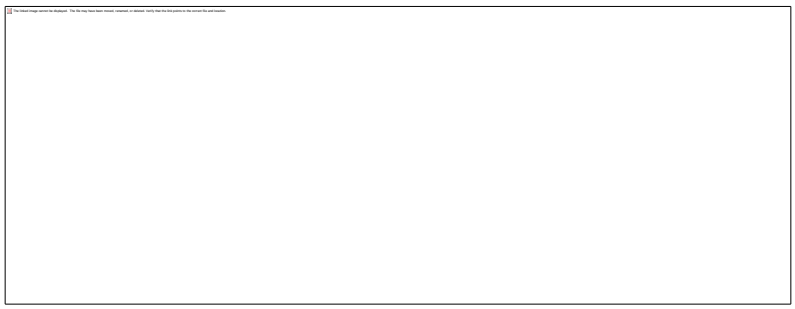


1.7.5.68 Real norm2 (const ComplexChebyCoeff & *f*, NormType *n*, bool *normalize* = true)

References chebyNorm2(), L2NORM, and L2Norm2().

```
1599                                                {
1600   return (n == L2NORM) ? L2Norm2(u,normalize) : chebyNorm2(u,normalize);
1601 }
```

Here is the call graph for this function:

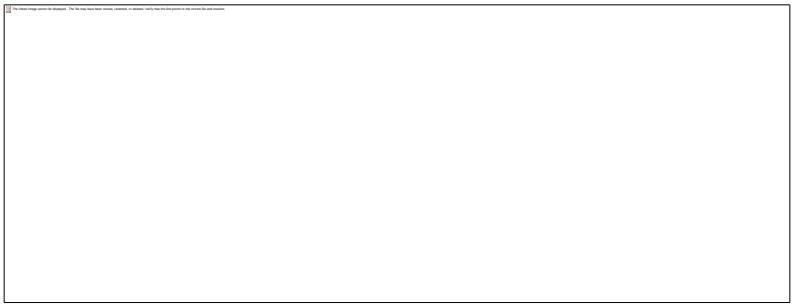


1.7.5.69 Real norm2 (const ChebyCoeff & *f*, NormType *n*, bool *normalize* = true)

References chebyNorm2(), L2NORM, and L2Norm2().

```
953                                                 {
954   return (n==L2NORM) ? L2Norm2(u,normalize) : chebyNorm2(u,normalize);
955 }
```

Here is the call graph for this function:



1.7.5.70 bool operator!= (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*)

```
1228                                                    {
1229   return !(u==v);
1230 }
```

1.7.5.71 bool operator!= (const ChebyCoeff & *f*, const ChebyCoeff & *g*)

```
620                                                 {
621   return !(u==v);
622 }
```

1.7.5.72 ComplexChebyCoeff operator* (Real *c*, const ComplexChebyCoeff & *g*)

```
1238                                                    {
1239   ComplexChebyCoeff rtn(v);
1240   rtn *= c;
1241   return rtn;
1242 }
```

1.7.5.73 ChebyCoeff operator* (Real *c*, const ChebyCoeff & *g*)

```
580                                                 {
581   ChebyCoeff rtn(v);
582   rtn *= c;
```

```
583   return rtn;
584 }
```

1.7.5.74 ComplexChebyCoeff operator+ (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*)

```
1244                                                    {
1245   ComplexChebyCoeff rtn(u);
1246   rtn += v;
1247   return rtn;
1248 }
```

1.7.5.75 ChebyCoeff operator+ (const ChebyCoeff & *f*, const ChebyCoeff & *g*)

```
599                                                {
600   ChebyCoeff rtn(u);
601   rtn += v;
602   return rtn;
603 }
```

1.7.5.76 ComplexChebyCoeff operator- (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*)

```
1250                                                    {
1251   ComplexChebyCoeff rtn(u);
1252   rtn -= v;
1253   return rtn;
1254 }
```

1.7.5.77 ChebyCoeff operator- (const ChebyCoeff & *f*, const ChebyCoeff & *g*)

```
605                                                {
606   ChebyCoeff rtn(u);
607   rtn -= v;
608   return rtn;
609 }
```

1.7.5.78 std::ostream& operator<< (std::ostream & *os*, const ComplexChebyCoeff & *f*)

References ComplexChebyCoeff::im, ComplexChebyCoeff::length(), and ComplexChebyCoeff::re.

```
1462                                                {
1463   for (int i=0; i<u.length(); ++i)
1464     os << '(' << u.re[i] << ", " << u.im[i] << ")\n";
1465   return os;
1466 }
```

Here is the call graph for this function:

1.7.5.79 bool operator== (const ComplexChebyCoeff & *f*, const ComplexChebyCoeff & *g*)

References ComplexChebyCoeff::im, and ComplexChebyCoeff::re.

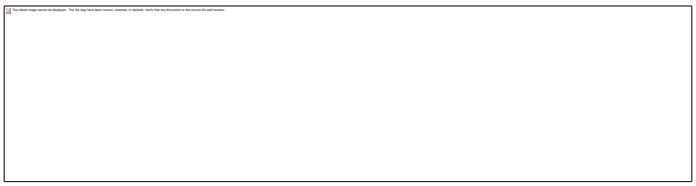
```
1225                                                {
1226   return u.re == v.re && u.im == v.im;
1227 }
```

1.7.5.80 bool operator== (const ChebyCoeff & *f*, const ChebyCoeff & *g*)

References ChebyCoeff::congruent(), and ChebyCoeff::numModes().

```
611                                                {
612   if (!u.congruent(v))
613     return false;
614   for (int i=0; i<u.numModes(); ++i)
615     if (u[i] != v[i])
616       return false;
617   return true;
618 }
```

Here is the call graph for this function:



1.7.5.81 Vector periodicpoints (int *N*, Real *L*)

```
763                             {
764   Vector x(N+1);
765   Real dx=L/N;
766   for (int n=0; n<=N; ++n)
767     x[n] = n*dx;
768   return x;
769 }
```

1.7.5.82 const ChebyCoeff & Re (const ComplexChebyCoeff & *f*) [inline]

References ComplexChebyCoeff::re.

```
358 {return f.re;}
```

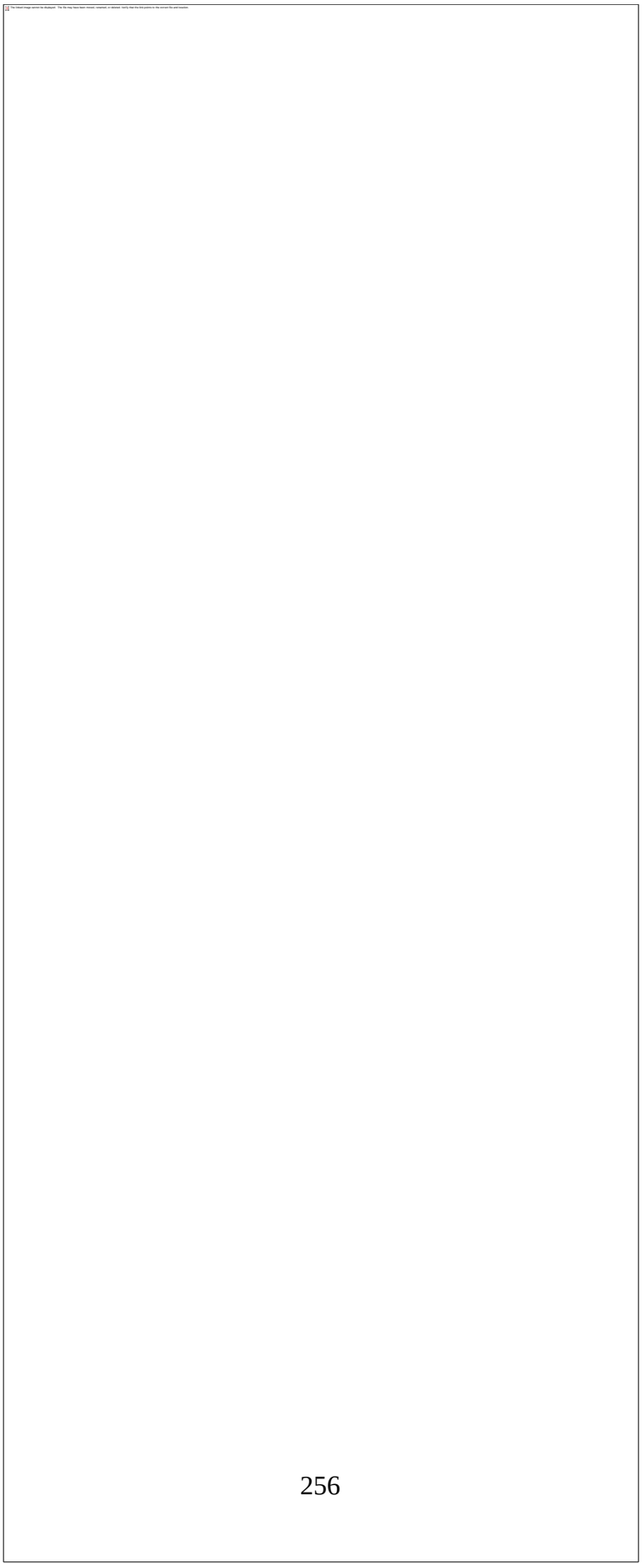
1.7.5.83 ChebyCoeff & Re (ComplexChebyCoeff & *f*) [inline]

References ComplexChebyCoeff::re.

Referenced by ComplexChebyCoeff::add(), TurbStats::addData(), PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), FlowField::asciiSave(), DNSAlgorithm::DNSAlgorithm(), FlowField::dudy_a(), FlowField::dudy_b(), exp(), FieldSeries::interpolate(), L1Dist(), L1Norm(), L2InnerProduct(), LinfDist(), LinfNorm(), ComplexChebyCoeff::operator*=(), operator<<(), save(), BasisFunc::save(), FlowField::saveProfile(), FlowField::saveSpectrum(), ComplexChebyCoeff::set(), ComplexChebyCoeff::sub(), TauSolver::verify(), and write().

```
356 {return f.re;}
```

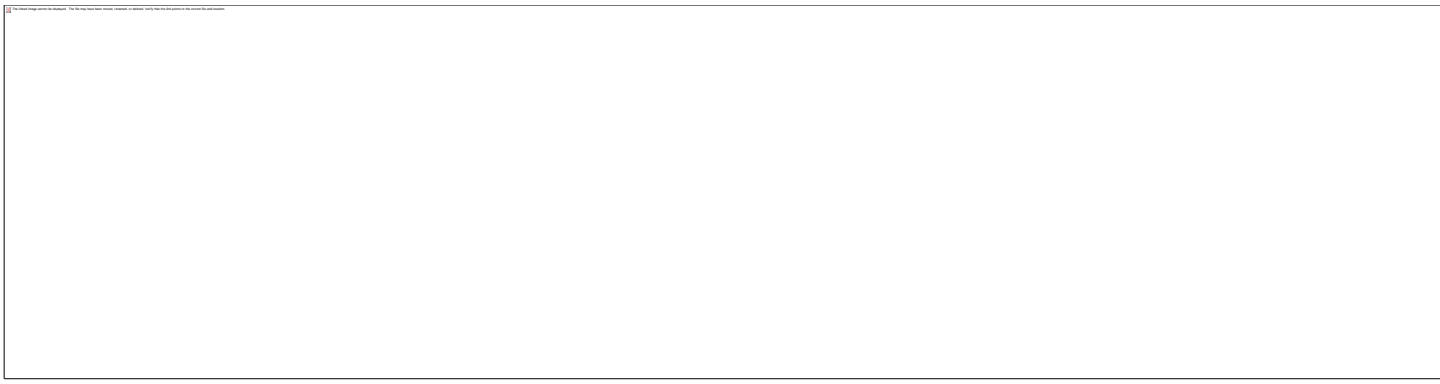
Here is the caller graph for this function:


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.

1.7.5.84

1.8 /_cuda/channelflow-0.9.22/channelflow/complexdefs.h File Reference

#include "channelflow/mathdefs.h"
Include dependency graph for complexdefs.h:



1.8.1 Functions

- Complex exp (const Complex &z)
- Complex log (const Complex &z)

1.8.2 Function Documentation

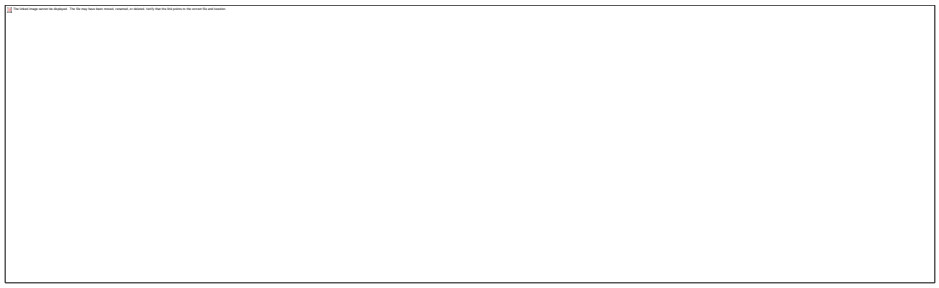
1.8.2.1 Complex exp (const Complex & z) [inline]

References Im(), and Re().
Referenced by assignOrrSommField(), PressureSolver::solve(), and FlowField::translate().

```
39                                {
40   return exp(Re(z)) * Complex(cos(Im(z)), sin(Im(z)));
41 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.8.2.2 Complex log (const Complex & *z*) [inline]

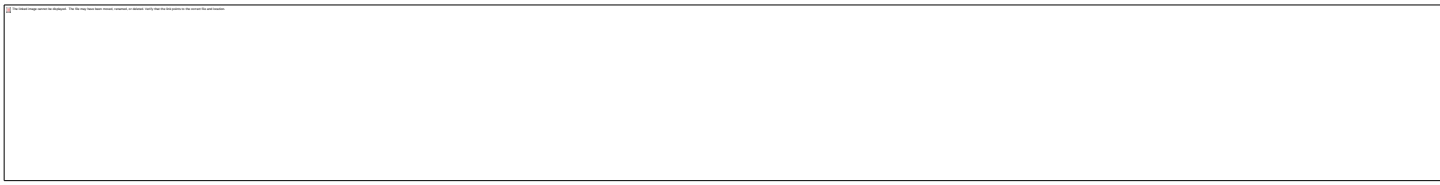
References abs().

Referenced by randomComplex().

```
42                                  {
43   return Complex(log(abs(z)), arg(z));
44 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.8.2.3

1.9 /_cuda/channelflow-0.9.22/channelflow/diffops.cpp File Reference

```
#include <fstream>
#include <iomanip>
#include "channelflow/diffops.h"
#include "channelflow/dns.h"
#include <time.h>
```

Include dependency graph for diffops.cpp:

1.9.1 Functions

- void xdiff (const FlowField &f_, FlowField &dfdx, int n)
- void ydiff (const FlowField &f_, FlowField &dfdy, int n)
- void zdiff (const FlowField &f_, FlowField &dfdz, int n)
- void diff (const FlowField &f_, FlowField &df, int nx, int ny, int nz)
- void grad (const FlowField &f_, FlowField &gradf)
- void lapl (const FlowField &f_, FlowField &laplf)
- void norm2 (const FlowField &f_, FlowField &norm2f)
- void norm (const FlowField &f_, FlowField &normf)
- void curl (const FlowField &f_, FlowField &curlf)
- void outer (const FlowField &f_, const FlowField &g_, FlowField &fg)
- void dot (const FlowField &f_, const FlowField &g_, FlowField &fdotg)
- void div (const FlowField &f_, FlowField &divf)
- void cross (const FlowField &f_, const FlowField &g_, FlowField &fcg)
- void energy (const FlowField &u_, FlowField &e)
- void energy (const FlowField &u_, const ChebyCoeff &U_, FlowField &e)
- FlowField xdiff (const FlowField &f, int n)
- FlowField ydiff (const FlowField &f, int n)
- FlowField zdiff (const FlowField &f, int n)
- FlowField diff (const FlowField &f, int nx, int ny, int nz)
- FlowField grad (const FlowField &f)
- FlowField lapl (const FlowField &f)
- FlowField curl (const FlowField &f)
- FlowField norm (const FlowField &f)
- FlowField norm2 (const FlowField &f)
- FlowField div (const FlowField &f)
- FlowField outer (const FlowField &f, const FlowField &g)
- FlowField cross (const FlowField &f, const FlowField &g)
- FlowField dot (const FlowField &f, const FlowField &g)
- FlowField energy (const FlowField &u)
- FlowField energy (const FlowField &u, ChebyCoeff &U)
- void navierstokesNL (const FlowField &u, const ChebyCoeff &U, FlowField &f, FlowField &tmp, NonlinearMethod &method)
- void rotationalNL (const FlowField &u_, const ChebyCoeff &U_, FlowField &f, FlowField &tmp)
- void convectionNL (const FlowField &u_, const ChebyCoeff &U_, FlowField &f, FlowField &tmp)
- void divergenceNL (const FlowField &u_, const ChebyCoeff &U_, FlowField &f, FlowField &tmp)
- void skewsymmetricNL (const FlowField &u_, const ChebyCoeff &U_, FlowField &f, FlowField &tmp)

- void skewsymmetricNL_GPU (const FlowField &u_, const ChebyCoeff &U_, FlowField &f, FlowField &tmp)
- void skewsymmetricNL_THREAD (const FlowField &u_, const ChebyCoeff &U_, FlowField &f, FlowField &tmp)
- void linearizedNL (const FlowField &u_, const ChebyCoeff &U_, FlowField &f)
- void uq2p (const FlowField &u_, const FlowField &q, FlowField &p, NonlinearMethod nonl_method)
- void up2q (const FlowField &u_, const FlowField &p, FlowField &q, NonlinearMethod nonl_method)
- ostream & operator<< (ostream &os, NonlinearMethod nonl)

1.9.2 Function Documentation

1.9.2.1 void convectionNL (const FlowField & *u_*, const ChebyCoeff & *U_*, FlowField & *f*, FlowField & *tmp*)

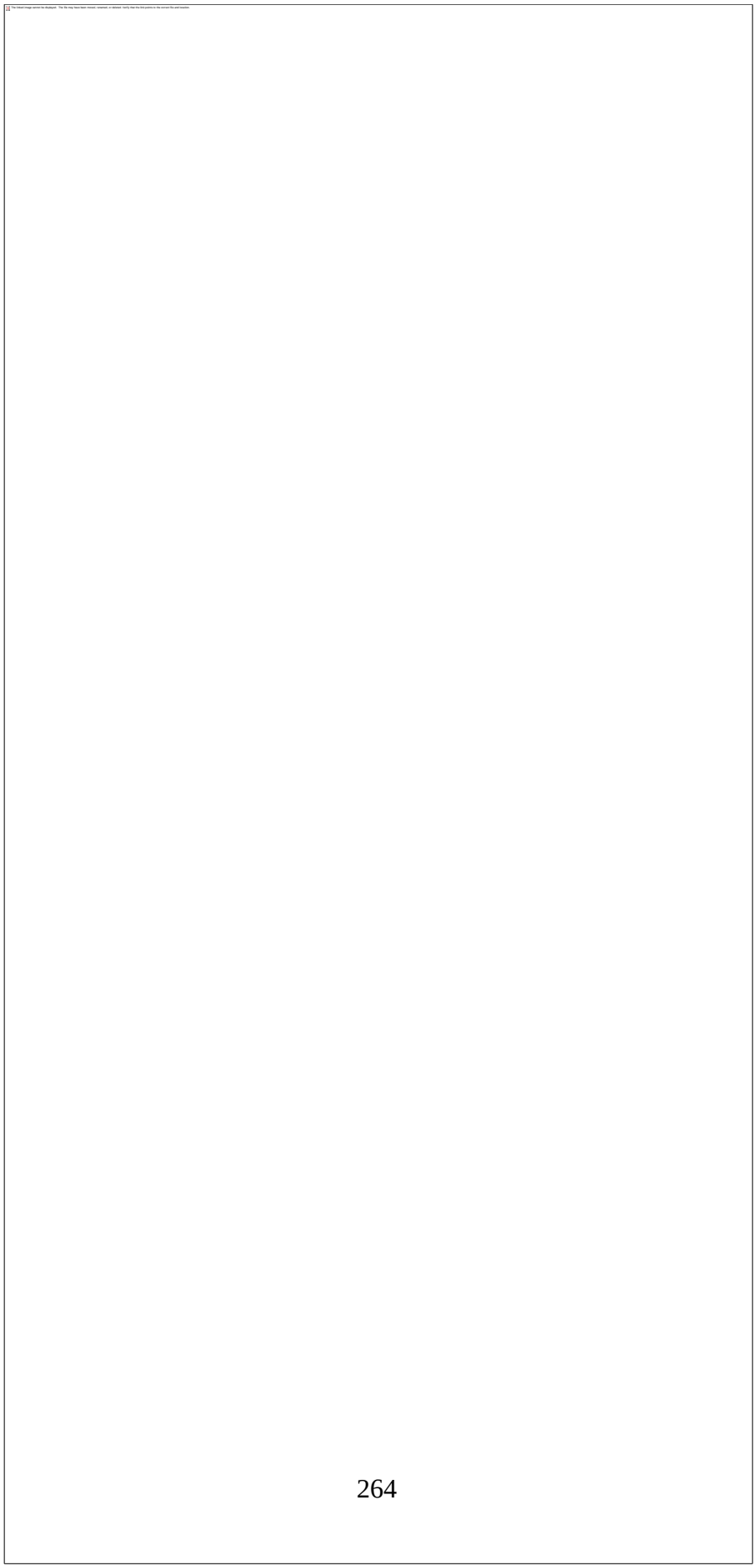
References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::geomCongruent(), grad(), i3j(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), ChebyCoeff::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), ChebyCoeff::state(), FlowField::xzstate(), and FlowField::ystate().

Referenced by navierstokesNL().

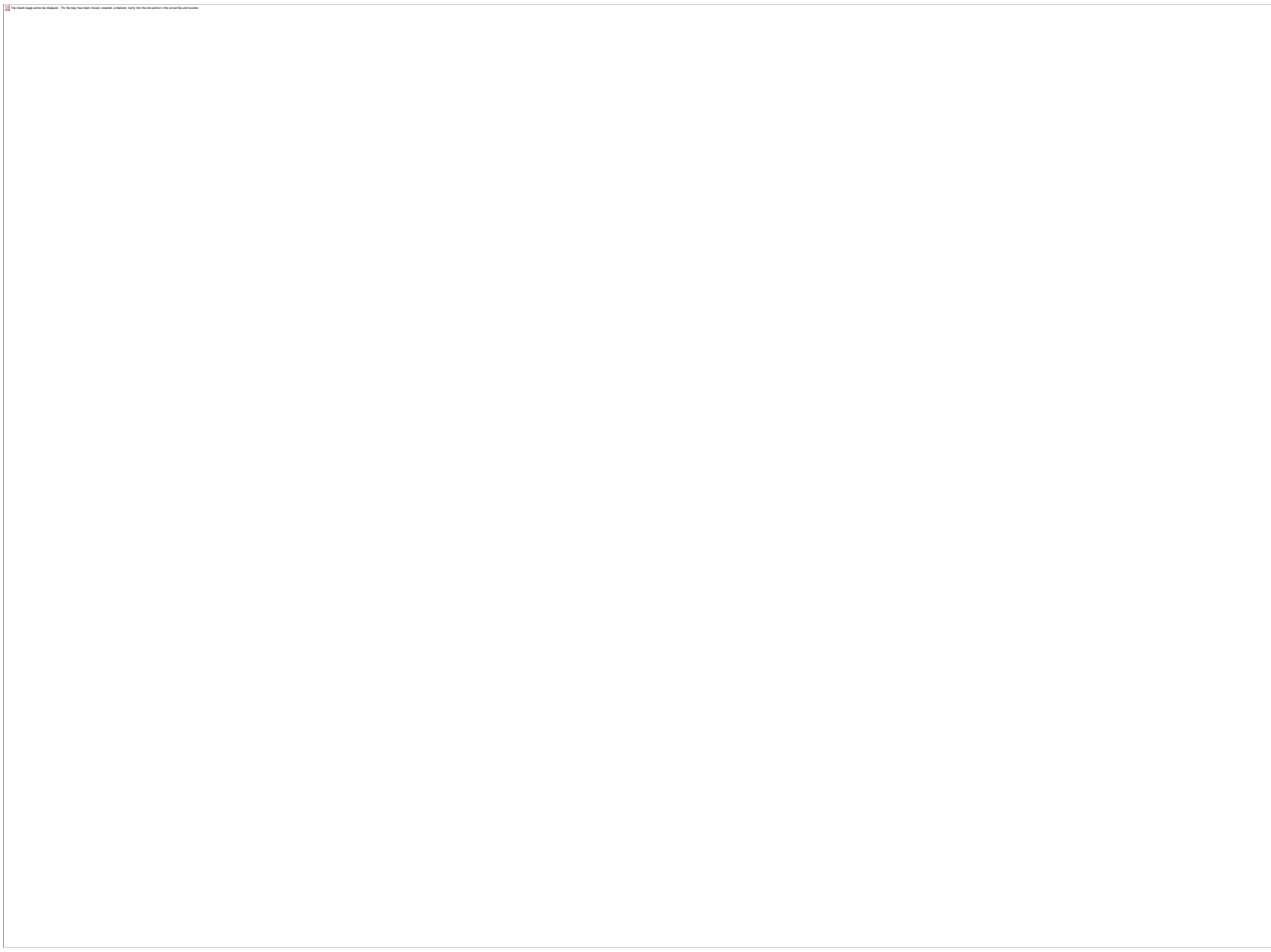
```
1031                                                    {
1032
1033  FlowField& u = (FlowField&) u_;
1034  ChebyCoeff& U = (ChebyCoeff&) U_;
1035  FlowField& grad_u = tmp;
1036
1037  assert(u.Nd() == 3);
1038  assert(U.numModes() == u.Ny());
1039  assert(U.a() == u.a() && U.b() == u.b());
1040
1041  fieldstate uxzstate = u.xzstate();
1042  fieldstate uystate  = u.ystate();
1043  fieldstate Ustate   = U.state();
1044
1045  if (!u.geomCongruent(f) || f.Nd() != 3)
1046    f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
1047  else
1048    f.setToZero();
1049  f.setState(Physical, Physical);
1050
```

```
1051   if (!u.geomCongruent(grad_u) || grad_u.Nd() < 9)
1052     grad_u.resize(u.Nx(), u.Ny(), u.Nz(), 9, u.Lx(), u.Lz(), u.a(), u.b());
1053
1054   U.makeSpectral();
1055   u.makeSpectral();
1056   u += U;
1057
1058   grad(u, grad_u);
1059   grad_u.makePhysical();
1060   u.makePhysical();
1061
1062   int Nx = u.Nx();
1063   int Ny = u.Ny();
1064   int Nz = u.Nz();
1065   for (int i=0; i<3; ++i)
1066     for (int j=0; j<3; ++j) {
1067       int ij = i3j(i,j);
1068       for (int ny=0; ny<Ny; ++ny)
1069         for (int nx=0; nx<Nx; ++nx)
1070           for (int nz=0; nz<Nz; ++nz)
1071             f(nx,ny,nz,i) += u(nx,ny,nz,j)*grad_u(nx,ny,nz,ij);
1072     }
1073
1074   f.makeSpectral();
1075   u.makeSpectral();
1076   u -= U;
1077
1078   // Possible extra transforms if entry states were Physical
1079   u.makeState(uxzstate, uystate);
1080   U.makeState(Ustate);
1081 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

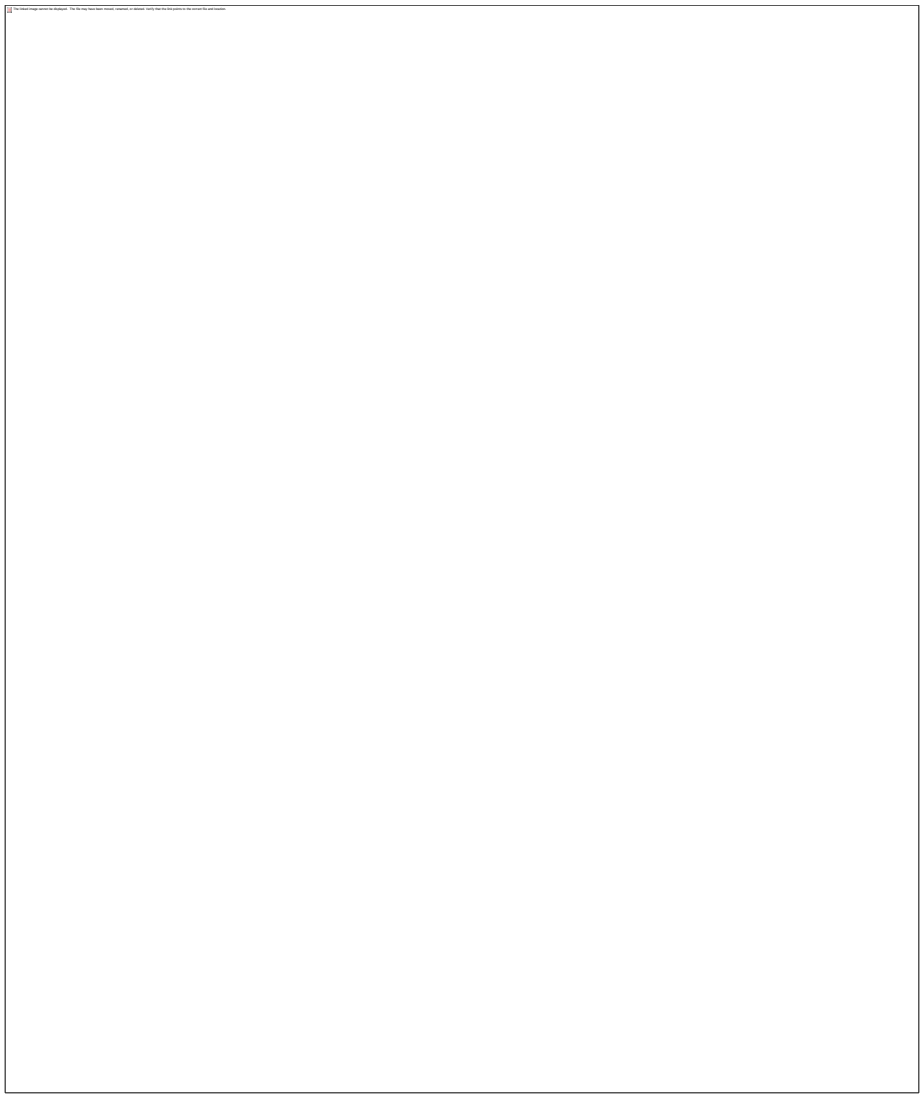


1.9.2.2 FlowField cross (const FlowField & *f*, const FlowField & *g*)

References cross().

```
859                                          {
860   FlowField fxg; cross(f,g,fxg); return fxg;
861 }
```

Here is the call graph for this function:



1.9.2.3 void cross (const FlowField & *f_*, const FlowField & *g_*, FlowField & *fcg*)

References FlowField::a(), FlowField::b(), FlowField::congruent(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), FlowField::xzstate(), and FlowField::ystate().

```
716                                                 {
717   FlowField& f = (FlowField&) f_;
718   FlowField& g = (FlowField&) g_;
719   fieldstate fxz = f.xzstate();
720   fieldstate fy  = f.ystate();
721   fieldstate gxz = g.xzstate();
722   fieldstate gy  = g.ystate();
```

```
723   assert(g.congruent(f));
724   assert(f.Nd() == 3 && g.Nd() == 3);
725
726   f.makePhysical();
727   g.makePhysical();
728
729   if (!f.geomCongruent(fcg) || fcg.Nd() != 3)
730     fcg.resize(f.Nx(), f.Ny(), f.Nz(), 3, f.Lx(), f.Lz(), f.a(), f.b());
731   else
732     fcg.setToZero();
733   fcg.setState(Physical,Physical);
734
735   int Nx = f.Nx();
736   int Ny = f.Ny();
737   int Nz = f.Nz();
738   int Nd = f.Nd();
739   for (int i=0; i<Nd; ++i)
740     for (int ny=0; ny<Ny; ++ny)
741       for (int nx=0; nx<Nx; ++nx)
742         for (int nz=0; nz<Nz; ++nz)
743           fcg(nx,ny,nz,i) =
744             f(nx,ny,nz,(i+1)%3)*g(nx,ny,nz,(i+2)%3) -
745             f(nx,ny,nz,(i+2)%3)*g(nx,ny,nz,(i+1)%3);
746
747   f.makeState(fxz, fy);
748   g.makeState(gxz, gy);
749   fcg.makeSpectral();
750 }
```

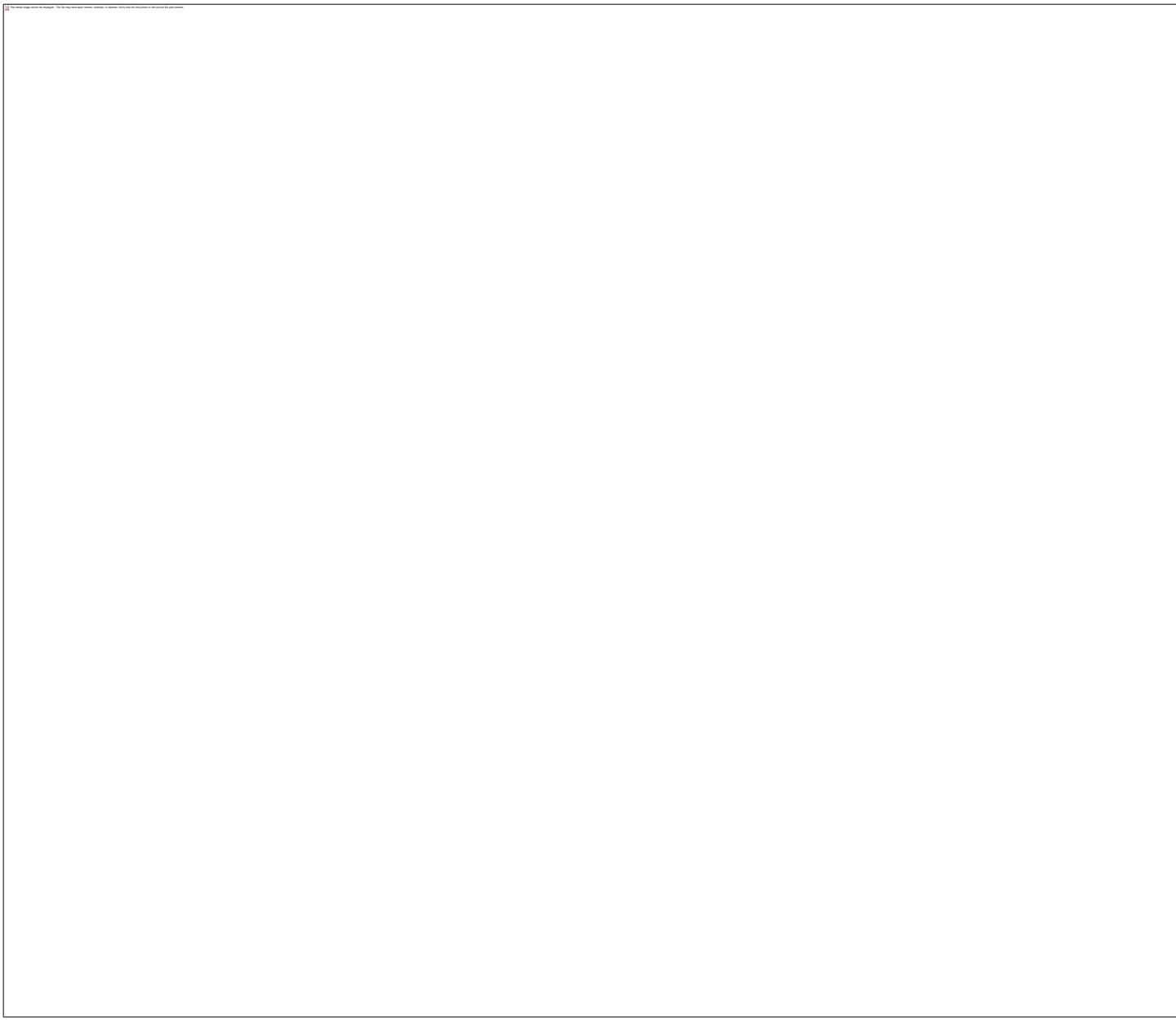
Here is the call graph for this function:

1.9.2.4 FlowField curl (const FlowField & *f*)

References curl().

```
851 {FlowField g;  curl(f,g); return g;}
```

Here is the call graph for this function:



1.9.2.5 void curl (const FlowField & *f_*, FlowField & *curlf*)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

```
489                                     {
490   FlowField& f = (FlowField&) f_;
491   assert(f.Nd() == 3);
492   fieldstate sxz = f.xzstate();
493   fieldstate sy  = f.ystate();
494   f.makeSpectral();
495
496   if (!f.geomCongruent(curlf) || curlf.vectorDim() != 3)
497     curlf.resize(f.Nx(), f.Ny(), f.Nz(), 3, f.Lx(), f.Lz(), f.a(), f.b());
498   else
499     curlf.setToZero();
500   curlf.setState(Spectral, Spectral);
501
502   int Mx = f.Mx();
503   int My = f.My();
504   int Mz = f.Mz();
505   int kxmax = f.kxmax();
506   int kzmax = f.kzmax();
507   Real Lx = f.Lx();
508   Real Lz = f.Lz();
509
510   // curl_x = w_y - v_z
511   // curl_y = u_z - w_x
512   // curl_z = v_x - u_y
513
514   // Assign d/dx and d/dz terms to curl
515   for (int my=0; my<My; ++my)
516     for (int mx=0; mx<Mx; ++mx) {
517       int kx = f.kx(mx);
518       Complex cx(0.0, 2.0*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
519       for (int mz=0; mz<Mz; ++mz) {
520         int kz = f.kz(mz);
521         Complex cz(0.0, 2.0*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
522         Complex u = f.cmplx(mx,my,mz,0);
523         Complex v = f.cmplx(mx,my,mz,1);
524         Complex w = f.cmplx(mx,my,mz,2);
525         curlf.cmplx(mx,my,mz,0) = -cz*v;
```

```
526         curlf.cmplx(mx,my,mz,1) = cz*u - cx*w;
527         curlf.cmplx(mx,my,mz,2) = cx*v;
528       }
529     }
530
531   // Assign df_i/dy to Df(i,1)
532   ComplexChebyCoeff w(f.Ny(),  f.a(), f.b(), Spectral);
533   ComplexChebyCoeff wy(f.Ny(), f.a(), f.b(), Spectral);
534   ComplexChebyCoeff u(f.Ny(),  f.a(), f.b(), Spectral);
535   ComplexChebyCoeff uy(f.Ny(), f.a(), f.b(), Spectral);
536   for (int mx=0; mx<Mx; ++mx)
537     for (int mz=0; mz<Mz; ++mz) {
538       for (int my=0; my<My; ++my) {
539         u.set(my, f.cmplx(mx,my,mz,0));
540         w.set(my, f.cmplx(mx,my,mz,2));
541       }
542       diff(u,uy);
543       diff(w,wy);
544       for (int my=0; my<My; ++my) {
545         curlf.cmplx(mx,my,mz,0) += wy[my];
546         curlf.cmplx(mx,my,mz,2) -= uy[my];
547       }
548     }
549   f.makeState(sxz, sy);
550 }
```

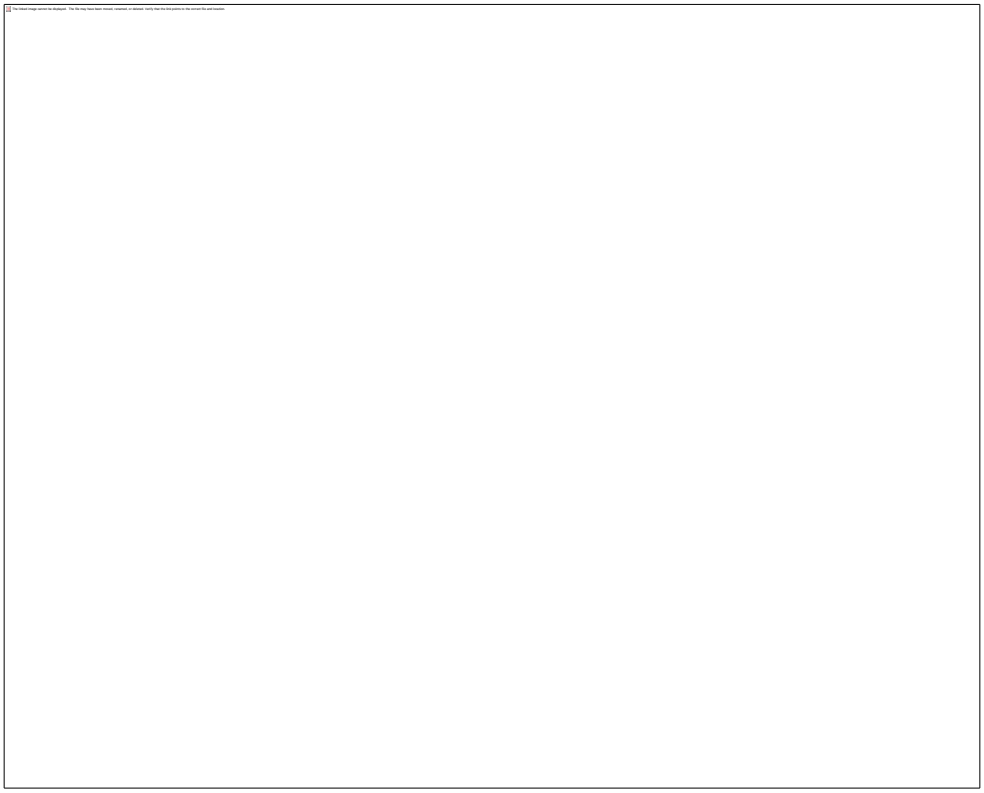
Here is the call graph for this function:

1.9.2.6 FlowField diff (const FlowField & *f*, int *nx*, int *ny*, int *nz*)

References diff().

```
843                                         {
844   FlowField g;
845   diff(f,g,nx,ny,nz);
846   return g;
847 }
```

Here is the call graph for this function:



1.9.2.7 void diff (const FlowField & *f_*, FlowField & *df*, int *nx*, int *ny*, int *nz*)

References FlowField::a(), FlowField::b(), cferror(), FlowField::cmplx(), FlowField::congruent(), diff(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, pow(), FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, swap(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

```
199                                                 {
200   FlowField& f = (FlowField&) f_;
201   fieldstate sxz = f.xzstate();
202   fieldstate sy  = f.ystate();
203   f.makeSpectral();
204
205   Real Lx = f.Lx();
206   Real Lz = f.Lz();
207
208   if (!f.congruent(df))
209     df.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
210   else
211     df.setToZero();
212   df.setState(Spectral, Spectral);
213
214   Complex xrot(0.0, 0.0);
215   switch(nx % 4) {
216   case 0: xrot = Complex( 1.0, 0.0); break;
217   case 1: xrot = Complex( 0.0, 1.0); break;
218   case 2: xrot = Complex(-1.0, 0.0); break;
219   case 3: xrot = Complex( 0.0,-1.0); break;
220   default: cferror("diff(f, df,nx,ny,nz) : impossible: nx % 4 > 4 !!"); break;
221   }
222   Complex zrot(0.0, 0.0);
223   switch(nz % 4) {
224   case 0: zrot = Complex( 1.0, 0.0); break;
225   case 1: zrot = Complex( 0.0, 1.0); break;
226   case 2: zrot = Complex(-1.0, 0.0); break;
227   case 3: zrot = Complex( 0.0,-1.0); break;
228   default: cferror("diff(f, df,nx,ny,nz) : impossible: nz % 4 > 4 !!"); break;
229   }
230
231   int Nd = f.Nd();
232   int Mx = f.Mx();
233   int My = f.My();
234   int Mz = f.Mz();
235   int kxmax = f.kxmax();
236   int kzmax = f.kzmax();
237
238   // Do the x and z differentiation
239   Real ax = 2*pi/Lx;
240   Real az = 2*pi/Lz;
241   for (int i=0; i<Nd; ++i)
242     for (int my=0; my<My; ++my)
243       for (int mx=0; mx<Mx; ++mx) {
244         int kx = f.kx(mx);
```

```
245       Complex cx = xrot*(pow(ax*kx, nx)*zero_last_mode(kx,kxmax,nx));
246       for (int mz=0; mz<Mz; ++mz) {
247         int kz = f.kz(mz);
248         Complex cz = zrot*(pow(az*kz, nz)*zero_last_mode(kz,kzmax,nz));
249         df.cmplx(mx,my,mz,i) = cx*cz*f.cmplx(mx,my,mz,i);
250       }
251     }
252
253  // Do the y differentiation
254  ComplexChebyCoeff  g(f.Ny(), f.a(), f.b(), Spectral);
255  ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
256
257  for (int i=0; i<Nd; ++i)
258    for (int mx=0; mx<Mx; ++mx)
259      for (int mz=0; mz<Mz; ++mz) {
260        for (int my=0; my<My; ++my)
261          g.set(my, df.cmplx(mx,my,mz,i));
262
263        // differentiate n times and leave derivative in *g
264        for (int k=0; k<ny; ++k) {
265          diff(g,gy);
266          swap(g,gy);
267        }
268        for (int my=0; My<My; ++my)
269          df.cmplx(mx,my,mz,i) = g[my];
270      }
271
272  f.makeState(sxz,sy);
273  df.makeSpectral();
274  return;
275 }
```

Here is the call graph for this function:

1.9.2.8 FlowField div (const FlowField & *f*)

References div().

```
854 {FlowField g;   div(f,g); return g;}
```

Here is the call graph for this function:

1.9.2.9 void div (const FlowField & *f_*, FlowField & *divf*)

References FlowField::a(), FlowField::b(), cferror(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), i3j(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

Referenced by chebyProfile(), div(), divDist2(), divergence(), divergenceNL(), divNorm2(), randomProfile(), FlowField::saveDivSpectrum(), PressureSolver::solve(), and PressureSolver::verify().

```
626                                     {
627   FlowField& f = (FlowField&) f_;
628   fieldstate sxz = f.xzstate();
629   fieldstate sy  = f.ystate();
630   f.makeSpectral();
631
632   int Mx = f.Mx();
633   int My = f.My();
634   int Mz = f.Mz();
635   int kxmax = f.kxmax();
636   int kzmax = f.kzmax();
637   Real Lx = f.Lx();
638   Real Lz = f.Lz();
639
640   if (f.Nd() == 3) {
641     if (!f.geomCongruent(divf) || divf.Nd() != 1)
642       divf.resize(f.Nx(), f.Ny(), f.Nz(), 1, f.Lx(), f.Lz(), f.a(), f.b());
643     else
644       divf.setToZero();
645     divf.setState(Spectral, Spectral);
646
647     // Add df0/dx + df2/dz to divf
648     for (int my=0; my<My; ++my)
649       for (int mx=0; mx<Mx; ++mx) {
650         int kx = f.kx(mx);
651         Complex cxkx(0.0, 2.0*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
652         for (int mz=0; mz<Mz; ++mz) {
653           int kz = f.kz(mz);
654           Complex czkz(0.0, 2.0*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
655           divf.cmplx(mx,my,mz,0)
656             = f.cmplx(mx,my,mz,0)*cxkx + f.cmplx(mx,my,mz,2)*czkz;
657         }
```

```
658       }
659
660     // Add df1/dy to to divf
661     ComplexChebyCoeff g(f.Ny(),  f.a(), f.b(), Spectral);
662     ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
663     for (int mx=0; mx<Mx; ++mx)
664       for (int mz=0; mz<Mz; ++mz) {
665         for (int my=0; my<My; ++my)
666           g.set(my, f.cmplx(mx,my,mz,1));
667         diff(g,gy);
668         for (int my=0; my<My; ++my)
669           divf.cmplx(mx,my,mz,0) += gy[my];
670       }
671   }
672   else if (f.Nd() == 9) {
673     if (!f.geomCongruent(divf) || divf.Nd() != 3)
674       divf.resize(f.Nx(), f.Ny(), f.Nz(), 3, f.Lx(), f.Lz(), f.a(), f.b());
675     else
676       divf.setToZero();
677     divf.setState(Spectral, Spectral);
678
679     for (int j=0; j<3; ++j) {
680
681       // Add df0j/dx + df2j/dz to divfj
682       for (int my=0; my<My; ++my)
683         for (int mx=0; mx<Mx; ++mx) {
684           int kx = f.kx(mx);
685           Complex cxkx(0.0, 2.0*pi*f.kx(mx)/Lx*zero_last_mode(kx,kxmax,1));
686           for (int mz=0; mz<Mz; ++mz) {
687             int kz = f.kz(mz);
688             Complex czkz(0.0, 2.0*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
689               divf.cmplx(mx,my,mz,j) +=
690                 f.cmplx(mx,my,mz,i3j(0,j))*cxkx +
691                 f.cmplx(mx,my,mz,i3j(2,j))*czkz;
692           }
693         }
694
695       // Add df1j/dy to to divf
696       ComplexChebyCoeff g(f.Ny(),  f.a(), f.b(), Spectral);
697       ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
698       for (int mx=0; mx<Mx; ++mx)
699         for (int mz=0; mz<Mz; ++mz) {
700           for (int my=0; my<My; ++my)
701             g.set(my, f.cmplx(mx,my,mz,i3j(1,j)));
702           diff(g,gy);
703           for (int my=0; my<My; ++my)
```

```
704             divf.cmplx(mx,my,mz,j) += gy[my];
705           }
706       }
707     }
708     else
709       cferror("div(FlowField f, FlowField divf): f must be 3d or 9d");
710
711     f.makeState(sxz, sy);
712     divf.makeSpectral();
713     return;
714   }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.9.2.10 void divergenceNL (const FlowField & *u_*, const ChebyCoeff & *U_*, FlowField & *f*, FlowField & *tmp*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), div(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), outer(), FlowField::resize(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), and FlowField::ystate().

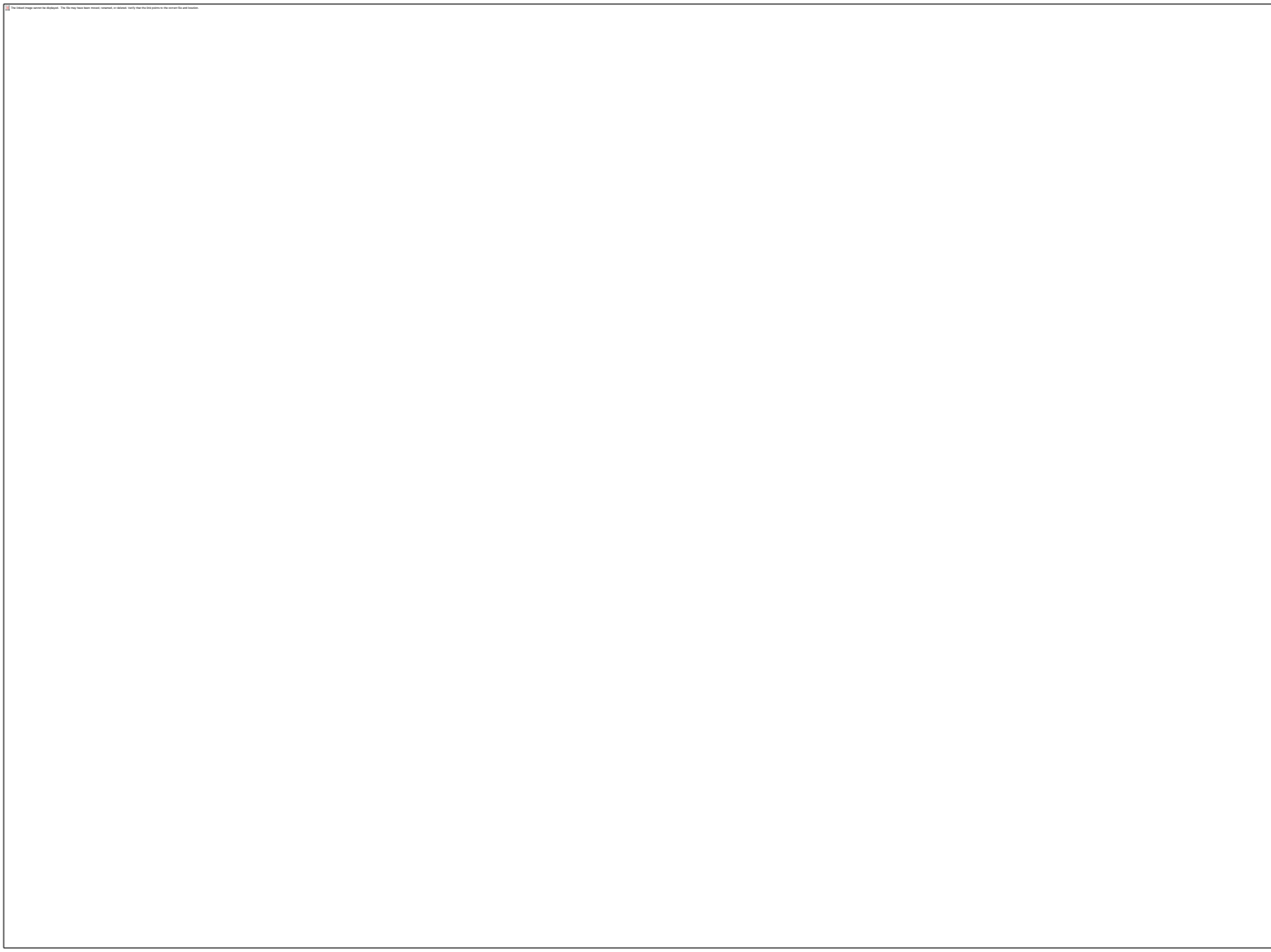
Referenced by navierstokesNL().

```
1083                                                    {
1084   FlowField& u = (FlowField&) u_;
1085   ChebyCoeff& U = (ChebyCoeff&) U_;
1086   FlowField& uu = tmp;
1087 
1088   assert(u.Nd() == 3);
1089   assert(U.numModes() == u.Ny());
1090   assert(U.a() == u.a() && U.b() == u.b());
1091 
1092   fieldstate uxzstate = u.xzstate();
1093   fieldstate uystate  = u.ystate();
1094   fieldstate Ustate   = U.state();
1095 
1096   if (!u.geomCongruent(f) || f.Nd() != 3)
1097     f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
```

```
1098
1099   if (!u.geomCongruent(uu) || uu.Nd() < 9)
1100     uu.resize(u.Nx(), u.Ny(), u.Nz(), 9, u.Lx(), u.Lz(), u.a(), u.b());
1101
1102   // Possible extra transforms if entry states are Physical
1103   u.makeState(Spectral, uystate);
1104   U.makeState(uystate);
1105   u += U;
1106
1107   u.makePhysical();
1108   outer(u,u, uu);
1109
1110   uu.makeSpectral();
1111   div(uu, f);         // produces f spectral
1112
1113   u.makeState(Spectral, uystate);
1114   u -= U;
1115
1116   // Possible extra transforms if entry states are Physical
1117   u.makeState(uxzstate, uystate);
1118   U.makeState(Ustate);
1119
1120   return;
1121 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.9.2.11 FlowField dot (const FlowField & *f*, const FlowField & *g*)

References dot().

```
862                                         {
863   FlowField fdotg; dot(f,g,fdotg); return fdotg;
864 }
```

Here is the call graph for this function:

1.9.2.12 void dot (const FlowField & *f_*, const FlowField & *g_*, FlowField & *fdotg*)

References FlowField::a(), FlowField::b(), FlowField::congruent(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), FlowField::xzstate(), and FlowField::ystate().

```
591                                                 {
592   FlowField& f = (FlowField&) f_;
593   FlowField& g = (FlowField&) g_;
594   fieldstate fxz = f.xzstate();
595   fieldstate fy  = f.ystate();
596   fieldstate gxz = g.xzstate();
597   fieldstate gy  = g.ystate();
598
599   assert(f.congruent(g));
600   f.makePhysical();
601   g.makePhysical();
602
603   if (!f.geomCongruent(fdotg) || fdotg.Nd() != 1)
604     fdotg.resize(f.Nx(), f.Ny(), f.Nz(), 1, f.Lx(), f.Lz(), f.a(),f.b());
605   else
606     fdotg.setToZero();
607   fdotg.setState(Physical, Physical);
608
```

```
609   int Nx = f.Nx();
610   int Ny = f.Ny();
611   int Nz = f.Nz();
612   int Nd = f.Nd();
613
614   for (int i=0; i<Nd; ++i)
615     for (int ny=0; ny<Ny; ++ny)
616       for (int nx=0; nx<Nx; ++nx)
617         for (int nz=0; nz<Nz; ++nz)
618           fdotg(nx,ny,nz,0) += f(nx,ny,nz,i)*g(nx,ny,nz,i);
619
620   f.makeState(fxz, fy);
621   g.makeState(gxz, gy);
622   fdotg.makeSpectral();
623   return;
624 }
```

Here is the call graph for this function:

1.9.2.13 FlowField energy (const FlowField & *u*, ChebyCoeff & *U*)

References energy().

```
868                                         {
869   FlowField e; energy(u,U,e); return e;
870 }
```

Here is the call graph for this function:

1.9.2.14 FlowField energy (const FlowField & *u*)

References energy().

```
865                                 {
866   FlowField e; energy(u,e); return e;
867 }
```

Here is the call graph for this function:

1.9.2.15 void energy (const FlowField & *u_*, const ChebyCoeff & *U_*, FlowField & *e*)

References FlowField::a(), FlowField::b(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), ChebyCoeff::makePhysical(), FlowField::makePhysical(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), square(), ChebyCoeff::state(), FlowField::xzstate(), and FlowField::ystate().

```
781                                                 {
782
783   FlowField& u = (FlowField&) u_;
784   ChebyCoeff& U = (ChebyCoeff&) U_;
785   fieldstate uxzstate = u.xzstate();
786   fieldstate uystate  = u.ystate();
787   fieldstate Ustate   = U.state();
788   assert(U.numModes() == u.Ny());
789
790   if (!u.geomCongruent(e) || e.Nd() != 1)
791     e.resize(u.Nx(), u.Ny(), u.Nz(), 1, u.Lx(), u.Lz(), u.a(), u.b());
792   else
793     e.setToZero();
794   e.setState(Physical,Physical);
795
796   u.makePhysical();
797   U.makePhysical();
798
799   e.setToZero();
800   e.setState(Physical, Physical);
801
802   int Nx = u.Nx();
803   int Ny = u.Ny();
804   int Nz = u.Nz();
805   int Nd = u.Nd();
806
807   for (int ny=0; ny<Ny; ++ny)
808     for (int nx=0; nx<Nx; ++nx)
809       for (int nz=0; nz<Nz; ++nz)
810         e(nx,ny,nz,0) += 0.5*square(u(nx,ny,nz,0)+U(ny));
811
812   for (int i=1; i<Nd; ++i)
813     for (int ny=0; ny<Ny; ++ny)
814       for (int nx=0; nx<Nx; ++nx)
815         for (int nz=0; nz<Nz; ++nz)
```

```
816           e(nx,ny,nz,0) += 0.5*square(u(nx,ny,nz,i));
817
818    e.makeSpectral();
819    u.makeState(uxzstate,uystate);
820    U.makeState(Ustate);
821
822    return;
823  }
```

Here is the call graph for this function:

1.9.2.16 void energy (const FlowField & *u_*, FlowField & *e*)

References FlowField::a(), FlowField::b(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), square(), FlowField::xzstate(), and FlowField::ystate().

Referenced by energy().

```
752                                     {
753   FlowField& u = (FlowField&) u_;
754   fieldstate sxz = u.xzstate();
755   fieldstate sy  = u.ystate();
756   u.makePhysical();
757
758   if (!u.geomCongruent(e) || e.Nd() != 1)
759     e.resize(u.Nx(), u.Ny(), u.Nz(), 1, u.Lx(), u.Lz(), u.a(), u.b());
760   else
761     e.setToZero();
762   e.setState(Physical,Physical);
763
764   int Nx = u.Nx();
765   int Ny = u.Ny();
766   int Nz = u.Nz();
767   int Nd = u.Nd();
768
769   for (int i=0; i<Nd; ++i)
770     for (int ny=0; ny<Ny; ++ny)
771       for (int nx=0; nx<Nx; ++nx)
772         for (int nz=0; nz<Nz; ++nz)
773           e(nx,ny,nz,0) += 0.5*square(u(nx,ny,nz,i));
774
775   e.makeSpectral();
776   u.makeState(sxz, sy);
777   return;
778 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.9.2.17 FlowField grad (const FlowField & *f*)

References grad().

```
849 {FlowField g;  grad(f,g); return g;}
```

Here is the call graph for this function:

1.9.2.18 void grad (const FlowField & *f_*, FlowField & *gradf*)

References FlowField::a(), FlowField::b(), cferror(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), i3j(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

Referenced by PPEDNS::advance(), convectionNL(), grad(), gradient(), PPEDNS::PPEDNS(), PPEDNS::push(), skewsymmetricNL(), skewsymmetricNL_GPU(), and skewsymmetricNL_THREAD().

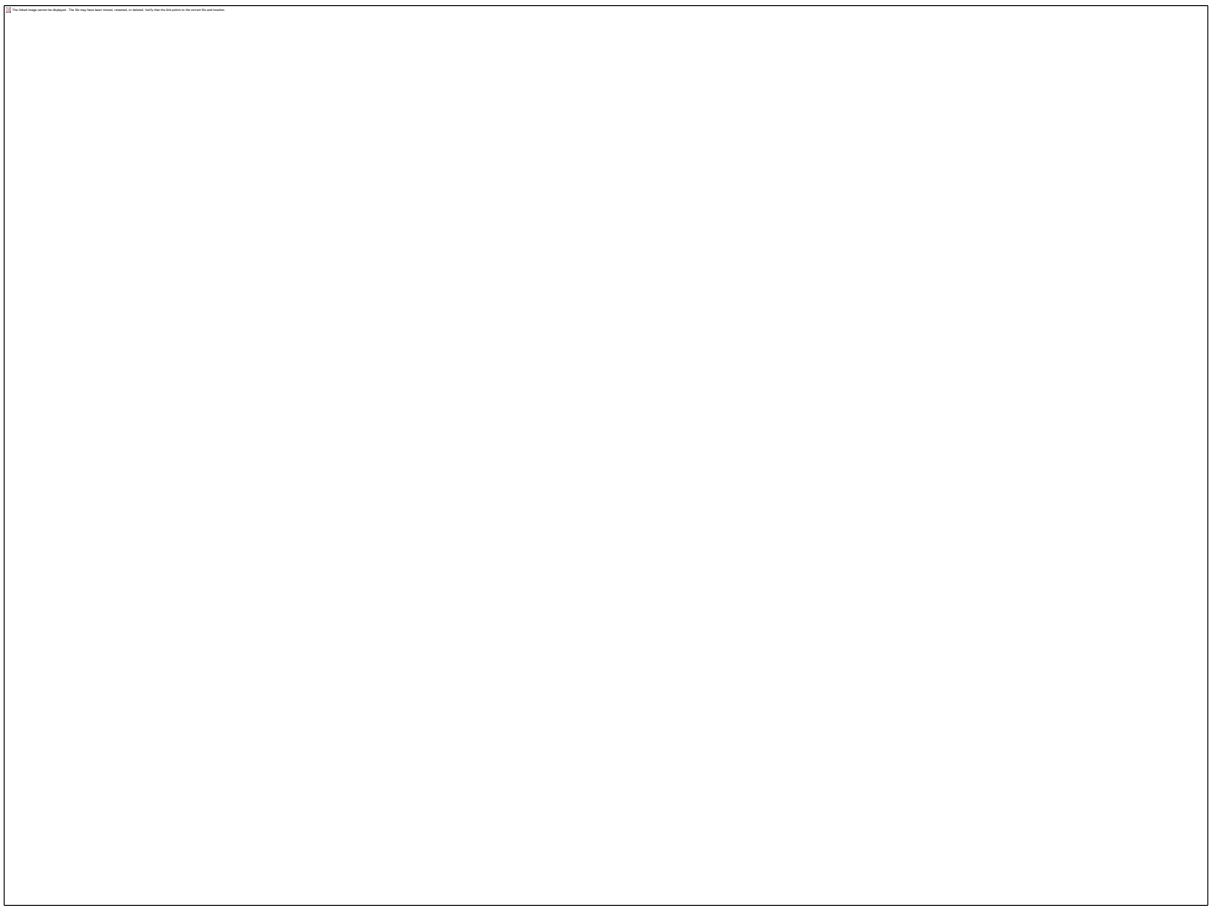
```
280                                     {
281
282   FlowField& f = (FlowField&) f_;
283   fieldstate sxz = f.xzstate();
284   fieldstate sy  = f.ystate();
285   f.makeSpectral();
286
287   Real Lx = f.Lx();
288   Real Lz = f.Lz();
289
290   // compute gradf(nx,ny,nz,i) = df(nx,ny,nz,0)/dx_i for scalar-valued f
291   if (f.Nd() == 1) {
292     if (!f.geomCongruent(gradf) || gradf.vectorDim() != 3)
293       gradf.resize(f.Nx(), f.Ny(), f.Nz(), 3, f.Lx(), f.Lz(), f.a(), f.b());
294     else
295       gradf.setToZero();
296     gradf.setState(Spectral, Spectral);
297
298     int Mx = f.Mx();
299     int My = f.My();
300     int Mz = f.Mz();
301     int kxmax = f.kxmax();
302     int kzmax = f.kzmax();
303
304     // Assign df/dx to Df(i) and df/dz to Df(i)
305     for (int my=0; my<My; ++my)
306       for (int mx=0; mx<Mx; ++mx) {
307         int kx = f.kx(mx);
308         Complex cxkx(0.0, 2.0*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
309         for (int mz=0; mz<Mz; ++mz) {
310           int kz = f.kz(mz);
311           Complex czkz(0.0, 2.0*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
```

```
312         Complex fval = f.cmplx(mx,my,mz,0);
313         gradf.cmplx(mx,my,mz,0) = fval*cxkx;
314         gradf.cmplx(mx,my,mz,2) = fval*czkz;
315       }
316     }
317
318   // Assign df/dy to Df(i,1)
319   ComplexChebyCoeff g(f.Ny(),  f.a(), f.b(), Spectral);
320   ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
321   for (int mx=0; mx<Mx; ++mx)
322     for (int mz=0; mz<Mz; ++mz) {
323       for (int my=0; my<My; ++my)
324         g.set(my, f.cmplx(mx,my,mz,0));
325       diff(g,gy);
326       for (int my=0; my<My; ++my)
327         gradf.cmplx(mx,my,mz,1) = gy[my];
328     }
329 }
330 else if (f.Nd() == 3) {
331   if (!f.geomCongruent(gradf) || gradf.vectorDim() != 9)
332     gradf.resize(f.Nx(), f.Ny(), f.Nz(), 9, f.Lx(), f.Lz(), f.a(), f.b());
333   else
334     gradf.setToZero();
335   gradf.setState(Spectral, Spectral);
336
337   int Mx = f.Mx();
338   int My = f.My();
339   int Mz = f.Mz();
340   int Nd = 3;
341   int kxmax = f.kxmax();
342   int kzmax = f.kzmax();
343
344   // Assign df_i/dx to Df(i,0) and df_i/dz to Df(i,2)
345   for (int i=0; i<Nd; ++i)
346     for (int my=0; my<My; ++my)
347       for (int mx=0; mx<Mx; ++mx) {
348         int kx = f.kx(mx);
349         Complex cxkx(0.0, 2.0*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
350         for (int mz=0; mz<Mz; ++mz) {
351           int kz = f.kz(mz);
352           Complex czkz(0.0, 2.0*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
353           Complex fval = f.cmplx(mx,my,mz,i);
354           gradf.cmplx(mx,my,mz,i3j(i,0)) = fval*cxkx;
355           gradf.cmplx(mx,my,mz,i3j(i,2)) = fval*czkz;
356         }
357       }
```

```
358
359   // Assign df_i/dy to Df(i,1)
360   ComplexChebyCoeff g(f.Ny(),  f.a(), f.b(), Spectral);
361   ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
362   for (int i=0; i<Nd; ++i)
363     for (int mx=0; mx<Mx; ++mx)
364       for (int mz=0; mz<Mz; ++mz) {
365         for (int my=0; my<My; ++my)
366           g.set(my, f.cmplx(mx,my,mz,i));
367         diff(g,gy);
368         for (int my=0; my<My; ++my)
369           gradf.cmplx(mx,my,mz,i3j(i,1)) = gy[my];
370       }
371   }
372   else
373     cferror("grad(FlowField f, FlowField grad_f) : f must be 1d or 3d");
374
375   f.makeState(sxz, sy);
376   return;
377 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.9.2.19 FlowField lapl (const FlowField & *f*)

References lapl().

```
850 {FlowField g;  lapl(f,g); return g;}
```

Here is the call graph for this function:

1.9.2.20 void lapl (const FlowField & *f_*, FlowField & *laplf*)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), diff2(), FlowField::kx(), FlowField::kz(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, square(), FlowField::xzstate(), and FlowField::ystate().

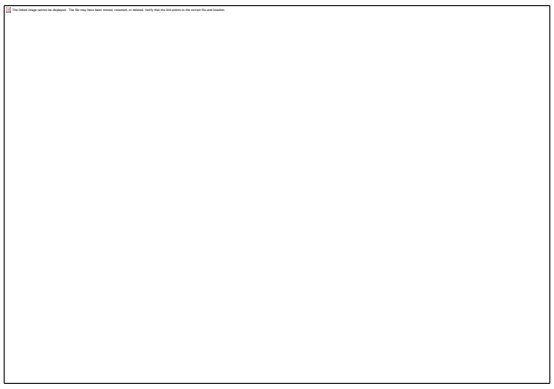
Referenced by lapl(), laplacian(), PressureSolver::verify(), and PoissonSolver::verify().

```
379                                     {
380   FlowField& f = (FlowField&) f_;
381   fieldstate sxz = f.xzstate();
382   fieldstate sy  = f.ystate();
383   f.makeSpectral();
384
385   if (!f.congruent(laplf))
386     laplf.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
387   else
388     laplf.setToZero();
389   laplf.setState(Spectral, Spectral);
390
391   // compute gradf(nx,ny,nz,i) = df(nx,ny,nz,0)/dx_i for scalar-valued f
392   Real Lx = f.Lx();
393   Real Lz = f.Lz();
394   int Mx = f.Mx();
395   int My = f.My();
396   int Mz = f.Mz();
397   int Nd = f.Nd();
398
399   ComplexChebyCoeff g(f.Ny(),  f.a(), f.b(), Spectral);
400   ComplexChebyCoeff gyy(f.Ny(), f.a(), f.b(), Spectral);
401
402   Real az = -1.0*square(2.0*pi/Lz);
403   for (int i=0; i<Nd; ++i) {
404     // Assign lapl f_i = (d^2/dx^2 + d^2/dz^2) f_i
405     for (int my=0; my<My; ++my)
406       for (int mx=0; mx<Mx; ++mx) {
407         int kx = f.kx(mx);
408         Complex cx = -1.0*square(2.0*pi*kx/Lx);
409         for (int mz=0; mz<Mz; ++mz)
410           laplf.cmplx(mx,my,mz,i) =
411             (cx + az*square(f.kz(mz)))*f.cmplx(mx,my,mz,i);
412       }
413
```

```
414     // Add d^2/dy^2 f_i on to previous result
415     for (int mx=0; mx<Mx; ++mx)
416       for (int mz=0; mz<Mz; ++mz) {
417         for (int my=0; my<My; ++my)
418           g.set(my, f.cmplx(mx,my,mz,i));
419         diff2(g,gyy);
420         for (int my=0; my<My; ++my)
421           laplf.cmplx(mx,my,mz,i) += gyy[my];
422       }
423   }
424   f.makeState(sxz,sy);
425 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.9.2.21 void linearizedNL (const FlowField & *u_*, const ChebyCoeff & *U_*, FlowField & *f*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), FlowField::kx(), FlowField::kxmax(), FlowField::Lx(), FlowField::Lz(), ChebyCoeff::makePhysical(), FlowField::makeSpectral(), ChebyCoeff::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Mx(), FlowField::Mz(), ChebyCoeff::N(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

Referenced by navierstokesNL().

```
1530                                                {
1531   FlowField& u = (FlowField&) u_;
1532   ChebyCoeff& U = (ChebyCoeff&) U_;
1533
1534   assert(u.Nd() == 3);
1535   assert(U.N() == u.Ny());
1536   assert(U.a() == u.a() && U.b() == u.b());
1537
1538   fieldstate uxzstate = u.xzstate();
1539   fieldstate uystate  = u.ystate();
1540   fieldstate Ustate   = U.state();
1541
1542   if (!u.geomCongruent(f) || f.Nd() != 3)
1543     f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
1544   else
1545     f.setToZero();
1546   f.setState(Spectral, Physical);
1547
```

```
1548   ChebyTransform trans(U.N());
1549   U.makeSpectral(trans);
1550   ChebyCoeff Uy;
1551   diff(U,Uy);
1552   U.makePhysical(trans);
1553   Uy.makePhysical(trans);
1554
1555   Complex cu;
1556   Complex cv;
1557   Complex cw;
1558
1559   int Mx = u.Mx();
1560   int Ny = u.Ny();
1561   int Mz = u.Mz();
1562   Real Lx = u.Lx();
1563   int kxmax = u.kxmax();
1564
1565   u.makeState(Spectral, Physical);
1566   for (int ny=0; ny<Ny; ++ny)
1567     for (int mx=0; mx<Mx; ++mx) {
1568       int kx = u.kx(mx);
1569       Complex d_dx(0.0, 2*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
1570       for (int mz=0; mz<Mz; ++mz) {
1571         cu = u.cmplx(mx,ny,mz, 0);
1572         cv = u.cmplx(mx,ny,mz, 1);
1573         cw = u.cmplx(mx,ny,mz, 2);
1574
1575         f.cmplx(mx,ny,mz, 0) = U[ny]*d_dx*cu + cv*Uy[ny];
1576         f.cmplx(mx,ny,mz, 1) = U[ny]*d_dx*cv;
1577         f.cmplx(mx,ny,mz, 2) = U[ny]*d_dx*cw;
1578       }
1579     }
1580
1581   f.makeSpectral();
1582   u.makeState(uxzstate,uystate);
1583   U.makeState(Ustate);
1584   return;
1585 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.9.2.22 void navierstokesNL (const FlowField & *u*, const ChebyCoeff & *U*, FlowField & *f*, FlowField & *tmp*, NonlinearMethod & *method*)

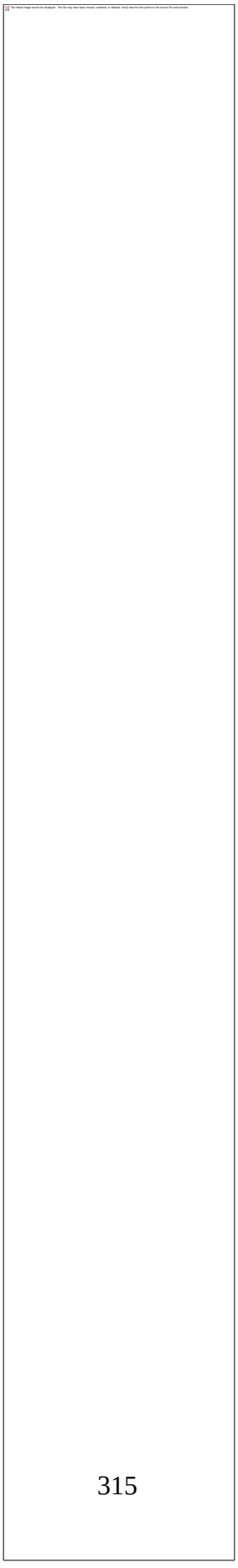
References Alternating, Alternating_, cferror(), Convection, convectionNL(), CPU, Divergence, divergenceNL(), cmndline::GetInstance(), cmndline::getRun(), GPU, Linearized, linearizedNL(), Rotational, rotationalNL(), SkewSymmetric, skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), and THREAD.

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), MultistepDNS::MultistepDNS(), PPEDNS::PPEDNS(), PPEDNS::push(), CNABstyleDNS::push(), MultistepDNS::push(), PressureSolver::solve(), and PressureSolver::verify().

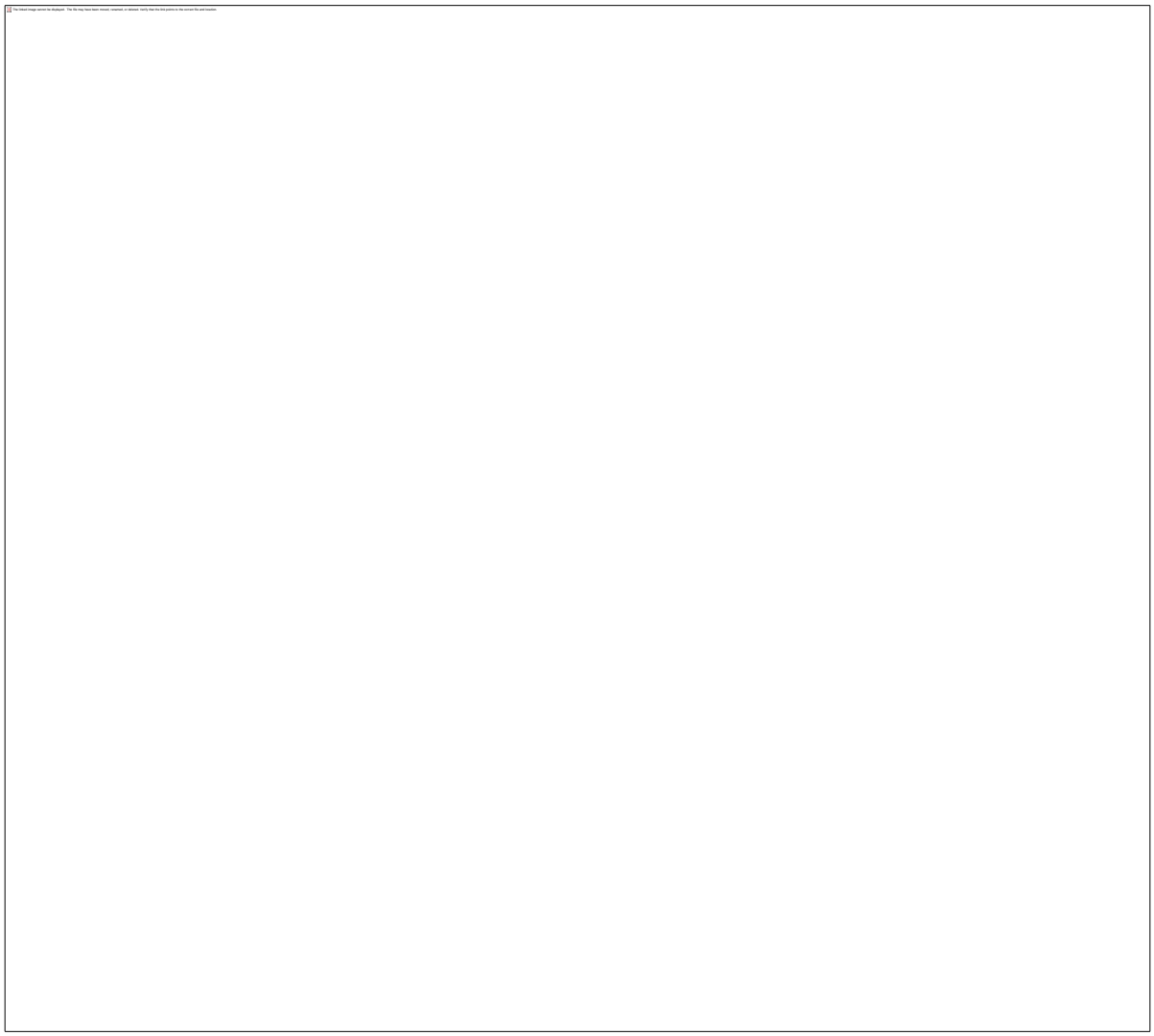
```
873                                         {
874
875 //IG
876   cmndline *cmd = cmndline::GetInstance();
877   //u.print();
```

```
878   //cmd->show();
879   //cout << "Run=" << cmd->getRun() << endl;
880   HowToRun howToRun = cmd->getRun();
881 //IG-e
882
883
884   switch (method) {
885   case Rotational:
886     rotationalNL(u,U,f,tmp);
887     break;
888   case Convection:
889     convectionNL(u,U,f,tmp);
890     break;
891   case SkewSymmetric:
892     if(howToRun == CPU) skewsymmetricNL(u,U,f,tmp);
893     if(howToRun == GPU) skewsymmetricNL_GPU(u,U,f,tmp);
894     if(howToRun == THREAD) skewsymmetricNL_THREAD(u,U,f,tmp);
895     break;
896   case Divergence:
897     divergenceNL(u,U,f,tmp);
898     break;
899   case Alternating:
900     divergenceNL(u,U,f,tmp);
901     method = Alternating_;
902     break;
903   case Alternating_:
904     convectionNL(u,U,f,tmp);
905     method = Alternating;
906     break;
907   case Linearized:
908     linearizedNL(u,U,f); break;
909   default:
910     cferror("navierstokesNL(method, u,U,f,tmp,parity) : unknown method");
911   }
912 }
```

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.

Here is the caller graph for this function:



1.9.2.23 FlowField norm (const FlowField & *f*)

References norm().

```
852 {FlowField g;  norm(f,g); return g;}
```

Here is the call graph for this function:

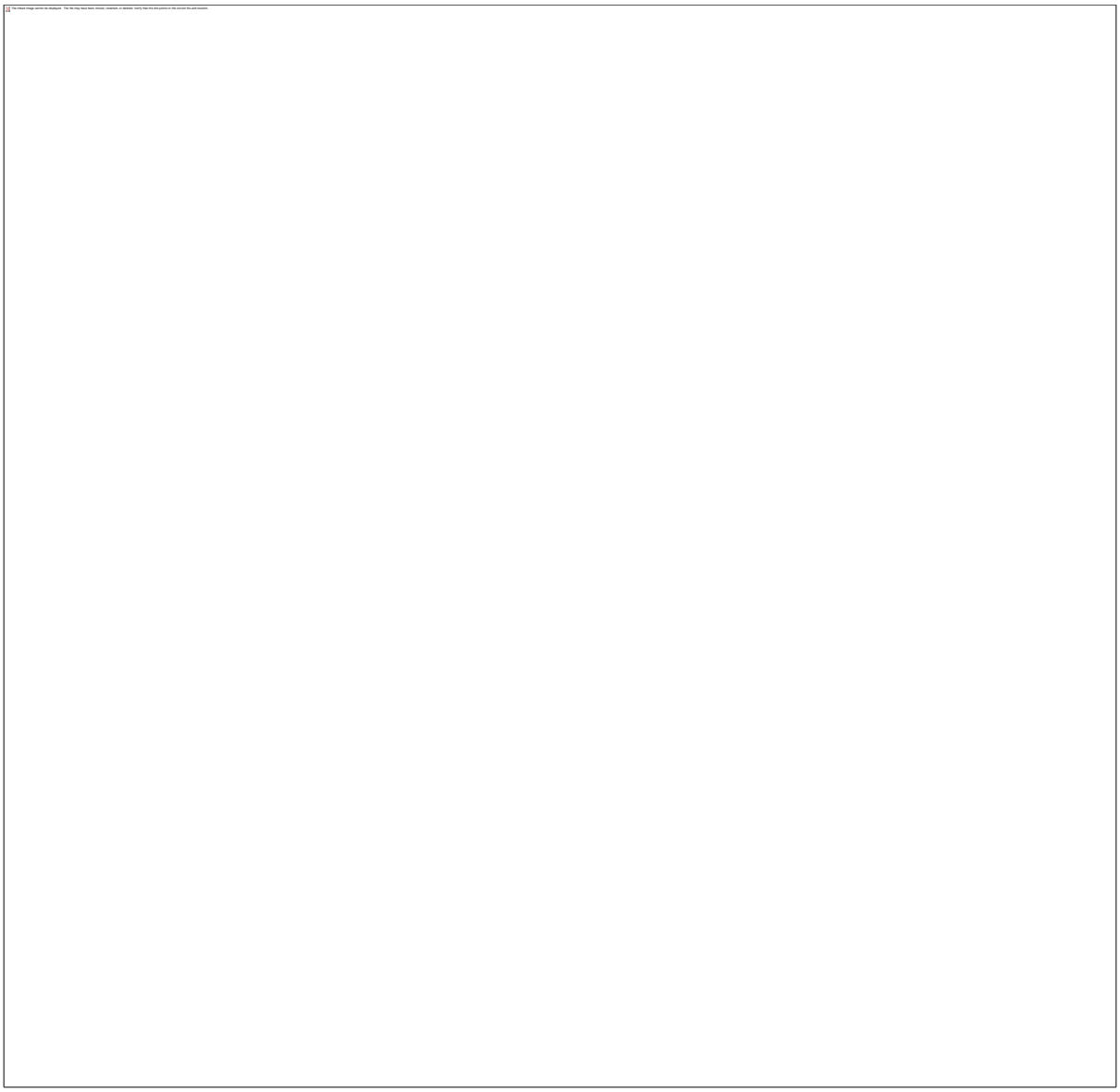
1.9.2.24 void norm (const FlowField & *f_*, FlowField & *normf*)

References FlowField::a(), FlowField::b(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), square(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
457                                     {
458   FlowField& f = (FlowField&) f_;
459   fieldstate sxz = f.xzstate();
460   fieldstate sy  = f.ystate();
461   f.makePhysical();
462
463   if (!f.geomCongruent(normf) || normf.vectorDim() != 1)
464     normf.resize(f.Nx(), f.Ny(), f.Nz(), 1, f.Lx(), f.Lz(), f.a(), f.b());
465   else
466     normf.setToZero();
467   normf.setState(Physical, Physical);
468
469   // compute gradf(nx,ny,nz,i) = df(nx,ny,nz,0)/dx_i for scalar-valued f
470   int Nx = f.Nx();
471   int Ny = f.Ny();
472   int Nz = f.Nz();
```

```
473   int Nd = f.Nd();
474
475   for (int ny=0; ny<Ny; ++ny)
476     for (int nx=0; nx<Nx; ++nx)
477       for (int nz=0; nz<Nz; ++nz) {
478         Real nrm2 = 0.0;
479         for (int i=0; i<Nd; ++i)
480           nrm2 += square(f(nx,ny,nz,i));
481         normf(nx,ny,nz,0) = sqrt(nrm2);
482       }
483
484   normf.makeSpectral();
485   f.makeState(sxz, sy);
486   return;
487 }
```

Here is the call graph for this function:



1.9.2.25 FlowField norm2 (const FlowField & *f*)

References norm2().

```
853 {FlowField g; norm2(f,g); return g;}
```

Here is the call graph for this function:

1.9.2.26 void norm2 (const FlowField & *f_*, FlowField & *norm2f*)

References FlowField::a(), FlowField::b(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, square(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
427                                     {
428   FlowField& f = (FlowField&) f_;
429   fieldstate sxz = f.xzstate();
430   fieldstate sy  = f.ystate();
431   f.makeState(Spectral, Spectral);
432
433   if (!f.geomCongruent(norm2f) || norm2f.vectorDim() != 1)
434     norm2f.resize(f.Nx(), f.Ny(), f.Nz(), 1, f.Lx(), f.Lz(), f.a(), f.b());
435   else
436     norm2f.setToZero();
437   norm2f.setState(Physical, Physical);
438
439   f.makePhysical();
440
441   int Nx = f.Nx();
442   int Ny = f.Ny();
443   int Nz = f.Nz();
444   int Nd = f.Nd();
445
446   for (int i=0; i<Nd; ++i)
447     for (int ny=0; ny<Ny; ++ny)
448       for (int nx=0; nx<Nx; ++nx)
449         for (int nz=0; nz<Nz; ++nz)
450           norm2f(nx,ny,nz,0) += square(f(nx,ny,nz,i));
451
452   f.makeState(sxz,sy);
453   norm2f.makeSpectral();
454   return;
```

```
455 }
```

Here is the call graph for this function:

1.9.2.27 ostream& operator<< (ostream & *os*, NonlinearMethod *nonl*)

References Alternating, Alternating_, Convection, Divergence, Linearized, Rotational, and SkewSymmetric.

```
1656                                        {
1657   string s;
1658   switch(nonl) {
1659   case Rotational: s="Rotational"; break;
1660   case Convection: s="Convection"; break;
1661   case Divergence: s="Divergence"; break;
1662   case SkewSymmetric: s="SkewSymmetric"; break;
1663   case Alternating: s="Alternating"; break;
1664   case Alternating_: s="Alternating_"; break;
1665   case Linearized: s="Linearized"; break;
1666   default: s = "Invalid NonlinearMethod: please submit bug report";
1667   }
1668   os << s;
1669   return os;
1670 }
```

1.9.2.28 FlowField outer (const FlowField & *f*, const FlowField & *g*)

References outer().

```
856                                         {
857   FlowField fg; outer(f,g,fg); return fg;
858 }
```

Here is the call graph for this function:

1.9.2.29 void outer (const FlowField & *f_*, const FlowField & *g_*, FlowField & *fg*)

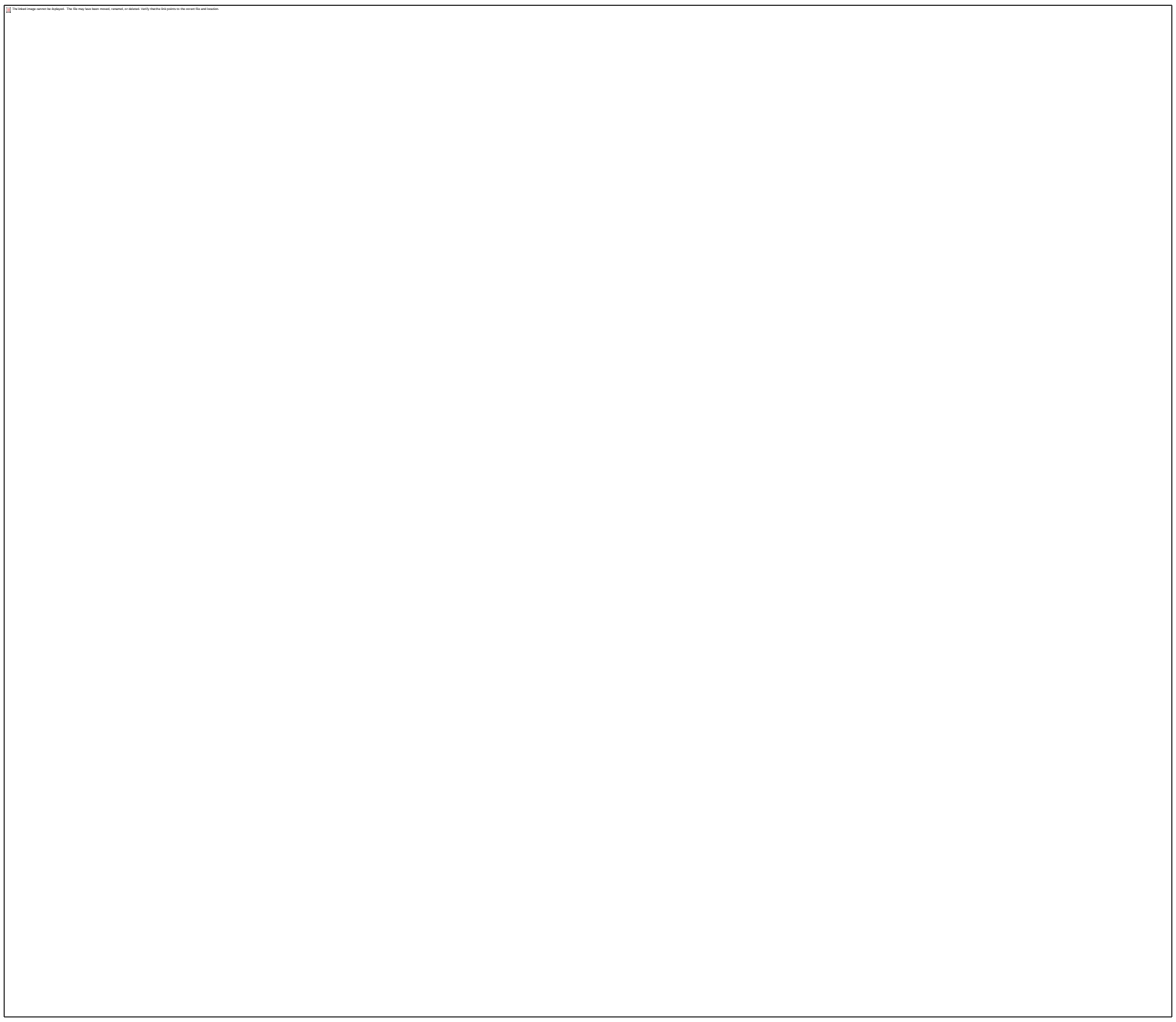
References FlowField::a(), FlowField::b(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), FlowField::xzstate(), and FlowField::ystate().

Referenced by divergenceNL(), and outer().

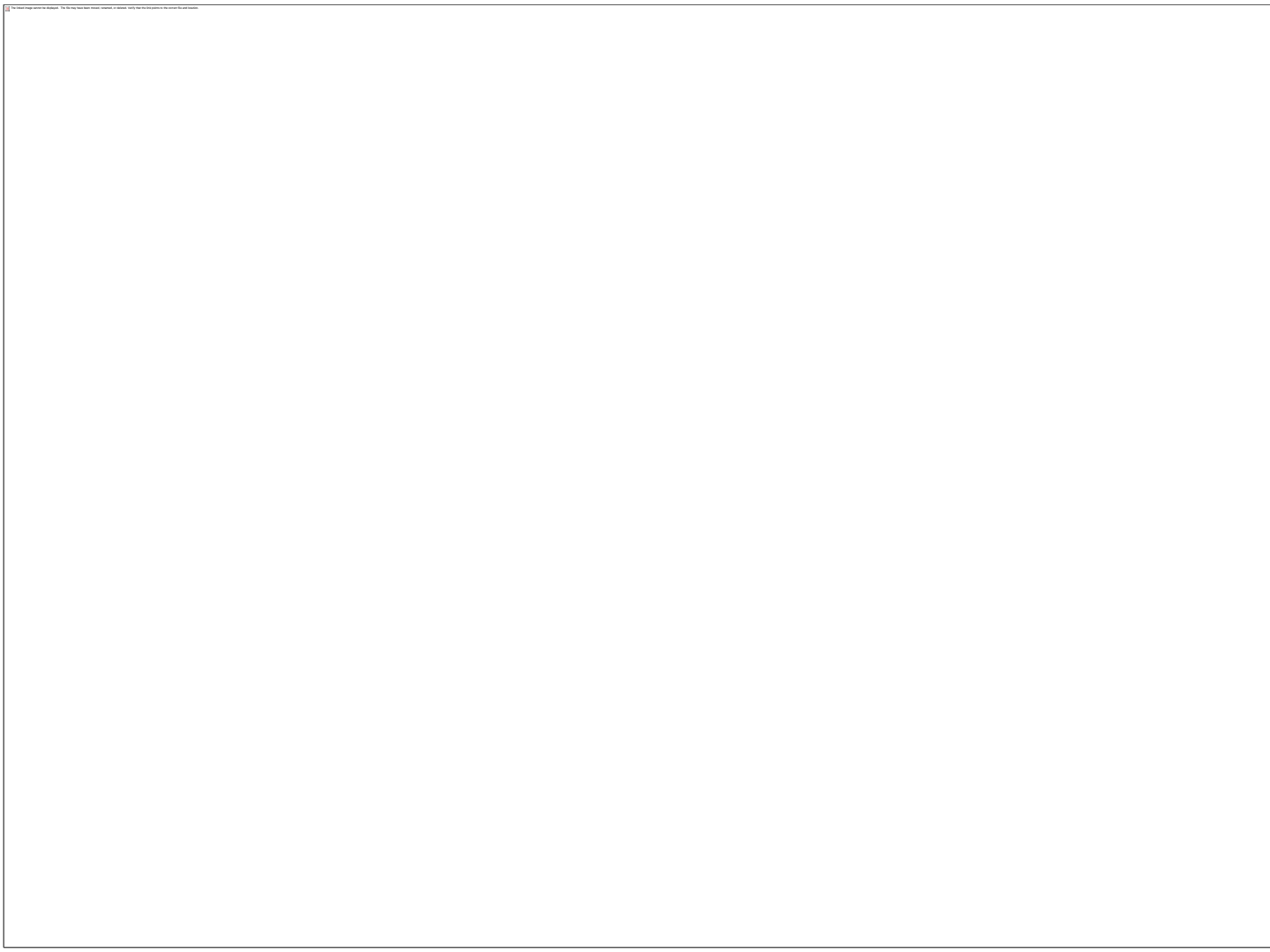
```
552                                                 {
553   FlowField& f = (FlowField&) f_;
554   FlowField& g = (FlowField&) g_;
555   fieldstate fxz = f.xzstate();
556   fieldstate fy  = f.ystate();
557   fieldstate gxz = g.xzstate();
558   fieldstate gy  = g.ystate();
559
560   assert(f.geomCongruent(g));
561   f.makePhysical();
562   g.makePhysical();
563
564   if (!f.geomCongruent(fg) || fg.Nd() != f.Nd()*g.Nd())
565     fg.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd()*g.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
566   else
567     fg.setToZero();
568   fg.setState(Physical, Physical);
569
570   int Nx = f.Nx();
571   int Ny = f.Ny();
572   int Nz = f.Nz();
573   int fd = f.Nd();
574   int gd = g.Nd();
575
576   for (int i=0; i<fd; ++i)
577     for (int j=0; j<gd; ++j) {
578       int ij = i*gd + j;   // generalized form of i3j == i*3 + j
579       for (int ny=0; ny<Ny; ++ny)
580         for (int nx=0; nx<Nx; ++nx)
581           for (int nz=0; nz<Nz; ++nz)
582             fg(nx,ny,nz,ij) = f(nx,ny,nz,i)*g(nx,ny,nz,j);
583     }
584
585   f.makeState(fxz, fy);
586   g.makeState(gxz, gy);
587   fg.makeSpectral();
```

```
588   return;
589 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.9.2.30 void rotationalNL (const FlowField & *u_*, const ChebyCoeff & *U_*, FlowField & *f*, FlowField & *tmp*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::cmplx(), curl(), diff(), FlowField::geomCongruent(), FlowField::kx(), FlowField::kxmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), ChebyCoeff::makePhysical(), ChebyCoeff::makeSpectral(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), ChebyCoeff::N(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

Referenced by navierstokesNL().

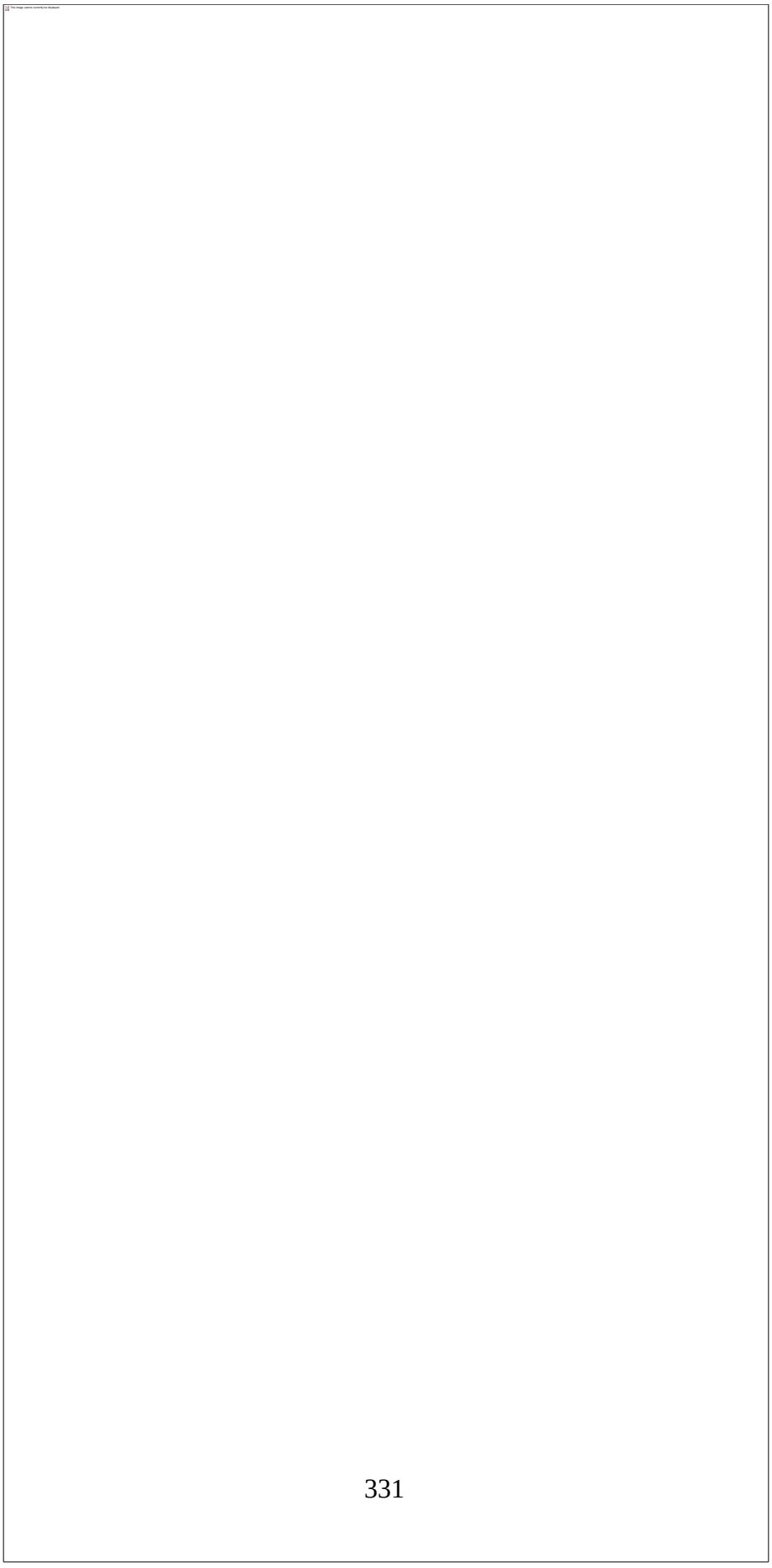
```
925                         {
```

```
926
927  FlowField& u = (FlowField&) u_;
928  ChebyCoeff& U = (ChebyCoeff&) U_;
929  FlowField& vort = tmp;
930
931  assert(u.Nd() == 3);
932  assert(U.numModes() == u.Ny());
933  assert(U.a() == u.a() && U.b() == u.b());
934  fieldstate uxzstate = u.xzstate();
935  fieldstate uystate  = u.ystate();
936  fieldstate Ustate   = U.state();
937
938  if (!u.geomCongruent(f) || f.Nd() != 3)
939    f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
940  else
941    f.setToZero();
942  f.setState(Physical, Physical);
943
944  if (!u.geomCongruent(vort) || vort.Nd() < 3)
945    vort.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
946
947  ChebyTransform trans(U.N());
948  u.makeSpectral();
949  U.makeSpectral(trans);
950  ChebyCoeff Uy = diff(U);
951
952  curl(u, vort);
953
954  u.makeState(Spectral, Physical);;
955  vort.makeState(Spectral, Physical);
956  f.setState(Spectral, Physical);
957  U.makePhysical(trans);
958  Uy.makePhysical(trans);
959
960  Complex uc;
961  Complex vc;
962  Complex wc;
963
964  int Mx = u.Mx();
965  int My = u.My();
966  int Mz = u.Mz();
967  int kxmax = f.kxmax();
968
969  for (int ny=0; ny<My; ++ny) {
970    Real rU  = U[ny];
971    Real rUy = Uy[ny];
```

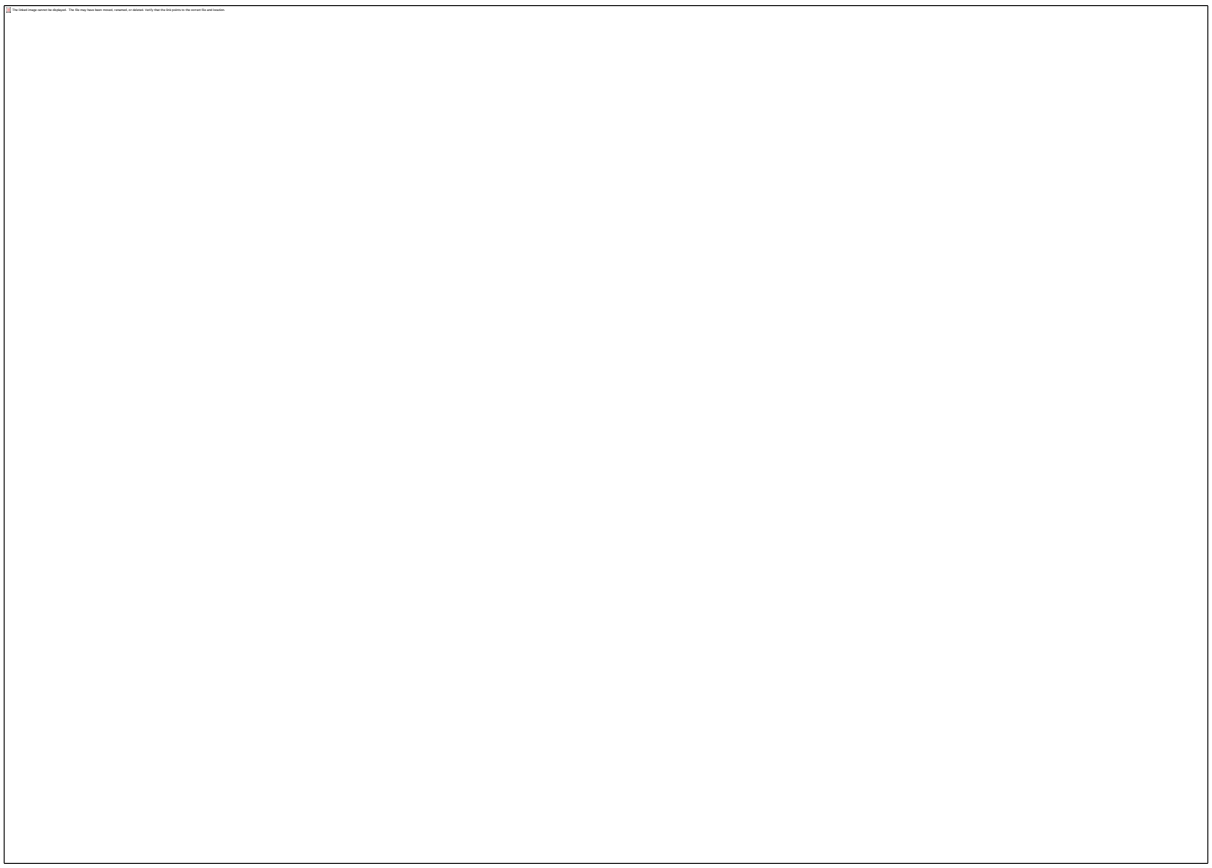
```
    for (int mx=0; mx<Mx; ++mx) {
      int kx = u.kx(mx);
      Complex d_dx(0.0, 2*pi*kx/u.Lx()*zero_last_mode(kx,kxmax,1));
      for (int mz=0; mz<Mz; ++mz) {
        uc = u.cmplx(mx,ny,mz, 0);
        vc = u.cmplx(mx,ny,mz, 1);
        wc = u.cmplx(mx,ny,mz, 2);

        f.cmplx(mx,ny,mz, 0) = rU*d_dx*uc + vc*rUy;
        f.cmplx(mx,ny,mz, 1) = rU*d_dx*vc;
        f.cmplx(mx,ny,mz, 2) = rU*d_dx*wc;
      }
    }
  }

  vort.makePhysical();
  u.makePhysical();
  f.makePhysical();

  int Nx = u.Nx();
  int Ny = u.Ny();
  int Nz = u.Nz();

  Real omega_x;
  Real omega_y;
  Real omega_z;
  Real ru;
  Real rv;
  Real rw;

  // Loop should be efficient if we get 9 cache lines.
  // Add in omega x u so that
  // f = omega x U + Omega x u + omega x u
  for (int ny=0; ny<Ny; ++ny) {
    //Real Uy = Ubasey[ny];
    for (int nx=0; nx<Nx; ++nx) {
      for (int nz=0; nz<Nz; ++nz) {
        ru = u(nx,ny,nz, 0);
        rv = u(nx,ny,nz, 1);
        rw = u(nx,ny,nz, 2);
        omega_x = vort(nx,ny,nz, 0);
        omega_y = vort(nx,ny,nz, 1);
        omega_z = vort(nx,ny,nz, 2);

        f(nx,ny,nz, 0) += omega_y*rw - omega_z*rv;
        f(nx,ny,nz, 1) += omega_z*ru - omega_x*rw;
```

```
1018         f(nx,ny,nz, 2) += omega_x*rv - omega_y*ru;
1019       }
1020     }
1021   }
1022   f.makeSpectral();
1023 
1024   // Possible extra transforms if entry state was Physical
1025   u.makeState(uxzstate, uystate);
1026   U.makeState(Ustate, trans);
1027 
1028   return;
1029 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.9.2.31 void skewsymmetricNL (const FlowField & *u_*, const ChebyCoeff & *U_*, FlowField & *f*, FlowField & *tmp*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), grad(), i3j(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), ChebyCoeff::makeSpectral(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

Referenced by navierstokesNL().

```
1130                                                    {
1131
1132  FlowField& u = (FlowField&) u_;
1133  ChebyCoeff& U = (ChebyCoeff&) U_;
```

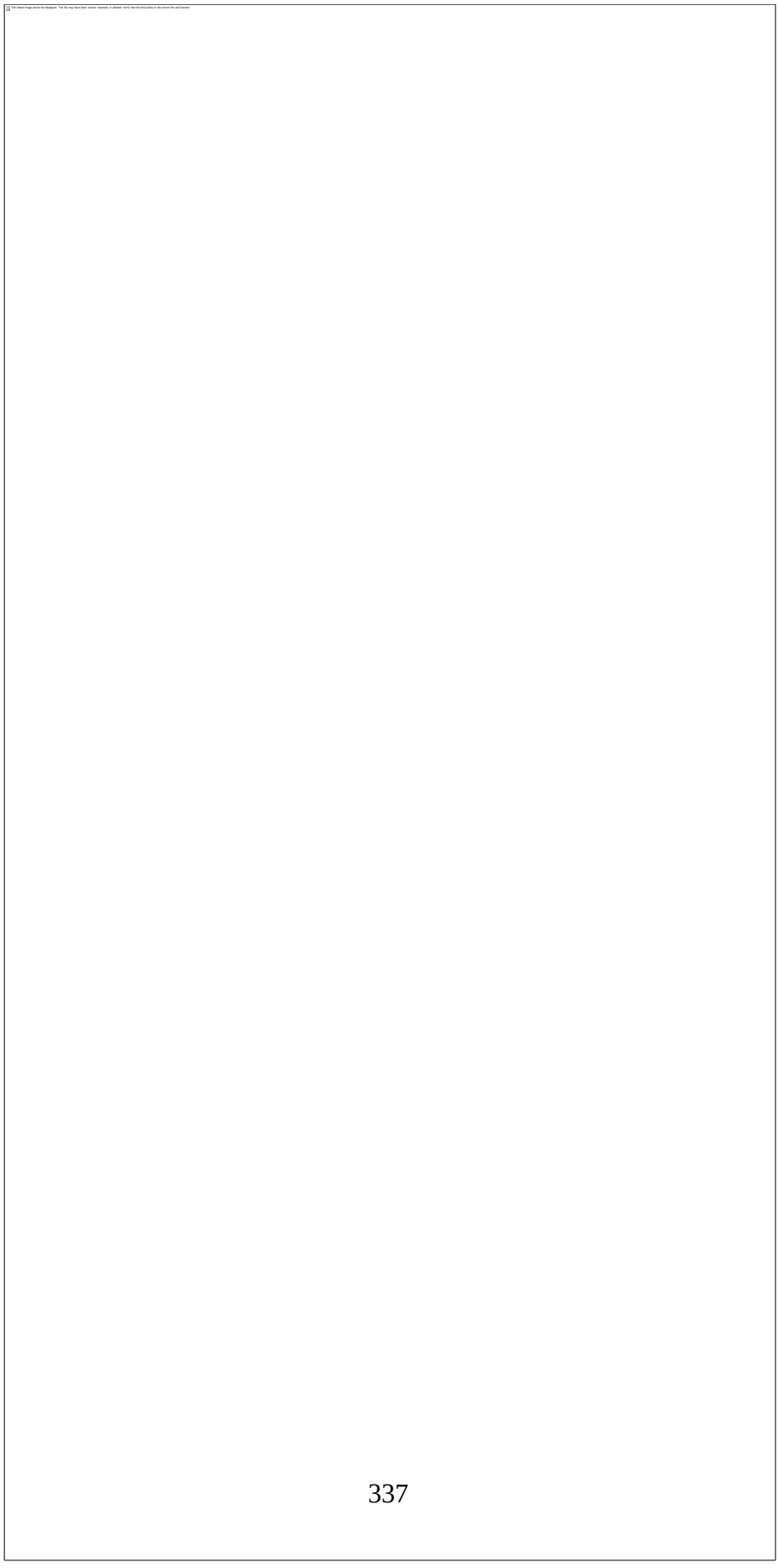
```
1134
1135  assert(u.Nd() == 3);
1136  assert(U.numModes() == u.Ny());
1137  assert(U.a() == u.a() && U.b() == u.b());
1138
1139  fieldstate uxzstate = u.xzstate();
1140  fieldstate uystate  = u.ystate();
1141  fieldstate Ustate   = U.state();
1142
1143  if (!u.geomCongruent(f) || f.Nd() != 3)
1144    f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
1145  else
1146    f.setToZero();
1147  f.setState(Physical, Physical);
1148
1149  if (!u.geomCongruent(tmp) || tmp.Nd() < 9)
1150    tmp.resize(u.Nx(), u.Ny(), u.Nz(), 9, u.Lx(), u.Lz(), u.a(), u.b());
1151  else
1152    tmp.setToZero();
1153
1154  // Possible extra transforms if entry states are Physical
1155  u.makeSpectral();
1156  U.makeSpectral();
1157  u += U;
1158
1159  //
==========================================================================
==
1160  // Compute 1/2 u dotgrad u. (where u is set to former u + U)
1161
1162  FlowField& grad_u = tmp;
1163  grad(u, grad_u);
1164
1165  grad_u.makePhysical();
1166  u.makePhysical();
1167
1168  int Nx = u.Nx();
1169  int Ny = u.Ny();
1170  int Nz = u.Nz();
1171
1172
1173  // Accumulate 1/2 u_j du_i/dx_j in f_i
1174  clock_t t = clock();
1175  for (int i=0; i<3; ++i)
1176    for (int j=0; j<3; ++j) {
1177      int ij = i3j(i,j);
```

```
1178      for (int ny=0; ny<Ny; ++ny)
1179        for (int nx=0; nx<Nx; ++nx)
1180          for (int nz=0; nz<Nz; ++nz)
1181            f(nx,ny,nz,i) += 0.5*u(nx,ny,nz,j)*grad_u(nx,ny,nz,ij);
1182    }
1183
1184   t = clock() -t;
1185   printf("IG: first task cpu  time = %f  \n",((float)t)/CLOCKS_PER_SEC);
1186   //exit(0);
1187
1188
1189   //
==========================================================================
1190   // II. Add grad dot (u u) to f. Spell out loops because div(uu, f)
1191   // would overwrite results already in f (and changing order of div
1192   // and convec calculations would require an extra transform)
1193
1194   FlowField& uu = tmp;
1195
1196   //outer(u,u,uu);
1197    t = clock();
1198
1199   for (int ny=0; ny<Ny; ++ny) {
1200     for (int nx=0; nx<Nx; ++nx)
1201       for (int nz=0; nz<Nz; ++nz) {
1202         Real u0 = u(nx,ny,nz,0);
1203         Real u1 = u(nx,ny,nz,1);
1204         Real u2 = u(nx,ny,nz,2);
1205         uu(nx,ny,nz,0) = u0*u0;
1206         uu(nx,ny,nz,1) = tmp(nx,ny,nz,3) = u0*u1;
1207         uu(nx,ny,nz,2) = tmp(nx,ny,nz,6) = u0*u2;
1208         uu(nx,ny,nz,4) = u1*u1;
1209         uu(nx,ny,nz,5) = tmp(nx,ny,nz,7) = u1*u2;
1210         uu(nx,ny,nz,8) = u2*u2;
1211       }
1212   }
1213
1214    t = clock() -t;
1215   printf("IG: second task cpu  time = %f  \n",((float)t)/CLOCKS_PER_SEC);
1216
1217   uu.makeSpectral();
1218   f.makeSpectral();
1219
1220   int Mx = u.Mx();
1221   int My = u.My();
1222   int Mz = u.Mz();
```

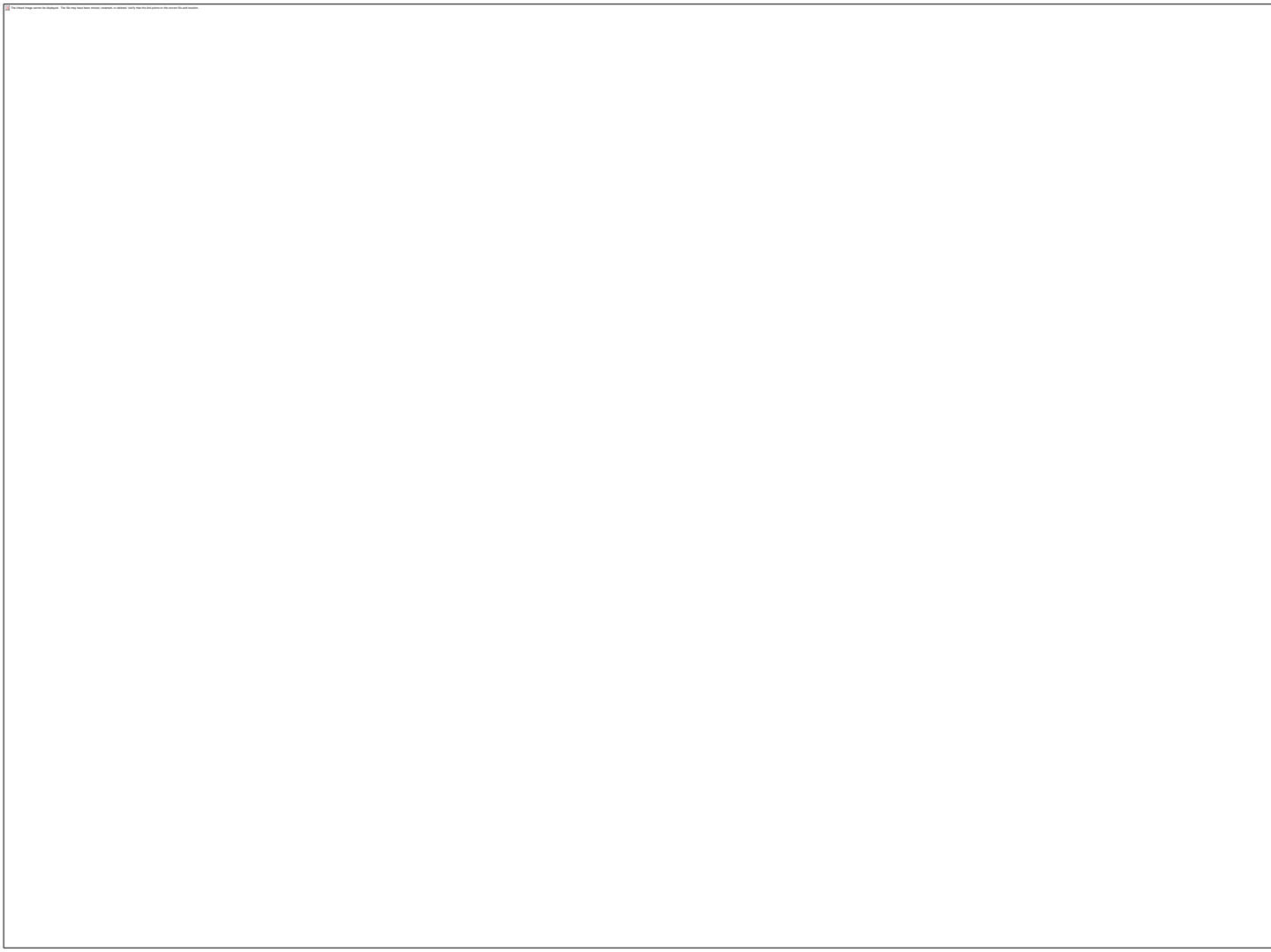
```
1223  int kxmax = u.kxmax();
1224  int kzmax = u.kzmax();
1225  Real Lx = u.Lx();
1226  Real Lz = u.Lz();
1227
1228  ComplexChebyCoeff tmpProfile(My, u.a(), u.b(), Spectral);
1229  ComplexChebyCoeff tmpProfile_y(My, u.a(), u.b(), Spectral);
1230
1231  // Now set f_i += d/dx_j (u_i u_j)
1232   t = clock();
1233
1234  for (int i=0; i<3; ++i) {
1235    int i0 = i3j(i,0);
1236    int i1 = i3j(i,1);
1237    int i2 = i3j(i,2);
1238
1239    // Add in du_i/dx and du_i/dz, that is, d/dx_j (u_i u_j) for j=0,2
1240    for (int my=0; my<My; ++my)
1241      for (int mx=0; mx<Mx; ++mx) {
1242        int kx = u.kx(mx);
1243        Complex d_dx(0.0, 2*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
1244        for (int mz=0; mz<Mz; ++mz) {
1245          int kz = u.kz(mz);
1246          Complex d_dz(0.0, 2*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
1247          f.cmplx(mx,my,mz,i)
1248            += 0.5*(d_dx*uu.cmplx(mx,my,mz,i0)+d_dz*uu.cmplx(mx,my,mz,i2));
1249        }
1250      }
1251    // Add in du_i/dy, that is d/dx_j (u_i u_j) for j=1
1252    for (int mx=0; mx<Mx; ++mx)
1253      for (int mz=0; mz<Mz; ++mz) {
1254        for (int my=0; my<My; ++my)
1255          tmpProfile.set(my, uu.cmplx(mx,my,mz,i1));
1256        diff(tmpProfile, tmpProfile_y);
1257        for (int my=0; my<My; ++my)
1258          f.cmplx(mx,my,mz,i) += 0.5*tmpProfile_y[my]; // j=1
1259      }
1260  }
1261
1262   t = clock() -t;
1263  printf("IG: thirt  task cpu  time = %f  \n",((float)t)/CLOCKS_PER_SEC);
1264
1265  u.makeSpectral();
1266  U.makeSpectral();
1267  u -= U;
1268
```

```
1269   // Possible extra transforms if entry states are Physical
1270   u.makeState(uxzstate, uystate);
1271   U.makeState(Ustate);
1272
1273   //exit(0);
1274   return;
1275 }
```

Here is the call graph for this function:


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.

Here is the caller graph for this function:



1.9.2.32 void skewsymmetricNL_GPU (const FlowField & *u_*, const ChebyCoeff & *U_*, FlowField & *f*, FlowField & *tmp*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), grad(), i3j(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), ChebyCoeff::makeSpectral(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

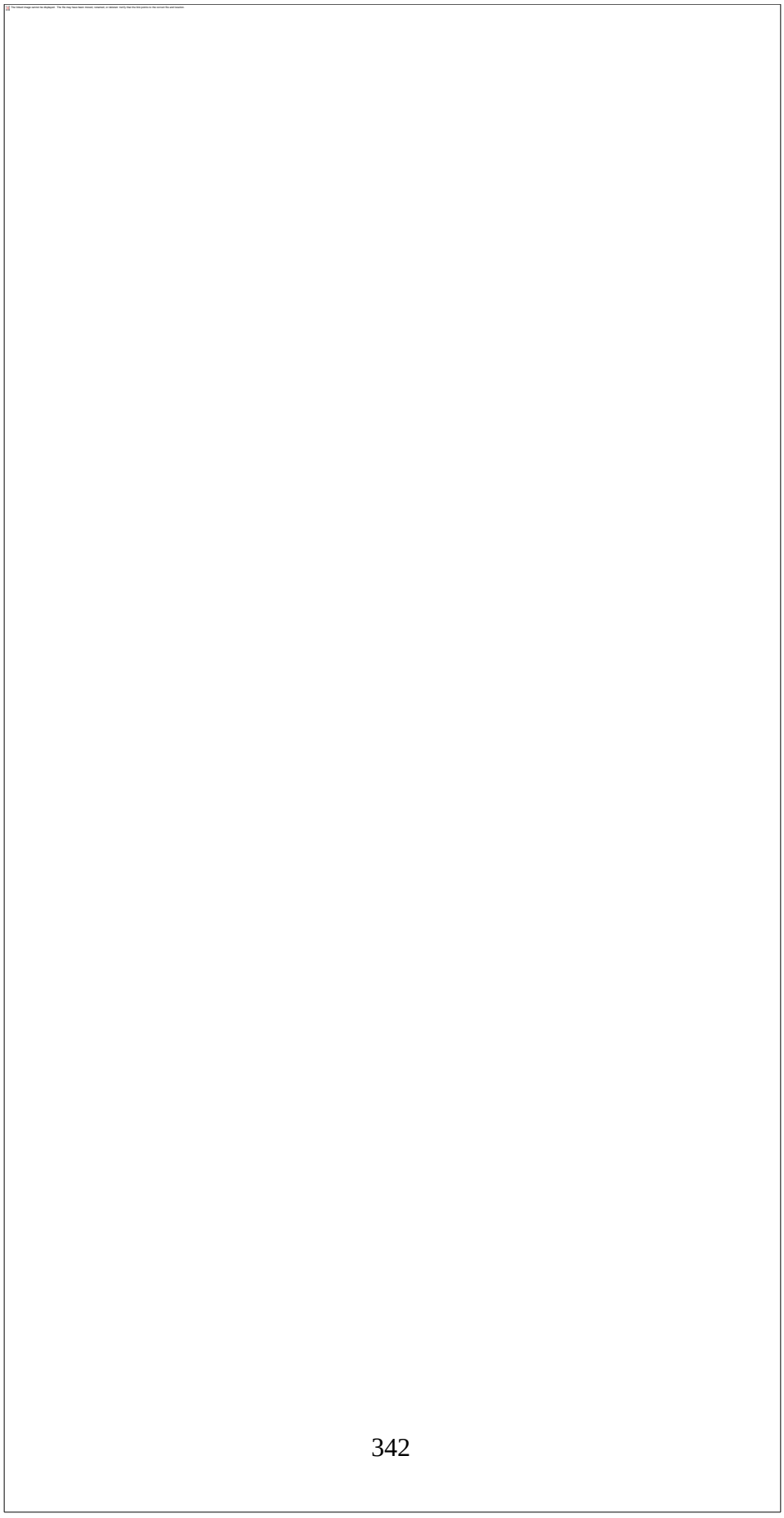
Referenced by navierstokesNL().

```
1277                                                                {
1278
1279  FlowField& u = (FlowField&) u_;
1280  ChebyCoeff& U = (ChebyCoeff&) U_;
1281
1282  assert(u.Nd() == 3);
1283  assert(U.numModes() == u.Ny());
1284  assert(U.a() == u.a() && U.b() == u.b());
1285
1286  fieldstate uxzstate = u.xzstate();
1287  fieldstate uystate  = u.ystate();
1288  fieldstate Ustate   = U.state();
1289
1290  if (!u.geomCongruent(f) || f.Nd() != 3)
1291    f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
1292  else
1293    f.setToZero();
1294  f.setState(Physical, Physical);
1295
1296  if (!u.geomCongruent(tmp) || tmp.Nd() < 9)
1297    tmp.resize(u.Nx(), u.Ny(), u.Nz(), 9, u.Lx(), u.Lz(), u.a(), u.b());
1298  else
1299    tmp.setToZero();
1300
1301  // Possible extra transforms if entry states are Physical
1302  u.makeSpectral();
1303  U.makeSpectral();
1304  u += U;
1305
1306  //
==========================================================================
==
1307  // Compute 1/2 u dotgrad u. (where u is set to former u + U)
1308
1309  FlowField& grad_u = tmp;
1310  grad(u, grad_u);
1311
1312  grad_u.makePhysical();
1313  u.makePhysical();
1314
1315  int Nx = u.Nx();
1316  int Ny = u.Ny();
1317  int Nz = u.Nz();
1318
1319  // Accumulate 1/2 u_j du_i/dx_j in f_i
```

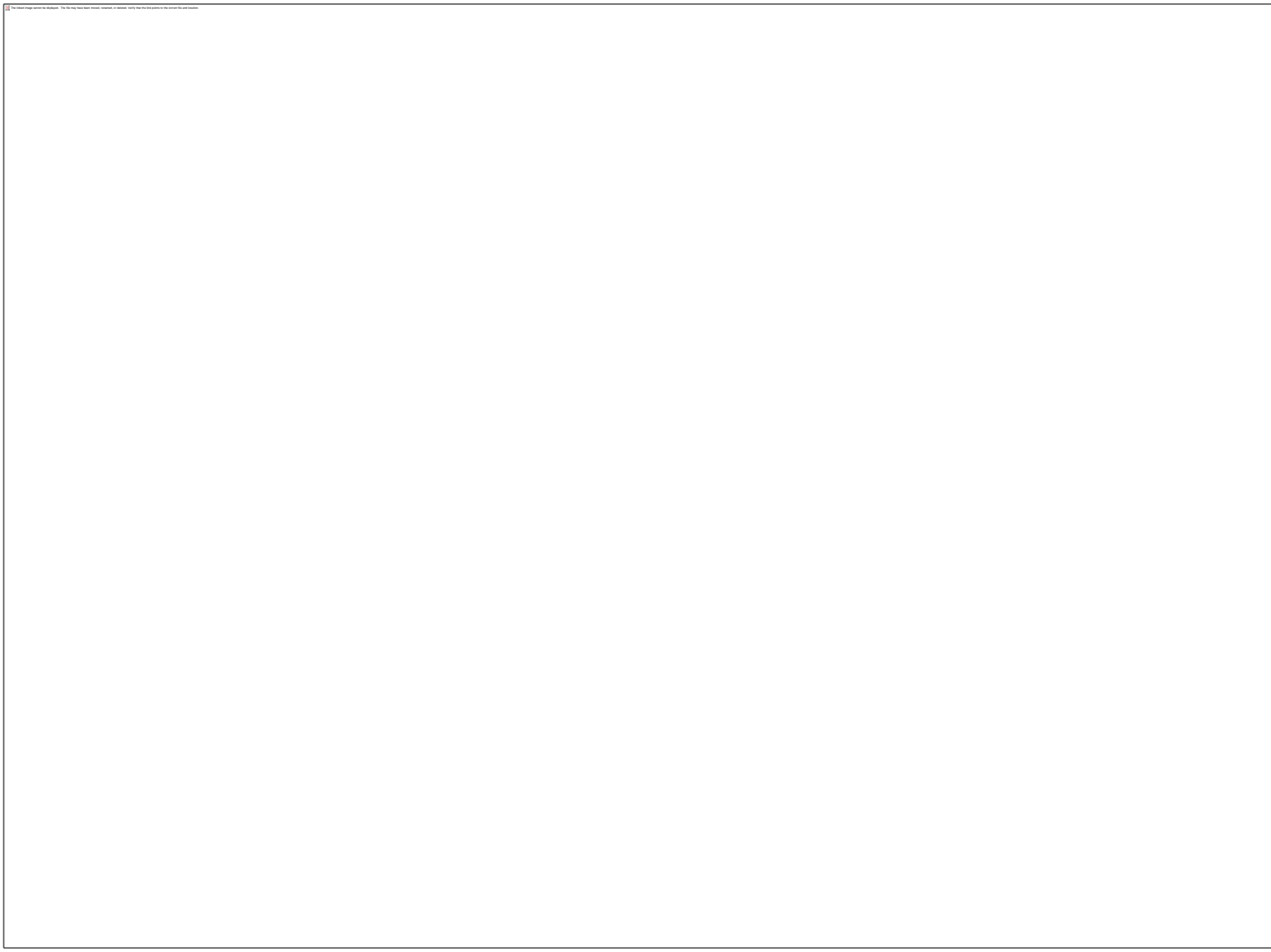
```
1320  for (int i=0; i<3; ++i)
1321    for (int j=0; j<3; ++j) {
1322      int ij = i3j(i,j);
1323      for (int ny=0; ny<Ny; ++ny)
1324        for (int nx=0; nx<Nx; ++nx)
1325          for (int nz=0; nz<Nz; ++nz)
1326            f(nx,ny,nz,i) += 0.5*u(nx,ny,nz,j)*grad_u(nx,ny,nz,ij);
1327    }
1328
1329  //
==============================================================
1330  // II. Add grad dot (u u) to f. Spell out loops because div(uu, f)
1331  // would overwrite results already in f (and changing order of div
1332  // and convec calculations would require an extra transform)
1333
1334  FlowField& uu = tmp;
1335
1336  //outer(u,u,uu);
1337
1338  for (int ny=0; ny<Ny; ++ny) {
1339    for (int nx=0; nx<Nx; ++nx)
1340      for (int nz=0; nz<Nz; ++nz) {
1341        Real u0 = u(nx,ny,nz,0);
1342        Real u1 = u(nx,ny,nz,1);
1343        Real u2 = u(nx,ny,nz,2);
1344        uu(nx,ny,nz,0) = u0*u0;
1345        uu(nx,ny,nz,1) = tmp(nx,ny,nz,3) = u0*u1;
1346        uu(nx,ny,nz,2) = tmp(nx,ny,nz,6) = u0*u2;
1347        uu(nx,ny,nz,4) = u1*u1;
1348        uu(nx,ny,nz,5) = tmp(nx,ny,nz,7) = u1*u2;
1349        uu(nx,ny,nz,8) = u2*u2;
1350      }
1351  }
1352
1353  uu.makeSpectral();
1354  f.makeSpectral();
1355
1356  int Mx = u.Mx();
1357  int My = u.My();
1358  int Mz = u.Mz();
1359  int kxmax = u.kxmax();
1360  int kzmax = u.kzmax();
1361  Real Lx = u.Lx();
1362  Real Lz = u.Lz();
1363
1364  ComplexChebyCoeff tmpProfile(My, u.a(), u.b(), Spectral);
```

```
1365  ComplexChebyCoeff tmpProfile_y(My, u.a(), u.b(), Spectral);
1366
1367  // Now set f_i += d/dx_j (u_i u_j)
1368  for (int i=0; i<3; ++i) {
1369    int i0 = i3j(i,0);
1370    int i1 = i3j(i,1);
1371    int i2 = i3j(i,2);
1372
1373    // Add in du_i/dx and du_i/dz, that is, d/dx_j (u_i u_j) for j=0,2
1374    for (int my=0; my<My; ++my)
1375      for (int mx=0; mx<Mx; ++mx) {
1376        int kx = u.kx(mx);
1377        Complex d_dx(0.0, 2*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
1378        for (int mz=0; mz<Mz; ++mz) {
1379          int kz = u.kz(mz);
1380          Complex d_dz(0.0, 2*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
1381          f.cmplx(mx,my,mz,i)
1382            += 0.5*(d_dx*uu.cmplx(mx,my,mz,i0)+d_dz*uu.cmplx(mx,my,mz,i2));
1383        }
1384      }
1385    // Add in du_i/dy, that is d/dx_j (u_i u_j) for j=1
1386    for (int mx=0; mx<Mx; ++mx)
1387      for (int mz=0; mz<Mz; ++mz) {
1388        for (int my=0; my<My; ++my)
1389          tmpProfile.set(my, uu.cmplx(mx,my,mz,i1));
1390        diff(tmpProfile, tmpProfile_y);
1391        for (int my=0; my<My; ++my)
1392          f.cmplx(mx,my,mz,i) += 0.5*tmpProfile_y[my]; // j=1
1393      }
1394  }
1395
1396  u.makeSpectral();
1397  U.makeSpectral();
1398  u -= U;
1399
1400  // Possible extra transforms if entry states are Physical
1401  u.makeState(uxzstate, uystate);
1402  U.makeState(Ustate);
1403
1404  return;
1405 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.9.2.33 void skewsymmetricNL_THREAD (const FlowField & *u_*, const ChebyCoeff & *U_*, FlowField & *f*, FlowField & *tmp*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::cmplx(), diff(), ThreadPool::executeTask(), FlowField::geomCongruent(), ThreadPool::GetInstance(), grad(), i3j(), FlowField::kxmax(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), ChebyCoeff::makeSpectral(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), and FlowField::ystate().

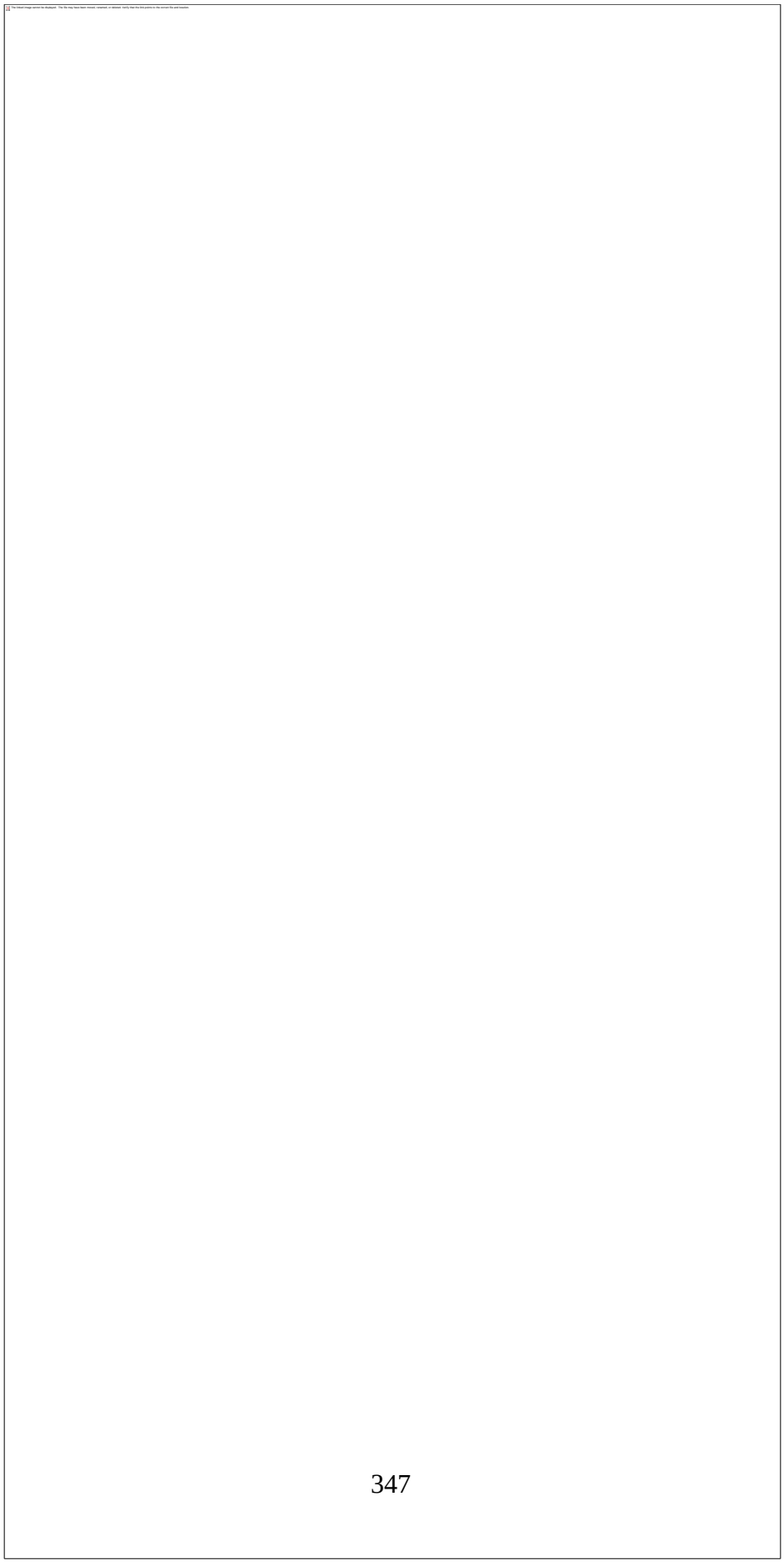
Referenced by navierstokesNL().

```
1408 {
1409   ThreadPool *pool = ThreadPool::GetInstance();
1410
1411   FlowField& u = (FlowField&) u_;
1412   ChebyCoeff& U = (ChebyCoeff&) U_;
1413
1414   assert(u.Nd() == 3);
1415   assert(U.numModes() == u.Ny());
1416   assert(U.a() == u.a() && U.b() == u.b());
1417
1418   fieldstate uxzstate = u.xzstate();
1419   fieldstate uystate  = u.ystate();
1420   fieldstate Ustate   = U.state();
1421
1422   if (!u.geomCongruent(f) || f.Nd() != 3)
1423     f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
1424   else
1425     f.setToZero();
1426   f.setState(Physical, Physical);
1427
1428   if (!u.geomCongruent(tmp) || tmp.Nd() < 9)
1429     tmp.resize(u.Nx(), u.Ny(), u.Nz(), 9, u.Lx(), u.Lz(), u.a(), u.b());
1430   else
1431     tmp.setToZero();
1432
1433   // Possible extra transforms if entry states are Physical
1434   u.makeSpectral();
1435   U.makeSpectral();
1436   u += U;
1437
1438   //
==========================================================================
==
1439   // Compute 1/2 u dotgrad u. (where u is set to former u + U)
1440
1441   FlowField& grad_u = tmp;
1442   grad(u, grad_u);
1443
1444   grad_u.makePhysical();
1445   u.makePhysical();
1446
1447
1448
1449
1450
```

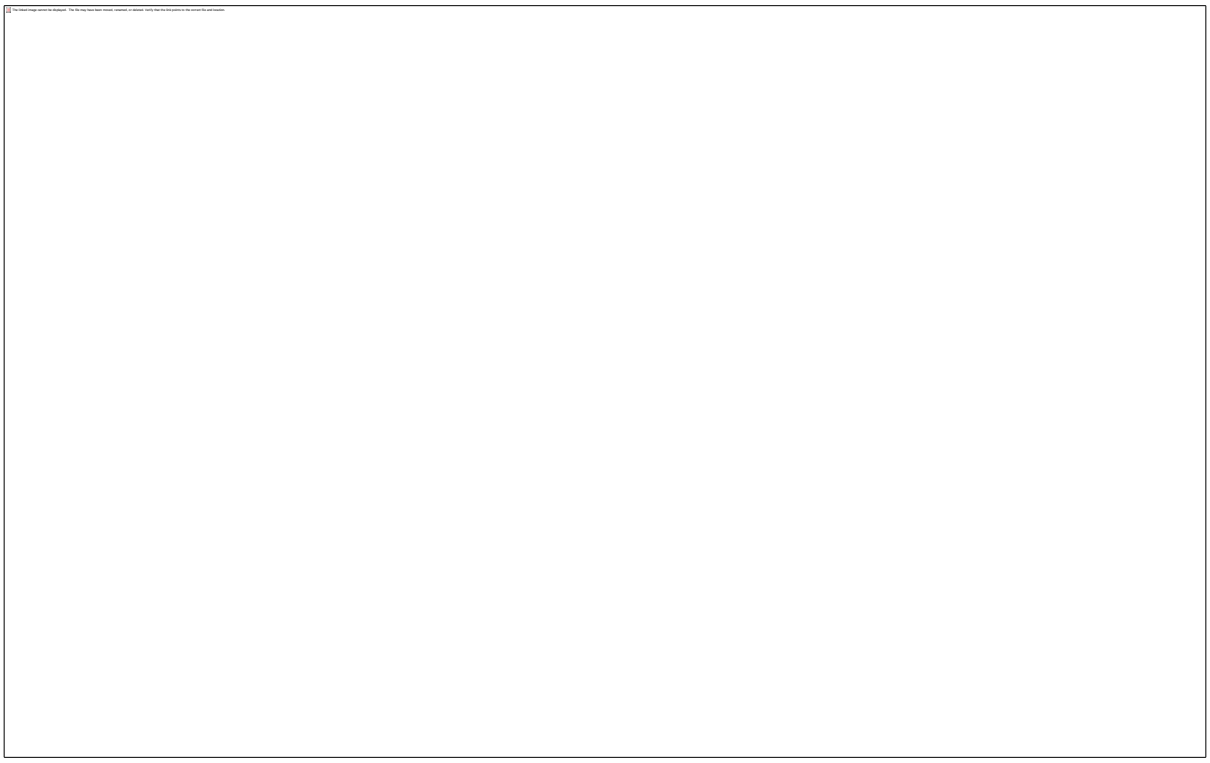
```
1451
1452        //  clock_t tm = clock();
1453        //struct timespec start, finish;
1454        //double elapsed;
1455        //clock_gettime(CLOCK_MONOTONIC, &start);
1456
1457       pool->executeTask( new skewsymmetricNL_task(u, f, grad_u));
1458
1459     //   clock_gettime(CLOCK_MONOTONIC, &finish);
1460     //         elapsed = (finish.tv_sec - start.tv_sec);
1461     //   elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
1462
1463      //  cout << "total time="<<elapsed<<endl;
1464
1465       //   tm = clock() -tm;
1466       //   printf("IG: total time = %f  \n",((float)tm)/CLOCKS_PER_SEC);
1467
1468
1469  //
==========================================================================
1470  // II. Add grad dot (u u) to f. Spell out loops because div(uu, f)
1471  // would overwrite results already in f (and changing order of div
1472  // and convec calculations would require an extra transform)
1473
1474  FlowField& uu = tmp;
1475
1476  //outer(u,u,uu);
1477
1478
1479       pool->executeTask( new skewsymmetricNL_task2(u,uu, tmp));
1480      //uu.print();
1481      //exit(0);
1482
1483  uu.makeSpectral();
1484  f.makeSpectral();
1485
1486  int Mx = u.Mx();
1487  int My = u.My();
1488  int Mz = u.Mz();
1489  int kxmax = u.kxmax();
1490  int kzmax = u.kzmax();
1491  Real Lx = u.Lx();
1492  Real Lz = u.Lz();
1493
1494  ComplexChebyCoeff tmpProfile(My, u.a(), u.b(), Spectral);
1495  ComplexChebyCoeff tmpProfile_y(My, u.a(), u.b(), Spectral);
```

```
1496
1497   pool->executeTask(new skewsymmetricNL_task3(u,uu,f,kxmax,kzmax,Lx,Lz,Mx,My,Mz));
1498  // pool->executeTask(new skewsymmetricNL_task4(u,uu,f,tmpProfile,tmpProfile_y,kxmax,kzmax,Lx,Lz,Mx,My,Mz));
1499
1500   // Now set f_i += d/dx_j (u_i u_j)
1501
1502   for (int i=0; i<3; ++i) {
1503     //int i0 = i3j(i,0);
1504     int i1 = i3j(i,1);
1505     //int i2 = i3j(i,2);
1506     // Add in du_i/dy, that is d/dx_j (u_i u_j) for j=1
1507     for (int mx=0; mx<Mx; ++mx)
1508       for (int mz=0; mz<Mz; ++mz) {
1509         for (int my=0; my<My; ++my)
1510           tmpProfile.set(my, uu.cmplx(mx,my,mz,i1));
1511         diff(tmpProfile, tmpProfile_y);
1512         for (int my=0; my<My; ++my)
1513           f.cmplx(mx,my,mz,i) += 0.5*tmpProfile_y[my]; // j=1
1514       }
1515   }
1516
1517
1518   u.makeSpectral();
1519   U.makeSpectral();
1520   u -= U;
1521
1522   // Possible extra transforms if entry states are Physical
1523   u.makeState(uxzstate, uystate);
1524   U.makeState(Ustate);
1525
1526 }
```

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.

Here is the caller graph for this function:



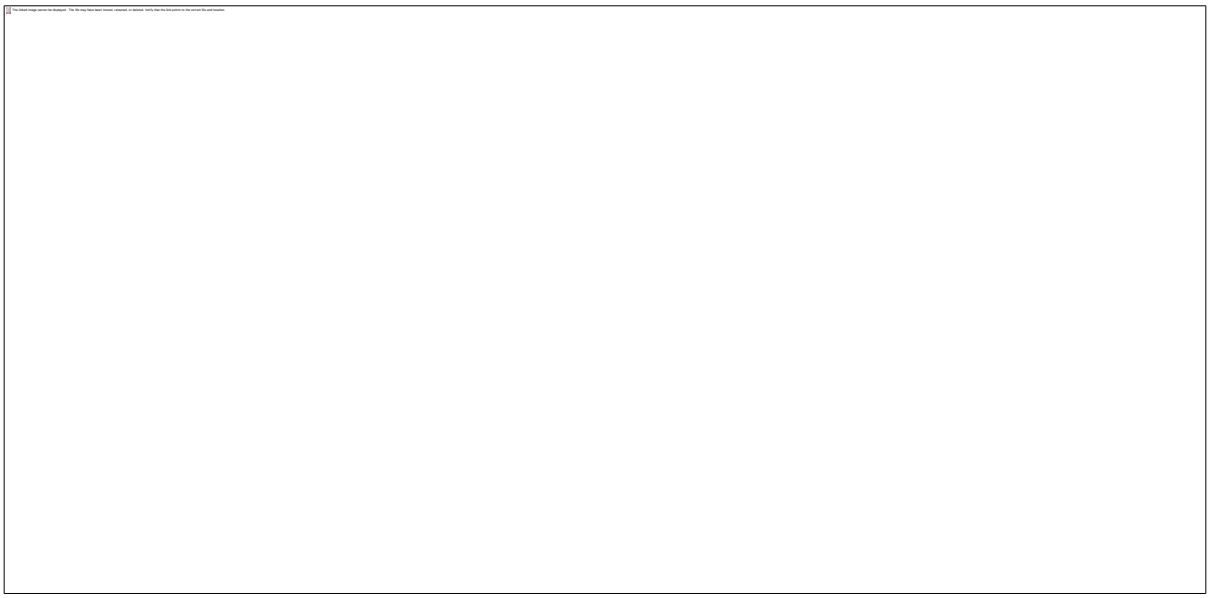
1.9.2.34 void up2q (const FlowField & *u_*, const FlowField & *p*, FlowField & *q*, NonlinearMethod *nonl_method*)

References FlowField::makePhysical(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Rotational, square(), FlowField::xzstate(), and FlowField::ystate().

```
1622                                {
1623
1624   assert(u_.Nd() == 3);
1625   assert(q.Nd() == 1);
1626   assert(p.Nd() == 1);
1627
1628   q = p;
1629   if (nonl_method != Rotational)
1630     return;
1631
1632   FlowField& u = (FlowField&) u_;
1633   fieldstate uxzstate = u.xzstate();
1634   fieldstate uystate = u.ystate();
1635
```

```
1636   u.makePhysical();
1637   q.makePhysical();
1638
1639   int Nx=u_.Nx();
1640   int Ny=u_.Ny();
1641   int Nz=u_.Nz();
1642
1643   // Add 1/2 u dot u to q
1644   for (int ny=0; ny<Ny; ++ny) {
1645     for (int nx=0; nx<Nx; ++nx)
1646       for (int nz=0; nz<Nz; ++nz)
1647         q(nx,ny,nz,0) += 0.5*(square(u(nx,ny,nz,0)) +
1648                               square(u(nx,ny,nz,1)) +
1649                               square(u(nx,ny,nz,2)));
1650   }
1651   u.makeState(uxzstate, uystate);
1652   q.makeState(p.xzstate(), p.ystate());
1653   return;
1654 }
```

Here is the call graph for this function:

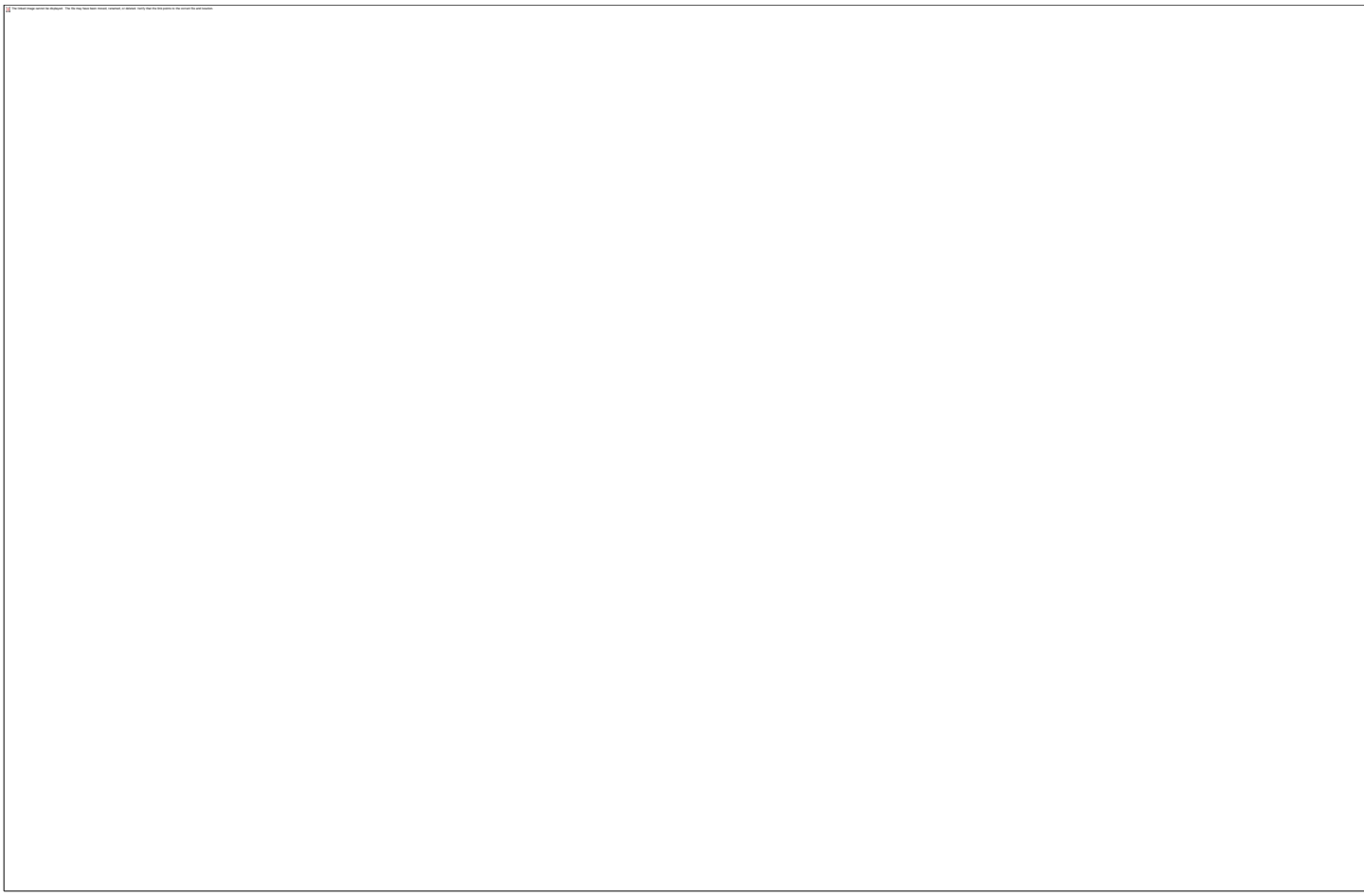


1.9.2.35 void uq2p (const FlowField & *u_*, const FlowField & *q*, FlowField & *p*, NonlinearMethod *nonl_method*)

References FlowField::makePhysical(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Rotational, square(), FlowField::xzstate(), and FlowField::ystate().

```
1588                                {
1589
1590   assert(u_.Nd() == 3);
1591   assert(q.Nd() == 1);
1592   assert(p.Nd() == 1);
1593
1594   p = q;
1595   if (nonl_method != Rotational)
1596     return;
1597
1598   FlowField& u = (FlowField&) u_;
1599   fieldstate uxzstate = u.xzstate();
1600   fieldstate uystate = u.ystate();
1601
1602   u.makePhysical();
1603   p.makePhysical();
1604
1605   int Nx=u_.Nx();
1606   int Ny=u_.Ny();
1607   int Nz=u_.Nz();
1608
1609   // Remove 1/2 u dot u or 1/2 (u+U) dot (u+U) from p
1610   for (int ny=0; ny<Ny; ++ny)
1611     for (int nx=0; nx<Nx; ++nx)
1612       for (int nz=0; nz<Nz; ++nz)
1613         p(nx,ny,nz,0) -= 0.5*(square(u(nx,ny,nz,0)) +
1614                               square(u(nx,ny,nz,1)) +
1615                               square(u(nx,ny,nz,2)));
1616
1617   u.makeState(uxzstate, uystate);
1618   p.makeState(q.xzstate(), q.ystate());
1619 }
```

Here is the call graph for this function:

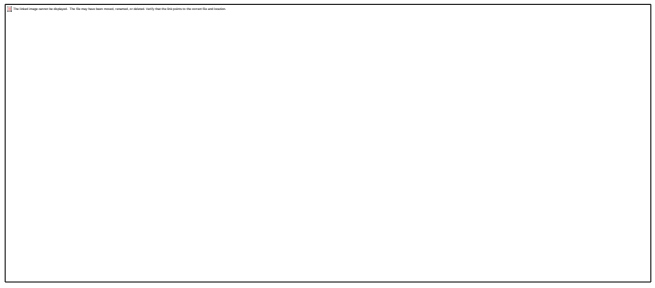


1.9.2.36 FlowField xdiff (const FlowField & *f*, int *n*)

References xdiff().

```
825                                 {
826   FlowField g;
827   xdiff(f,g,n);
828   return g;
829 }
```

Here is the call graph for this function:



1.9.2.37 void xdiff (const FlowField & *f_*, FlowField & *dfdx*, int *n*)

References FlowField::a(), FlowField::b(), cferror(), FlowField::cmplx(), FlowField::congruent(), FlowField::kx(), FlowField::kxmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeSpectral_xz(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, pow(), FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

```
39                                      {
40   if (n==0)
41     return;
42
43   FlowField& f = (FlowField&) f_;
44   fieldstate sxz = f.xzstate();
45   fieldstate sy  = f.ystate();
46   f.makeSpectral_xz();
47
48   Real Lx = f.Lx();
49
50   if (!f.congruent(dfdx))
51     dfdx.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
52   else
53     dfdx.setToZero();
54   dfdx.setState(Spectral, sy);
55
56   Complex rot(0.0, 0.0);
57   switch(n % 4) {
58   case 0: rot = Complex( 1.0, 0.0); break;
59   case 1: rot = Complex( 0.0, 1.0); break;
60   case 2: rot = Complex(-1.0, 0.0); break;
61   case 3: rot = Complex( 0.0,-1.0); break;
62   default: cferror("xdiff(f, dfdx, n) : impossible: n % 4 > 4 !!"); break;
63   }
64
65   int Nd = f.Nd();
66   int Mx = f.Mx();
67   int My = f.My();
68   int Mz = f.Mz();
69   int kxmax = f.kxmax();
70
71   for (int i=0; i<Nd; ++i)
72     for (int my=0; my<My; ++my)
73       for (int mx=0; mx<Mx; ++mx) {
```

```
74        int kx = f.kx(mx);
75        Complex cx = rot*(pow(2*pi*kx/Lx, n)*zero_last_mode(kx,kxmax,n));
76        for (int mz=0; mz<Mz; ++mz)
77          dfdx.cmplx(mx,my,mz,i) = cx*f.cmplx(mx,my,mz,i);
78      }
79
80   f.makeState(sxz,sy);
81   dfdx.makeSpectral();
82   return;
83 }
```

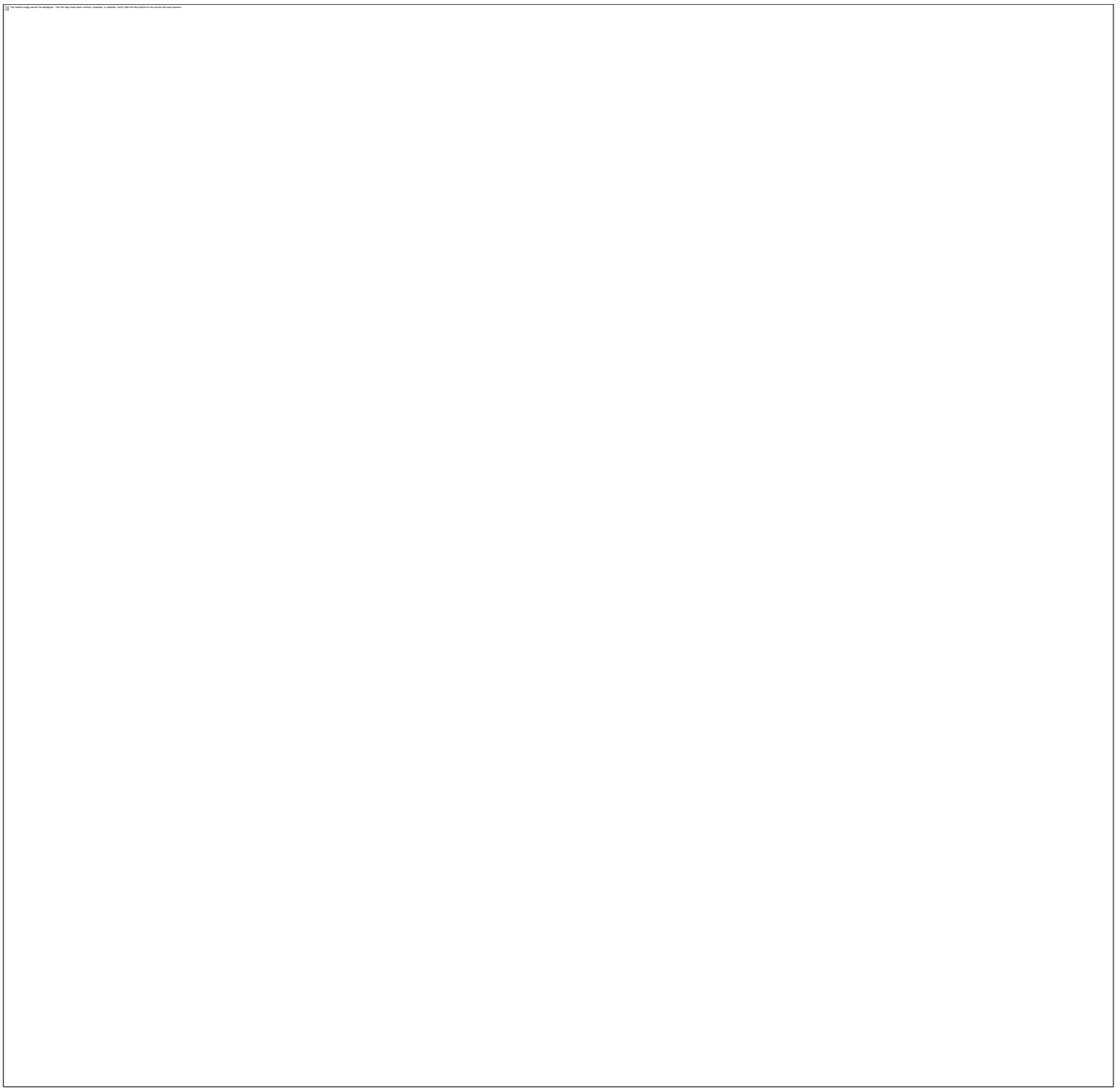
Here is the call graph for this function:

1.9.2.38 FlowField ydiff (const FlowField & *f*, int *n*)

References ydiff().

```
831                                 {
832   FlowField g;
833   ydiff(f,g,n);
834   return g;
835 }
```

Here is the call graph for this function:



1.9.2.39 void ydiff (const FlowField & *f_*, FlowField & *dfdy*, int *n*)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), diff(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeSpectral_y(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(),

FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, swap(), FlowField::xzstate(), and FlowField::ystate().

```
85                                          {
86   if (n==0)
87     return;
88
89   FlowField& f = (FlowField&) f_;
90   fieldstate sxz = f.xzstate();
91   fieldstate sy  = f.ystate();
92   f.makeSpectral_y();
93
94   if (!f.congruent(dfdy))
95     dfdy.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
96   else
97     dfdy.setToZero();
98   dfdy.setState(sxz, Spectral);
99
100    int Nd = f.Nd();
101    if (sxz == Spectral) {
102      int Mx = f.Mx();
103      int My = f.My();
104      int Mz = f.Mz();
105
106      ComplexChebyCoeff  g(f.Ny(), f.a(), f.b(), Spectral);
107      ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
108
109      for (int i=0; i<Nd; ++i)
110        for (int mx=0; mx<Mx; ++mx)
111          for (int mz=0; mz<Mz; ++mz) {
112            for (int my=0; my<My; ++my)
113              g.set(my, f.cmplx(mx,my,mz,i));
114
115            // differentiate n times and leave derivative in *g
116            for (int k=0; k<n; ++k) {
117              diff(g,gy);
118              swap(g,gy);
119            }
120            for (int my=0; my<My; ++my)
121              dfdy.cmplx(mx,my,mz,i) = g[my];
122          }
123    }
124    else {
125      int Nx = f.Nx();
126      int Ny = f.Ny();
127      int Nz = f.Nz();
128      ChebyCoeff g(Ny,  f.a(), f.b(), Spectral);
```

```
129    ChebyCoeff gy(Ny, f.a(), f.b(), Spectral);
130
131    for (int i=0; i<Nd; ++i)
132      for (int nx=0; nx<Nx; ++nx)
133        for (int nz=0; nz<Nz; ++nz) {
134          for (int ny=0; ny<Ny; ++ny)
135            g[ny] = f(nx,ny,nz,i);
136
137          // differentiate n times and leave derivative in g
138          for (int k=0; k<n; ++k) {
139            diff(g,gy);
140            swap(g,gy);
141          }
142          for (int ny=0; ny<Ny; ++ny)
143            dfdy(nx,ny,nz,i) = g[ny];
144        }
145    }
146
147   f.makeState(sxz, sy);
148   dfdy.makeSpectral();
149 }
```

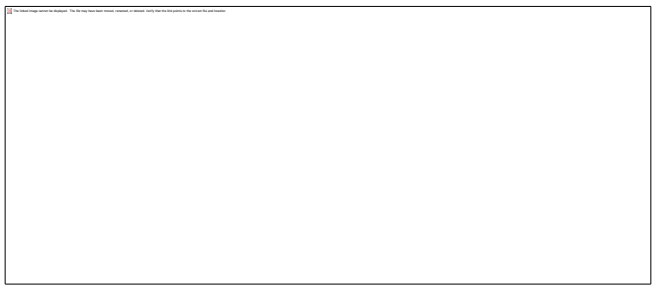
Here is the call graph for this function:

1.9.2.40 FlowField zdiff (const FlowField & *f*, int *n*)

References zdiff().

```
837                                {
838   FlowField g;
839   zdiff(f,g,n);
840   return g;
841 }
```

Here is the call graph for this function:



1.9.2.41 void zdiff (const FlowField & *f_*, FlowField & *dfdz*, int *n*)

References FlowField::a(), FlowField::b(), cferror(), FlowField::cmplx(), FlowField::congruent(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeSpectral_xz(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, pow(), FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

```
152                                        {
153   if (n==0)
154     return;
155
156   FlowField& f = (FlowField&) f_;
157   fieldstate sxz = f.xzstate();
158   fieldstate sy  = f.ystate();
159   f.makeSpectral_xz();
160
161   // compute gradf(nx,ny,nz,i) = df(nx,ny,nz,0)/dx_i for scalar-valued f
162   if (!f.congruent(dfdz))
163     dfdz.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
164   else
165     dfdz.setToZero();
```

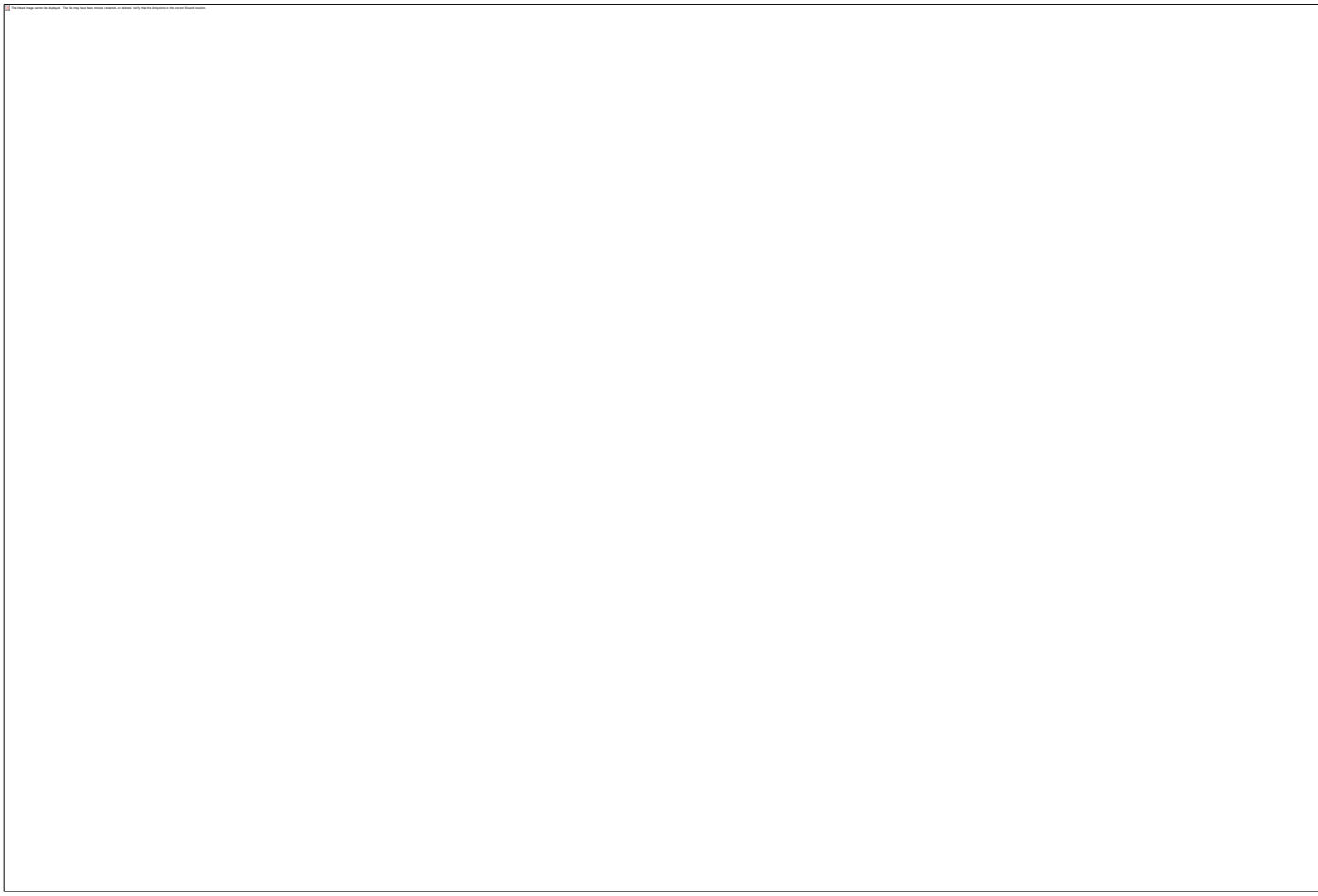
```
166   dfdz.setState(Spectral, sy);
167
168   Complex rot(0.0, 0.0);
169   switch(n % 4) {
170   case 0: rot = Complex( 1.0, 0.0); break;
171   case 1: rot = Complex( 0.0, 1.0); break;
172   case 2: rot = Complex(-1.0, 0.0); break;
173   case 3: rot = Complex( 0.0,-1.0); break;
174   default: cferror("zdiff(f, dfdz, n) : impossible: n % 4 > 4 !!"); break;
175   }
176
177   int Nd = f.Nd();
178   int Mx = f.Mx();
179   int My = f.My();
180   int Mz = f.Mz();
181   int kzmax = f.kzmax();
182   Real Lz = f.Lz();
183
184   Real az = 2*pi/Lz;
185   for (int i=0; i<Nd; ++i)
186     for (int my=0; my<My; ++my)
187       for (int mx=0; mx<Mx; ++mx)
188         for (int mz=0; mz<Mz; ++mz) {
189           int kz = f.kz(mz);
190           Complex cz = rot*(pow(az*kz, n)*zero_last_mode(kz,kzmax,n));
191           dfdz.cmplx(mx,my,mz,i) = cz*f.cmplx(mx,my,mz,i);
192         }
193
194   f.makeState(sxz,sy);
195   dfdz.makeSpectral();
196   return;
197 }
```

Here is the call graph for this function:

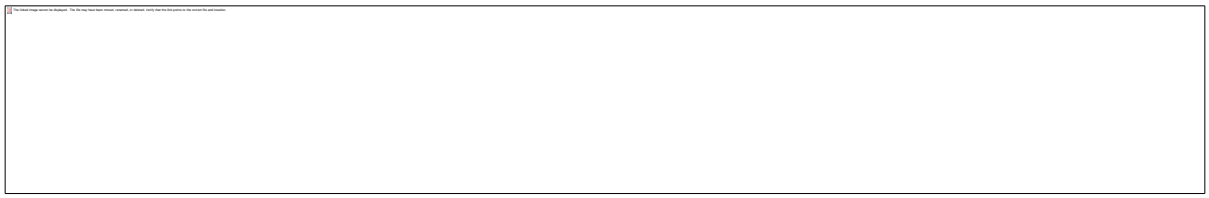
1.9.2.42

1.10 /_cuda/channelflow-0.9.22/channelflow/diffops.h File Reference

#include "channelflow/mathdefs.h"
#include "channelflow/vector.h"
#include "channelflow/chebyshev.h"
#include "channelflow/flowfield.h"
Include dependency graph for diffops.h:



This graph shows which files directly or indirectly include this file:



1.10.1 Enumerations

- enum NonlinearMethod
  { Rotational, Convection, Divergence, SkewSymmetric, Alternating, Alternating_, Linearized }

1.10.2 Functions

- FlowField xdiff (const FlowField &f, int n=1)
- FlowField ydiff (const FlowField &f, int n=1)
- FlowField zdiff (const FlowField &f, int n=1)
- FlowField diff (const FlowField &f, int nx, int ny, int nz)
- FlowField grad (const FlowField &f)
- FlowField lapl (const FlowField &f)
- FlowField curl (const FlowField &f)
- FlowField norm (const FlowField &f)
- FlowField norm2 (const FlowField &f)
- FlowField div (const FlowField &f)
- FlowField cross (const FlowField &f, const FlowField &g)
- FlowField outer (const FlowField &f, const FlowField &g)
- FlowField dot (const FlowField &f, const FlowField &g)
- FlowField energy (const FlowField &u)
- FlowField energy (const FlowField &u, ChebyCoeff &U)
- void xdiff (const FlowField &f, FlowField &dfdx, int n=1)
- void ydiff (const FlowField &f, FlowField &dfdy, int n=1)
- void zdiff (const FlowField &f, FlowField &dfdz, int n=1)
- void diff (const FlowField &f, FlowField &df, int nx, int ny, int nz)
- void grad (const FlowField &f, FlowField &grad_f)
- void lapl (const FlowField &f, FlowField &lapl_f)
- void curl (const FlowField &f, FlowField &curl_f)
- void norm (const FlowField &f, FlowField &norm_f)
- void norm2 (const FlowField &f, FlowField &norm2_f)
- void div (const FlowField &f, FlowField &divf)
- void cross (const FlowField &f, const FlowField &g, FlowField &f_cross_g)
- void outer (const FlowField &f, const FlowField &g, FlowField &fg)
- void dot (const FlowField &f, const FlowField &g, FlowField &f_dot_g)
- void energy (const FlowField &u, FlowField &e)
- void energy (const FlowField &u, const ChebyCoeff &U, FlowField &e)
- void rotationalNL (const FlowField &u, const ChebyCoeff &U, FlowField &f, FlowField &tmp)
- void convectionNL (const FlowField &u, const ChebyCoeff &U, FlowField &f, FlowField &tmp)
- void divergenceNL (const FlowField &u, const ChebyCoeff &U, FlowField &f, FlowField &tmp)
- void skewsymmetricNL (const FlowField &u, const ChebyCoeff &U, FlowField &f, FlowField &tmp)

- void skewsymmetricNL_GPU (const FlowField &u, const ChebyCoeff &U, FlowField &f, FlowField &tmp)
- void skewsymmetricNL_THREAD (const FlowField &u, const ChebyCoeff &U, FlowField &f, FlowField &tmp)
- void linearizedNL (const FlowField &u, const ChebyCoeff &U, FlowField &f)
- void navierstokesNL (const FlowField &u, const ChebyCoeff &U, FlowField &f, FlowField &tmp, NonlinearMethod &method)
- int i3j (int i, int j)
- std::ostream & operator<< (std::ostream &os, NonlinearMethod nonl)
- void uq2p (const FlowField &u, const FlowField &q, FlowField &p, NonlinearMethod nonlin_method)
- void up2q (const FlowField &u, const FlowField &p, FlowField &q, NonlinearMethod nonlin_method)
- void matlabSave (FlowField &f, int i, std::string &filename)

1.10.3 Enumeration Type Documentation

1.10.3.1 enum NonlinearMethod

Enumerator:

Rotational
Convection
Divergence
SkewSymmetric
Alternating
Alternating_
Linearized

```
109             {Rotational, Convection, Divergence, SkewSymmetric,
110              Alternating, Alternating_, Linearized};
```

1.10.4 Function Documentation

1.10.4.1 void convectionNL (const FlowField & *u*, const ChebyCoeff & *U*, FlowField & *f*, FlowField & *tmp*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::geomCongruent(), grad(), i3j(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), ChebyCoeff::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical,

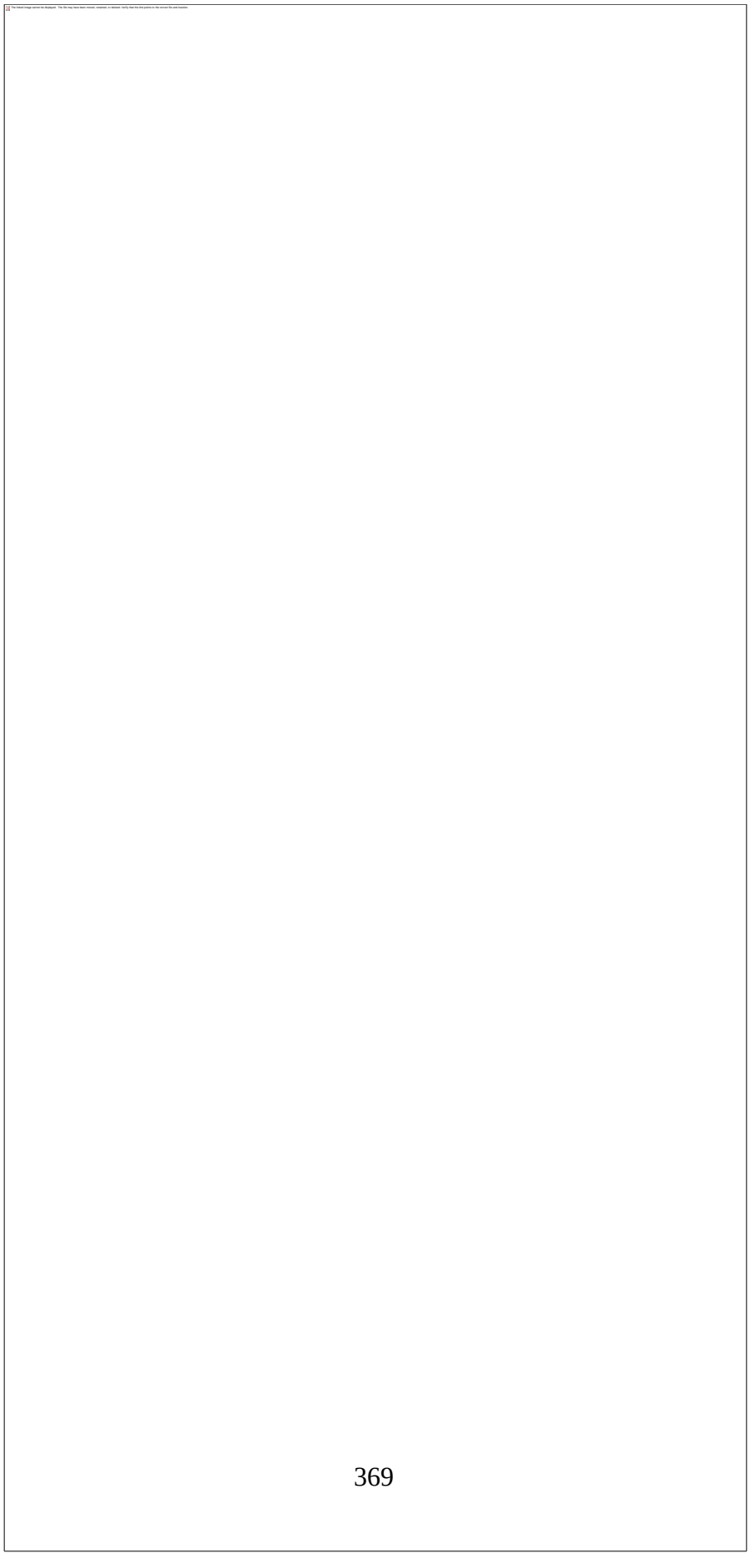
FlowField::resize(), FlowField::setState(), FlowField::setToZero(), ChebyCoeff::state(), FlowField::xzstate(), and FlowField::ystate().

Referenced by navierstokesNL().

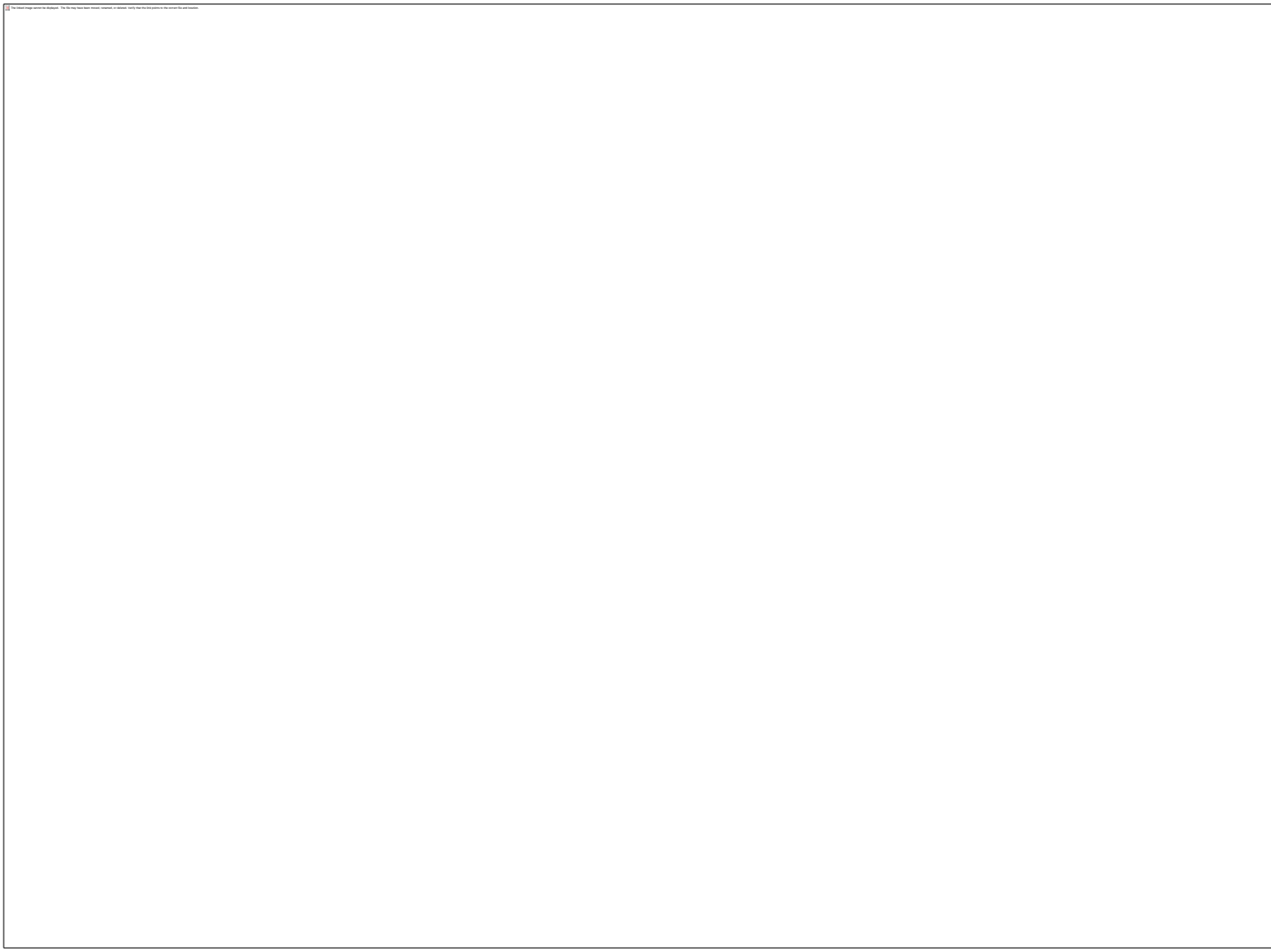
```
1031                                                                {
1032
1033  FlowField& u = (FlowField&) u_;
1034  ChebyCoeff& U = (ChebyCoeff&) U_;
1035  FlowField& grad_u = tmp;
1036
1037  assert(u.Nd() == 3);
1038  assert(U.numModes() == u.Ny());
1039  assert(U.a() == u.a() && U.b() == u.b());
1040
1041  fieldstate uxzstate = u.xzstate();
1042  fieldstate uystate  = u.ystate();
1043  fieldstate Ustate   = U.state();
1044
1045  if (!u.geomCongruent(f) || f.Nd() != 3)
1046    f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
1047  else
1048    f.setToZero();
1049  f.setState(Physical, Physical);
1050
1051  if (!u.geomCongruent(grad_u) || grad_u.Nd() < 9)
1052    grad_u.resize(u.Nx(), u.Ny(), u.Nz(), 9, u.Lx(), u.Lz(), u.a(), u.b());
1053
1054  U.makeSpectral();
1055  u.makeSpectral();
1056  u += U;
1057
1058  grad(u, grad_u);
1059  grad_u.makePhysical();
1060  u.makePhysical();
1061
1062  int Nx = u.Nx();
1063  int Ny = u.Ny();
1064  int Nz = u.Nz();
1065  for (int i=0; i<3; ++i)
1066    for (int j=0; j<3; ++j) {
1067      int ij = i3j(i,j);
1068      for (int ny=0; ny<Ny; ++ny)
1069        for (int nx=0; nx<Nx; ++nx)
1070          for (int nz=0; nz<Nz; ++nz)
1071            f(nx,ny,nz,i) += u(nx,ny,nz,j)*grad_u(nx,ny,nz,ij);
1072    }
1073
```

```
1074   f.makeSpectral();
1075   u.makeSpectral();
1076   u -= U;
1077
1078   // Possible extra transforms if entry states were Physical
1079   u.makeState(uxzstate, uystate);
1080   U.makeState(Ustate);
1081 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



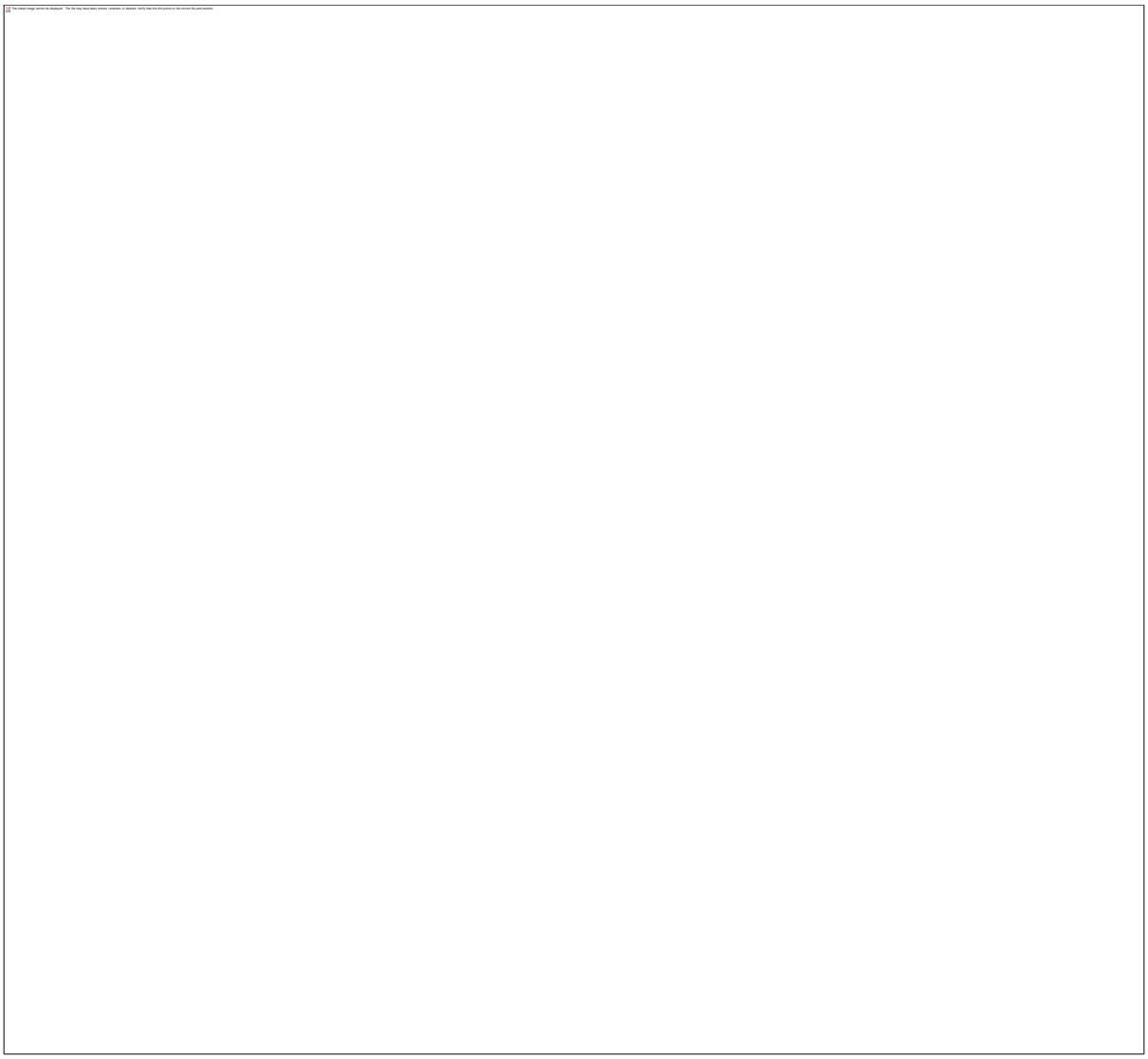
1.10.4.2 void cross (const FlowField & *f*, const FlowField & *g*, FlowField & *f_cross_g*)

References FlowField::a(), FlowField::b(), FlowField::congruent(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), FlowField::xzstate(), and FlowField::ystate().

```
716                                                 {
717   FlowField& f = (FlowField&) f_;
718   FlowField& g = (FlowField&) g_;
719   fieldstate fxz = f.xzstate();
720   fieldstate fy  = f.ystate();
```

```
721   fieldstate gxz = g.xzstate();
722   fieldstate gy  = g.ystate();
723   assert(g.congruent(f));
724   assert(f.Nd() == 3 && g.Nd() == 3);
725
726   f.makePhysical();
727   g.makePhysical();
728
729   if (!f.geomCongruent(fcg) || fcg.Nd() != 3)
730     fcg.resize(f.Nx(), f.Ny(), f.Nz(), 3, f.Lx(), f.Lz(), f.a(), f.b());
731   else
732     fcg.setToZero();
733   fcg.setState(Physical,Physical);
734
735   int Nx = f.Nx();
736   int Ny = f.Ny();
737   int Nz = f.Nz();
738   int Nd = f.Nd();
739   for (int i=0; i<Nd; ++i)
740     for (int ny=0; ny<Ny; ++ny)
741       for (int nx=0; nx<Nx; ++nx)
742         for (int nz=0; nz<Nz; ++nz)
743           fcg(nx,ny,nz,i) =
744             f(nx,ny,nz,(i+1)%3)*g(nx,ny,nz,(i+2)%3) -
745             f(nx,ny,nz,(i+2)%3)*g(nx,ny,nz,(i+1)%3);
746
747   f.makeState(fxz, fy);
748   g.makeState(gxz, gy);
749   fcg.makeSpectral();
750 }
```

Here is the call graph for this function:

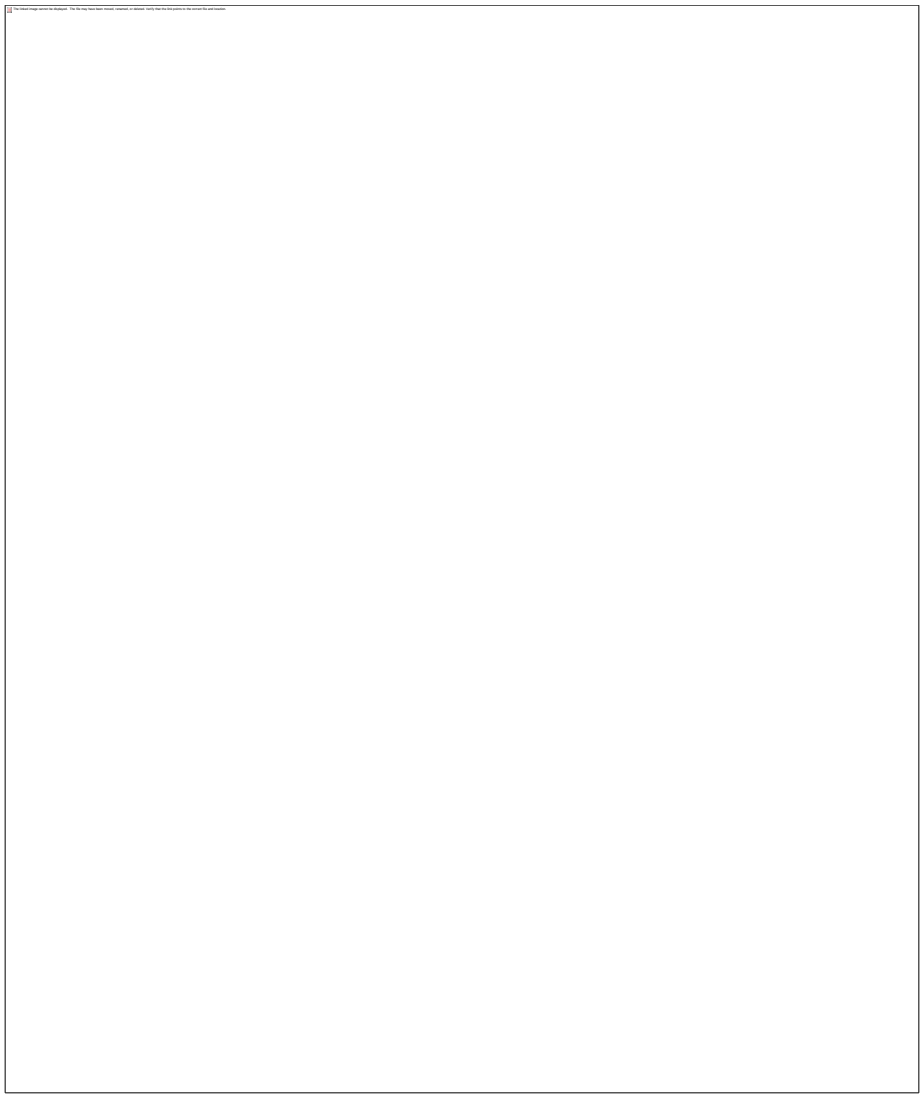


1.10.4.3 FlowField cross (const FlowField & *f*, const FlowField & *g*)

References cross().

```
859                                                 {
860   FlowField fxg; cross(f,g,fxg); return fxg;
861 }
```

Here is the call graph for this function:



1.10.4.4 void curl (const FlowField & *f*, FlowField & *curl_f*)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

```
489                                     {
490   FlowField& f = (FlowField&) f_;
491   assert(f.Nd() == 3);
492   fieldstate sxz = f.xzstate();
493   fieldstate sy  = f.ystate();
```

```
494  f.makeSpectral();
495
496  if (!f.geomCongruent(curlf) || curlf.vectorDim() != 3)
497    curlf.resize(f.Nx(), f.Ny(), f.Nz(), 3, f.Lx(), f.Lz(), f.a(), f.b());
498  else
499    curlf.setToZero();
500  curlf.setState(Spectral, Spectral);
501
502  int Mx = f.Mx();
503  int My = f.My();
504  int Mz = f.Mz();
505  int kxmax = f.kxmax();
506  int kzmax = f.kzmax();
507  Real Lx = f.Lx();
508  Real Lz = f.Lz();
509
510  // curl_x = w_y - v_z
511  // curl_y = u_z - w_x
512  // curl_z = v_x - u_y
513
514  // Assign d/dx and d/dz terms to curl
515  for (int my=0; my<My; ++my)
516    for (int mx=0; mx<Mx; ++mx) {
517      int kx = f.kx(mx);
518      Complex cx(0.0, 2.0*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
519      for (int mz=0; mz<Mz; ++mz) {
520        int kz = f.kz(mz);
521        Complex cz(0.0, 2.0*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
522        Complex u = f.cmplx(mx,my,mz,0);
523        Complex v = f.cmplx(mx,my,mz,1);
524        Complex w = f.cmplx(mx,my,mz,2);
525        curlf.cmplx(mx,my,mz,0) = -cz*v;
526        curlf.cmplx(mx,my,mz,1) = cz*u - cx*w;
527        curlf.cmplx(mx,my,mz,2) = cx*v;
528      }
529    }
530
531  // Assign df_i/dy to Df(i,1)
532  ComplexChebyCoeff w(f.Ny(),  f.a(), f.b(), Spectral);
533  ComplexChebyCoeff wy(f.Ny(), f.a(), f.b(), Spectral);
534  ComplexChebyCoeff u(f.Ny(),  f.a(), f.b(), Spectral);
535  ComplexChebyCoeff uy(f.Ny(), f.a(), f.b(), Spectral);
536  for (int mx=0; mx<Mx; ++mx)
537    for (int mz=0; mz<Mz; ++mz) {
538      for (int my=0; my<My; ++my) {
539        u.set(my, f.cmplx(mx,my,mz,0));
```

```
540         w.set(my, f.cmplx(mx,my,mz,2));
541       }
542       diff(u,uy);
543       diff(w,wy);
544       for (int my=0; my<My; ++my) {
545         curlf.cmplx(mx,my,mz,0) += wy[my];
546         curlf.cmplx(mx,my,mz,2) -= uy[my];
547       }
548     }
549   f.makeState(sxz, sy);
550 }
```

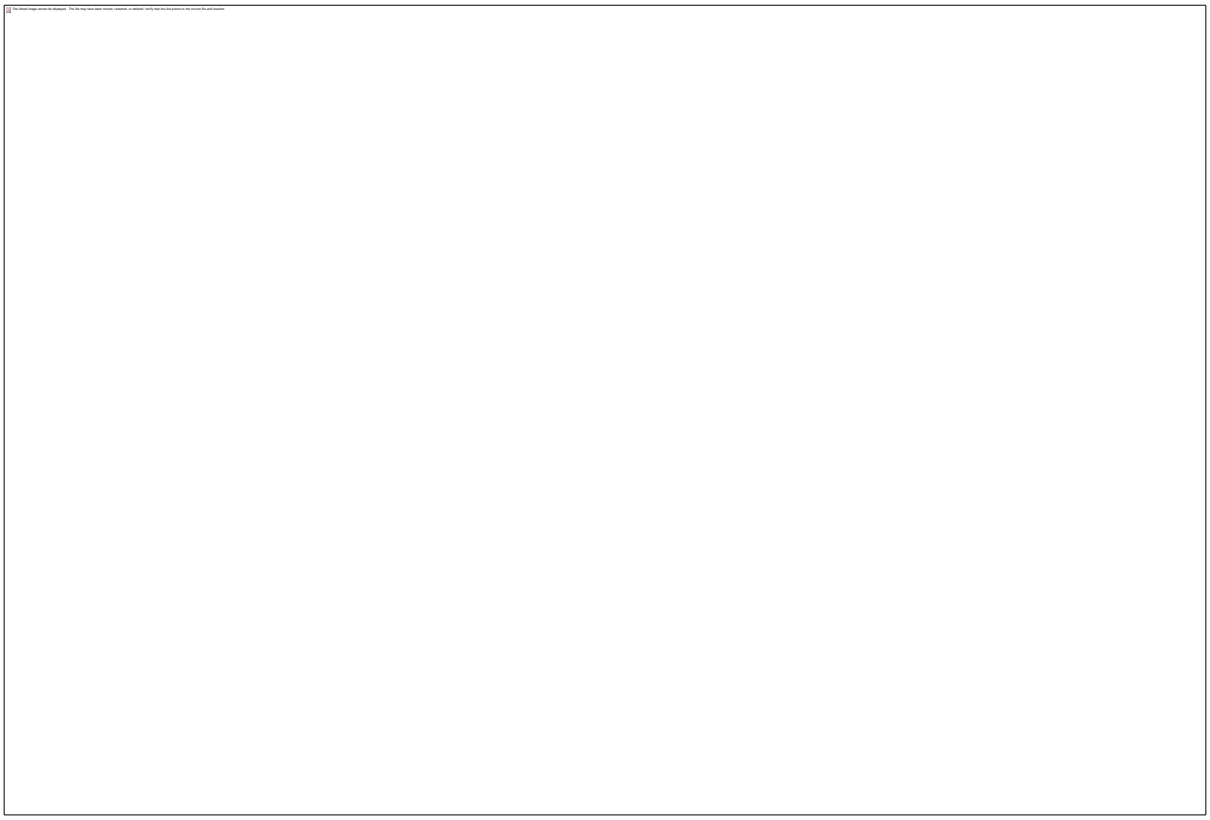
Here is the call graph for this function:

1.10.4.5 FlowField curl (const FlowField & *f*)

References curl().

```
851 {FlowField g;  curl(f,g); return g;}
```

Here is the call graph for this function:



1.10.4.6 void diff (const FlowField & *f*, FlowField & *df*, int *nx*, int *ny*, int *nz*)

References FlowField::a(), FlowField::b(), cferror(), FlowField::cmplx(), FlowField::congruent(), diff(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, pow(), FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, swap(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

```
199                                                 {
200   FlowField& f = (FlowField&) f_;
201   fieldstate sxz = f.xzstate();
```

```
202  fieldstate sy  = f.ystate();
203  f.makeSpectral();
204
205  Real Lx = f.Lx();
206  Real Lz = f.Lz();
207
208  if (!f.congruent(df))
209    df.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
210  else
211    df.setToZero();
212  df.setState(Spectral, Spectral);
213
214  Complex xrot(0.0, 0.0);
215  switch(nx % 4) {
216  case 0: xrot = Complex( 1.0, 0.0); break;
217  case 1: xrot = Complex( 0.0, 1.0); break;
218  case 2: xrot = Complex(-1.0, 0.0); break;
219  case 3: xrot = Complex( 0.0,-1.0); break;
220  default: cferror("diff(f, df,nx,ny,nz) : impossible: nx % 4 > 4 !!"); break;
221  }
222  Complex zrot(0.0, 0.0);
223  switch(nz % 4) {
224  case 0: zrot = Complex( 1.0, 0.0); break;
225  case 1: zrot = Complex( 0.0, 1.0); break;
226  case 2: zrot = Complex(-1.0, 0.0); break;
227  case 3: zrot = Complex( 0.0,-1.0); break;
228  default: cferror("diff(f, df,nx,ny,nz) : impossible: nz % 4 > 4 !!"); break;
229  }
230
231  int Nd = f.Nd();
232  int Mx = f.Mx();
233  int My = f.My();
234  int Mz = f.Mz();
235  int kxmax = f.kxmax();
236  int kzmax = f.kzmax();
237
238  // Do the x and z differentiation
239  Real ax = 2*pi/Lx;
240  Real az = 2*pi/Lz;
241  for (int i=0; i<Nd; ++i)
242    for (int my=0; my<My; ++my)
243      for (int mx=0; mx<Mx; ++mx) {
244        int kx = f.kx(mx);
245        Complex cx = xrot*(pow(ax*kx, nx)*zero_last_mode(kx,kxmax,nx));
246        for (int mz=0; mz<Mz; ++mz) {
247          int kz = f.kz(mz);
```

```
248         Complex cz = zrot*(pow(az*kz, nz)*zero_last_mode(kz,kzmax,nz));
249         df.cmplx(mx,my,mz,i) = cx*cz*f.cmplx(mx,my,mz,i);
250       }
251     }
252
253   // Do the y differentiation
254   ComplexChebyCoeff  g(f.Ny(), f.a(), f.b(), Spectral);
255   ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
256
257   for (int i=0; i<Nd; ++i)
258     for (int mx=0; mx<Mx; ++mx)
259       for (int mz=0; mz<Mz; ++mz) {
260         for (int my=0; my<My; ++my)
261           g.set(my, df.cmplx(mx,my,mz,i));
262
263         // differentiate n times and leave derivative in *g
264         for (int k=0; k<ny; ++k) {
265           diff(g,gy);
266           swap(g,gy);
267         }
268         for (int my=0; My<My; ++my)
269           df.cmplx(mx,my,mz,i) = g[my];
270       }
271
272   f.makeState(sxz,sy);
273   df.makeSpectral();
274   return;
275 }
```

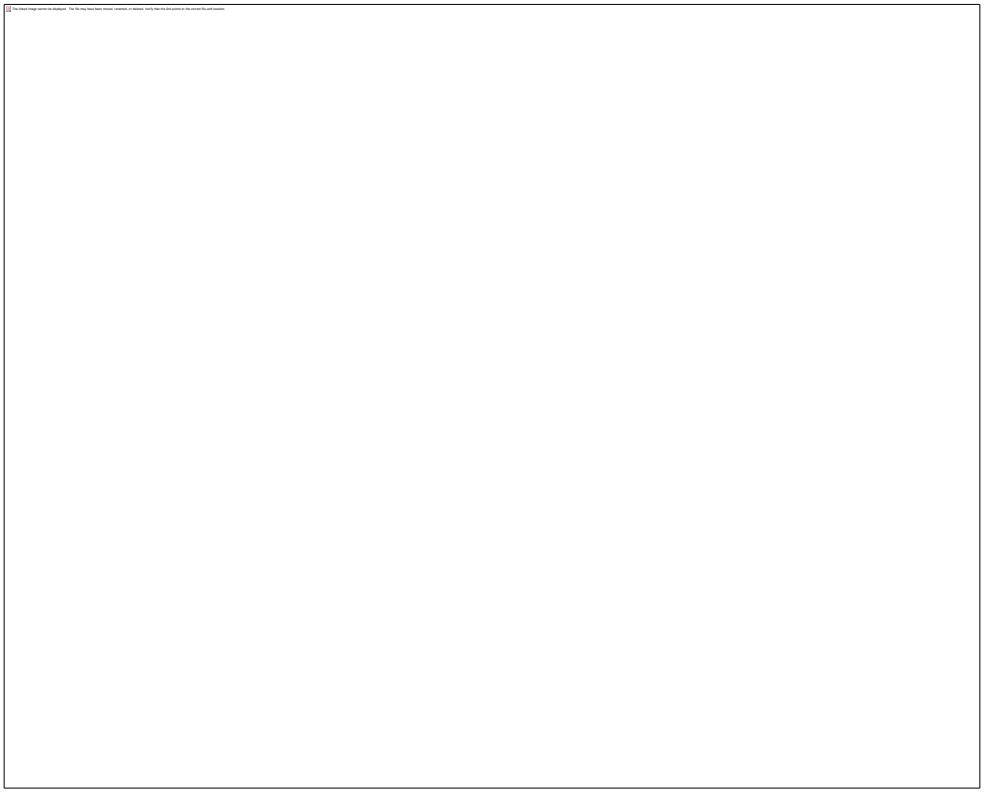
Here is the call graph for this function:

1.10.4.7 FlowField diff (const FlowField & *f*, int *nx*, int *ny*, int *nz*)

References diff().

```
843                                         {
844   FlowField g;
845   diff(f,g,nx,ny,nz);
846   return g;
847 }
```

Here is the call graph for this function:



1.10.4.8 void div (const FlowField & *f*, FlowField & *divf*)

References FlowField::a(), FlowField::b(), cferror(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), i3j(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

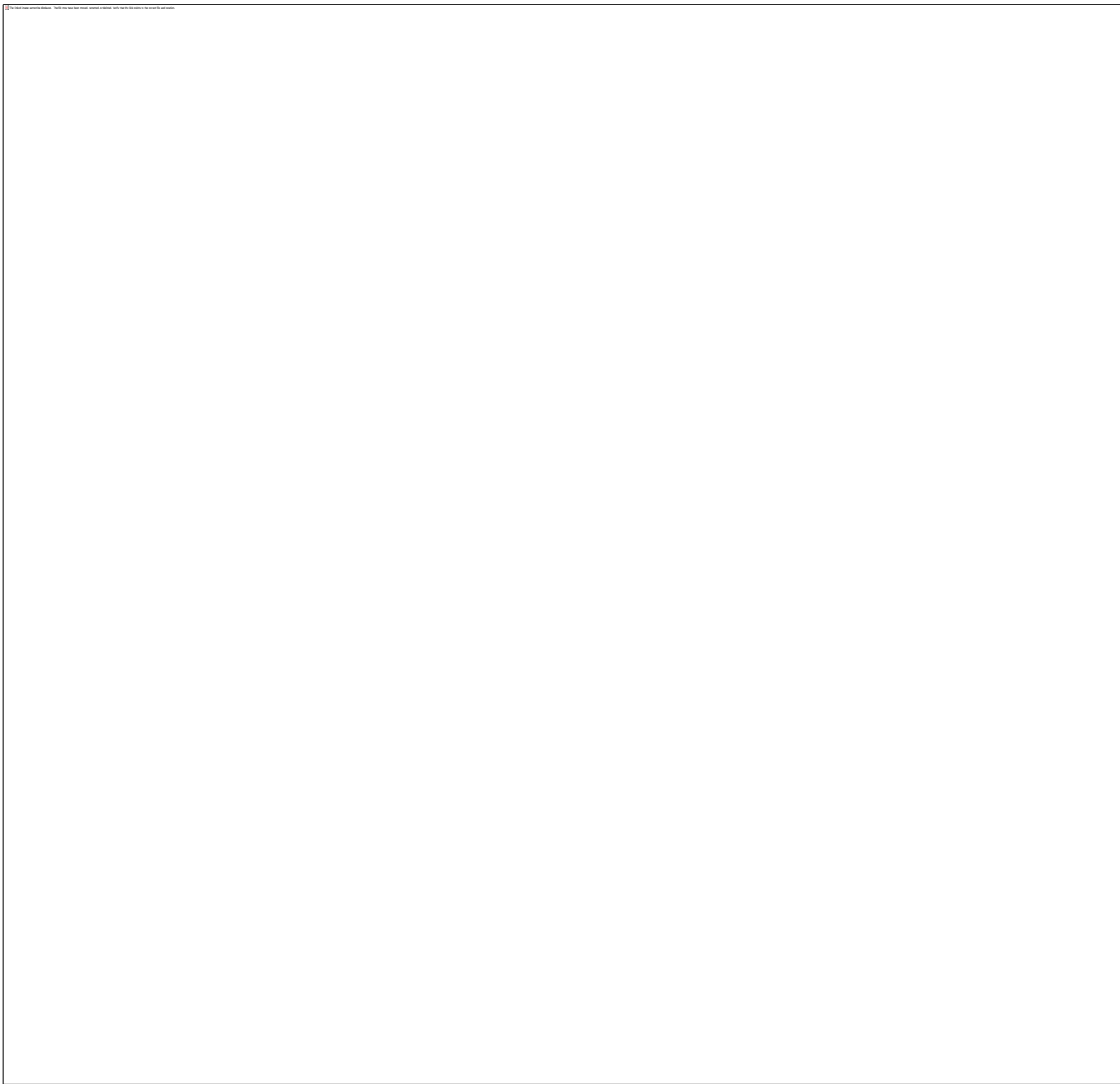
Referenced by chebyProfile(), div(), divDist2(), divergence(), divergenceNL(), divNorm2(), randomProfile(), FlowField::saveDivSpectrum(), PressureSolver::solve(), and PressureSolver::verify().

```
626                                     {
627   FlowField& f = (FlowField&) f_;
628   fieldstate sxz = f.xzstate();
629   fieldstate sy  = f.ystate();
630   f.makeSpectral();
631
632   int Mx = f.Mx();
633   int My = f.My();
634   int Mz = f.Mz();
635   int kxmax = f.kxmax();
636   int kzmax = f.kzmax();
637   Real Lx = f.Lx();
638   Real Lz = f.Lz();
639
640   if (f.Nd() == 3) {
641     if (!f.geomCongruent(divf) || divf.Nd() != 1)
642       divf.resize(f.Nx(), f.Ny(), f.Nz(), 1, f.Lx(), f.Lz(), f.a(), f.b());
643     else
644       divf.setToZero();
645     divf.setState(Spectral, Spectral);
646
647     // Add df0/dx + df2/dz to divf
648     for (int my=0; my<My; ++my)
649       for (int mx=0; mx<Mx; ++mx) {
650         int kx = f.kx(mx);
651         Complex cxkx(0.0, 2.0*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
652         for (int mz=0; mz<Mz; ++mz) {
653           int kz = f.kz(mz);
654           Complex czkz(0.0, 2.0*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
655           divf.cmplx(mx,my,mz,0)
656             = f.cmplx(mx,my,mz,0)*cxkx + f.cmplx(mx,my,mz,2)*czkz;
657         }
658       }
659
660     // Add df1/dy to to divf
661     ComplexChebyCoeff g(f.Ny(),  f.a(), f.b(), Spectral);
662     ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
663     for (int mx=0; mx<Mx; ++mx)
664       for (int mz=0; mz<Mz; ++mz) {
665         for (int my=0; my<My; ++my)
666           g.set(my, f.cmplx(mx,my,mz,1));
667         diff(g,gy);
668         for (int my=0; my<My; ++my)
```

```
669         divf.cmplx(mx,my,mz,0) += gy[my];
670       }
671     }
672     else if (f.Nd() == 9) {
673       if (!f.geomCongruent(divf) || divf.Nd() != 3)
674         divf.resize(f.Nx(), f.Ny(), f.Nz(), 3, f.Lx(), f.Lz(), f.a(), f.b());
675       else
676         divf.setToZero();
677       divf.setState(Spectral, Spectral);
678
679       for (int j=0; j<3; ++j) {
680
681         // Add df0j/dx + df2j/dz to divfj
682         for (int my=0; my<My; ++my)
683           for (int mx=0; mx<Mx; ++mx) {
684             int kx = f.kx(mx);
685             Complex cxkx(0.0, 2.0*pi*f.kx(mx)/Lx*zero_last_mode(kx,kxmax,1));
686             for (int mz=0; mz<Mz; ++mz) {
687               int kz = f.kz(mz);
688               Complex czkz(0.0, 2.0*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
689                 divf.cmplx(mx,my,mz,j) +=
690                   f.cmplx(mx,my,mz,i3j(0,j))*cxkx +
691                   f.cmplx(mx,my,mz,i3j(2,j))*czkz;
692             }
693           }
694
695         // Add df1j/dy to to divf
696         ComplexChebyCoeff g(f.Ny(),  f.a(), f.b(), Spectral);
697         ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
698         for (int mx=0; mx<Mx; ++mx)
699           for (int mz=0; mz<Mz; ++mz) {
700             for (int my=0; my<My; ++my)
701               g.set(my, f.cmplx(mx,my,mz,i3j(1,j)));
702             diff(g,gy);
703             for (int my=0; my<My; ++my)
704               divf.cmplx(mx,my,mz,j) += gy[my];
705           }
706       }
707     }
708     else
709       cferror("div(FlowField f, FlowField divf): f must be 3d or 9d");
710
711     f.makeState(sxz, sy);
712     divf.makeSpectral();
713     return;
714   }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.10.4.9 FlowField div (const FlowField & *f*)

References div().

```
854 {FlowField g;   div(f,g); return g;}
```

Here is the call graph for this function:

1.10.4.10 void divergenceNL (const FlowField & *u*, const ChebyCoeff & *U*, FlowField & *f*, FlowField & *tmp*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), div(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), outer(), FlowField::resize(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), and FlowField::ystate().

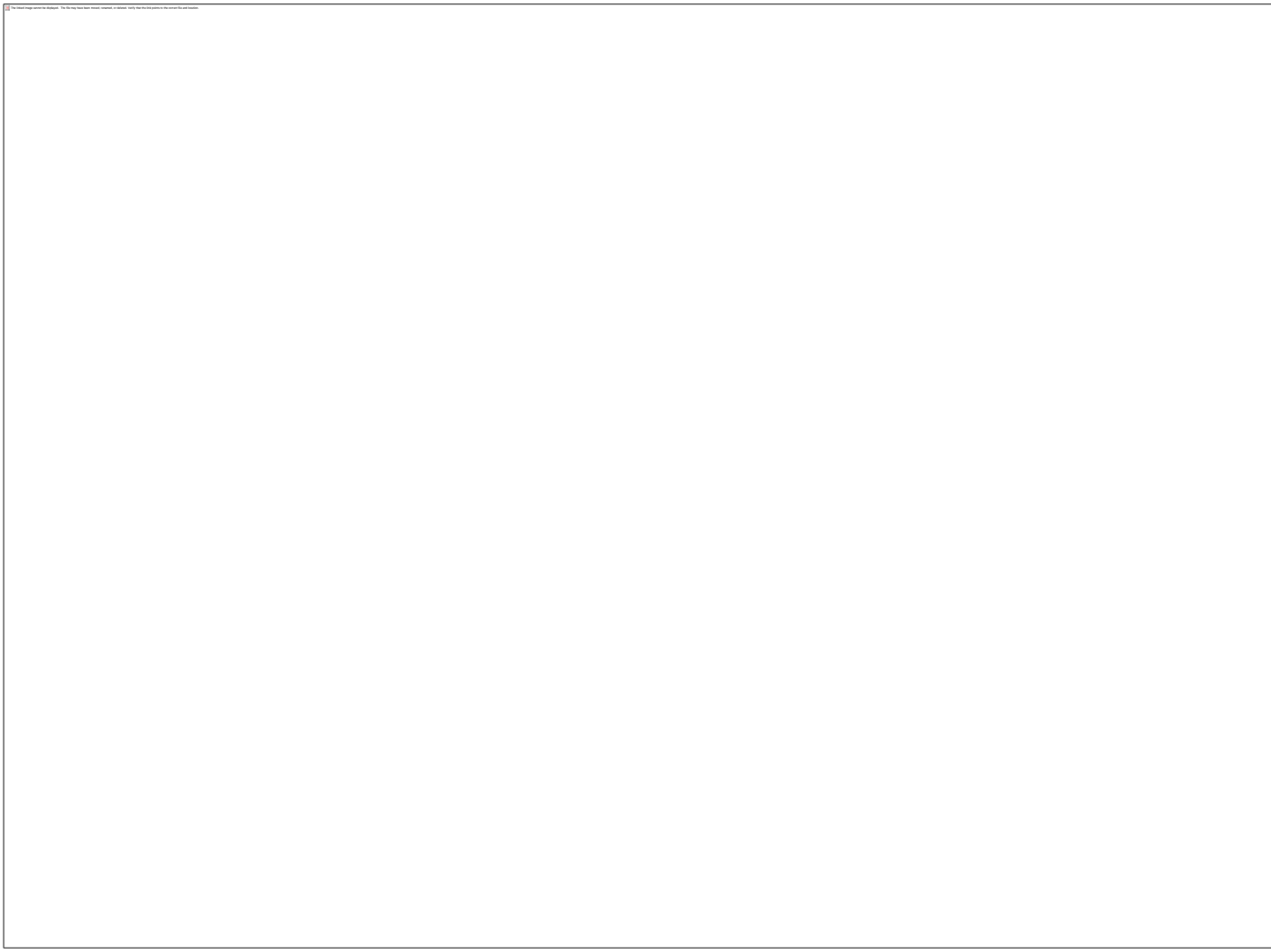
Referenced by navierstokesNL().

```
1083                                                    {
1084   FlowField& u = (FlowField&) u_;
1085   ChebyCoeff& U = (ChebyCoeff&) U_;
1086   FlowField& uu = tmp;
1087
1088   assert(u.Nd() == 3);
1089   assert(U.numModes() == u.Ny());
1090   assert(U.a() == u.a() && U.b() == u.b());
1091
1092   fieldstate uxzstate = u.xzstate();
1093   fieldstate uystate  = u.ystate();
1094   fieldstate Ustate   = U.state();
1095
1096   if (!u.geomCongruent(f) || f.Nd() != 3)
1097     f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
1098
1099   if (!u.geomCongruent(uu) || uu.Nd() < 9)
1100     uu.resize(u.Nx(), u.Ny(), u.Nz(), 9, u.Lx(), u.Lz(), u.a(), u.b());
1101
1102   // Possible extra transforms if entry states are Physical
1103   u.makeState(Spectral, uystate);
1104   U.makeState(uystate);
1105   u += U;
1106
1107   u.makePhysical();
1108   outer(u,u, uu);
1109
1110   uu.makeSpectral();
1111   div(uu, f);         // produces f spectral
1112
1113   u.makeState(Spectral, uystate);
1114   u -= U;
1115
1116   // Possible extra transforms if entry states are Physical
```

```
1117   u.makeState(uxzstate, uystate);
1118   U.makeState(Ustate);
1119
1120   return;
1121 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.10.4.11 void dot (const FlowField & *f*, const FlowField & *g*, FlowField & *f_dot_g*)

References FlowField::a(), FlowField::b(), FlowField::congruent(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), FlowField::xzstate(), and FlowField::ystate().

```
591                                                 {
592   FlowField& f = (FlowField&) f_;
593   FlowField& g = (FlowField&) g_;
594   fieldstate fxz = f.xzstate();
595   fieldstate fy  = f.ystate();
```

```
596   fieldstate gxz = g.xzstate();
597   fieldstate gy  = g.ystate();
598
599   assert(f.congruent(g));
600   f.makePhysical();
601   g.makePhysical();
602
603   if (!f.geomCongruent(fdotg) || fdotg.Nd() != 1)
604     fdotg.resize(f.Nx(), f.Ny(), f.Nz(), 1, f.Lx(), f.Lz(), f.a(),f.b());
605   else
606     fdotg.setToZero();
607   fdotg.setState(Physical, Physical);
608
609   int Nx = f.Nx();
610   int Ny = f.Ny();
611   int Nz = f.Nz();
612   int Nd = f.Nd();
613
614   for (int i=0; i<Nd; ++i)
615     for (int ny=0; ny<Ny; ++ny)
616       for (int nx=0; nx<Nx; ++nx)
617         for (int nz=0; nz<Nz; ++nz)
618           fdotg(nx,ny,nz,0) += f(nx,ny,nz,i)*g(nx,ny,nz,i);
619
620   f.makeState(fxz, fy);
621   g.makeState(gxz, gy);
622   fdotg.makeSpectral();
623   return;
624 }
```

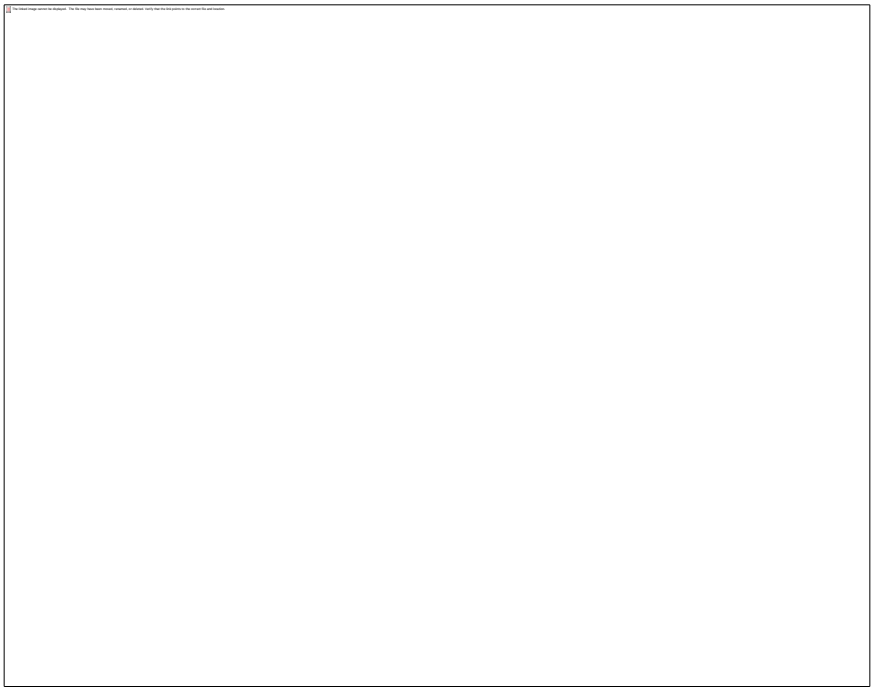
Here is the call graph for this function:

1.10.4.12 FlowField dot (const FlowField & *f*, const FlowField & *g*)

References dot().

```
862                                         {
863   FlowField fdotg; dot(f,g,fdotg); return fdotg;
864 }
```

Here is the call graph for this function:



1.10.4.13 void energy (const FlowField & *u*, const ChebyCoeff & *U*, FlowField & *e*)

References FlowField::a(), FlowField::b(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), ChebyCoeff::makePhysical(), FlowField::makePhysical(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), square(), ChebyCoeff::state(), FlowField::xzstate(), and FlowField::ystate().

```
781                                             {
782
783   FlowField& u = (FlowField&) u_;
784   ChebyCoeff& U = (ChebyCoeff&) U_;
785   fieldstate uxzstate = u.xzstate();
```

```
786  fieldstate uystate  = u.ystate();
787  fieldstate Ustate   = U.state();
788  assert(U.numModes() == u.Ny());
789
790  if (!u.geomCongruent(e) || e.Nd() != 1)
791    e.resize(u.Nx(), u.Ny(), u.Nz(), 1, u.Lx(), u.Lz(), u.a(), u.b());
792  else
793    e.setToZero();
794  e.setState(Physical,Physical);
795
796  u.makePhysical();
797  U.makePhysical();
798
799  e.setToZero();
800  e.setState(Physical, Physical);
801
802  int Nx = u.Nx();
803  int Ny = u.Ny();
804  int Nz = u.Nz();
805  int Nd = u.Nd();
806
807  for (int ny=0; ny<Ny; ++ny)
808    for (int nx=0; nx<Nx; ++nx)
809      for (int nz=0; nz<Nz; ++nz)
810        e(nx,ny,nz,0) += 0.5*square(u(nx,ny,nz,0)+U(ny));
811
812  for (int i=1; i<Nd; ++i)
813    for (int ny=0; ny<Ny; ++ny)
814      for (int nx=0; nx<Nx; ++nx)
815        for (int nz=0; nz<Nz; ++nz)
816          e(nx,ny,nz,0) += 0.5*square(u(nx,ny,nz,i));
817
818  e.makeSpectral();
819  u.makeState(uxzstate,uystate);
820  U.makeState(Ustate);
821
822  return;
823 }
```

Here is the call graph for this function:

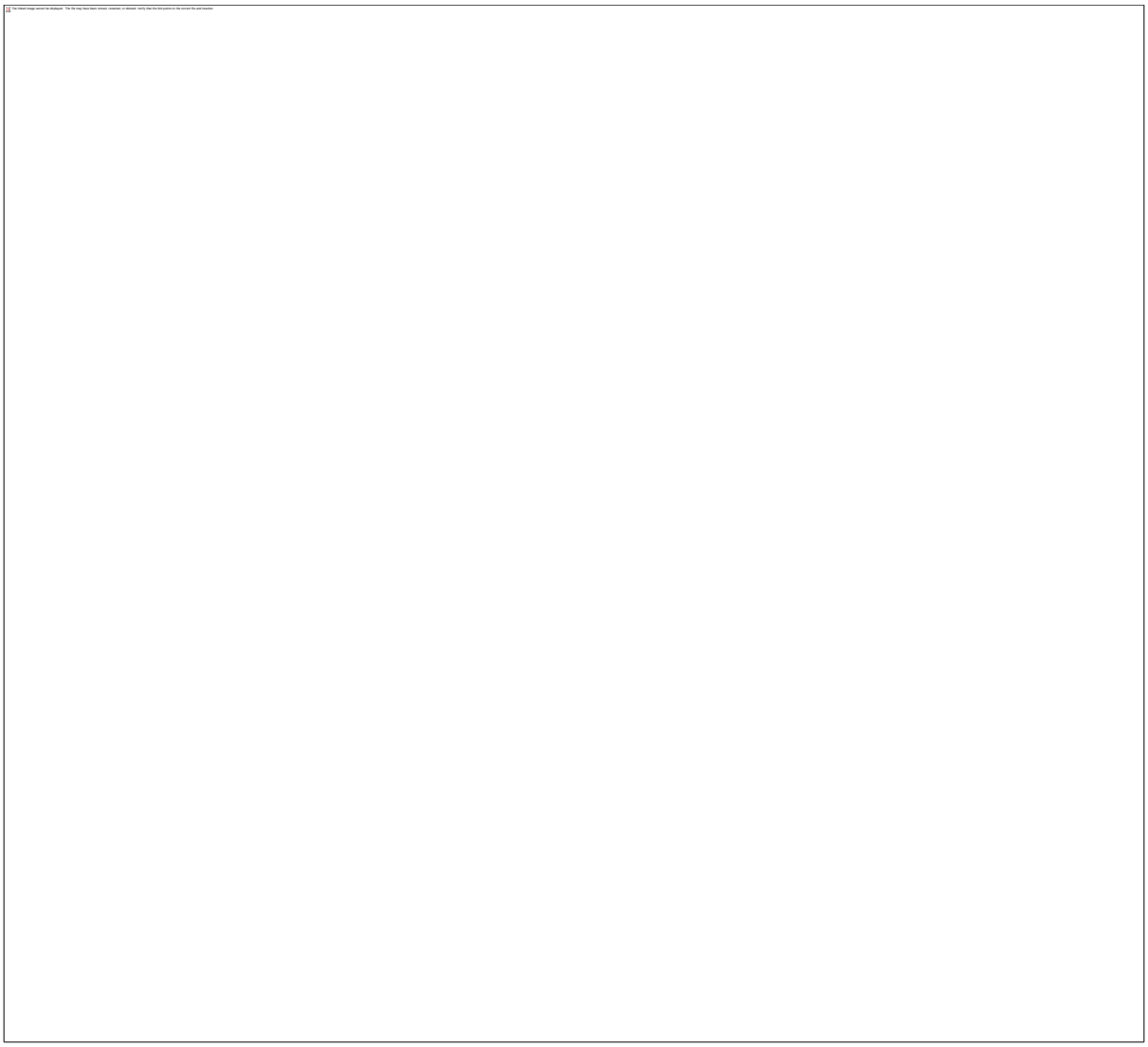
1.10.4.14 void energy (const FlowField & *u*, FlowField & *e*)

References FlowField::a(), FlowField::b(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), square(), FlowField::xzstate(), and FlowField::ystate().

Referenced by energy().

```
752                                     {
753   FlowField& u = (FlowField&) u_;
754   fieldstate sxz = u.xzstate();
755   fieldstate sy  = u.ystate();
756   u.makePhysical();
757
758   if (!u.geomCongruent(e) || e.Nd() != 1)
759     e.resize(u.Nx(), u.Ny(), u.Nz(), 1, u.Lx(), u.Lz(), u.a(), u.b());
760   else
761     e.setToZero();
762   e.setState(Physical,Physical);
763
764   int Nx = u.Nx();
765   int Ny = u.Ny();
766   int Nz = u.Nz();
767   int Nd = u.Nd();
768
769   for (int i=0; i<Nd; ++i)
770     for (int ny=0; ny<Ny; ++ny)
771       for (int nx=0; nx<Nx; ++nx)
772         for (int nz=0; nz<Nz; ++nz)
773           e(nx,ny,nz,0) += 0.5*square(u(nx,ny,nz,i));
774
775   e.makeSpectral();
776   u.makeState(sxz, sy);
777   return;
778 }
```

Here is the call graph for this function:



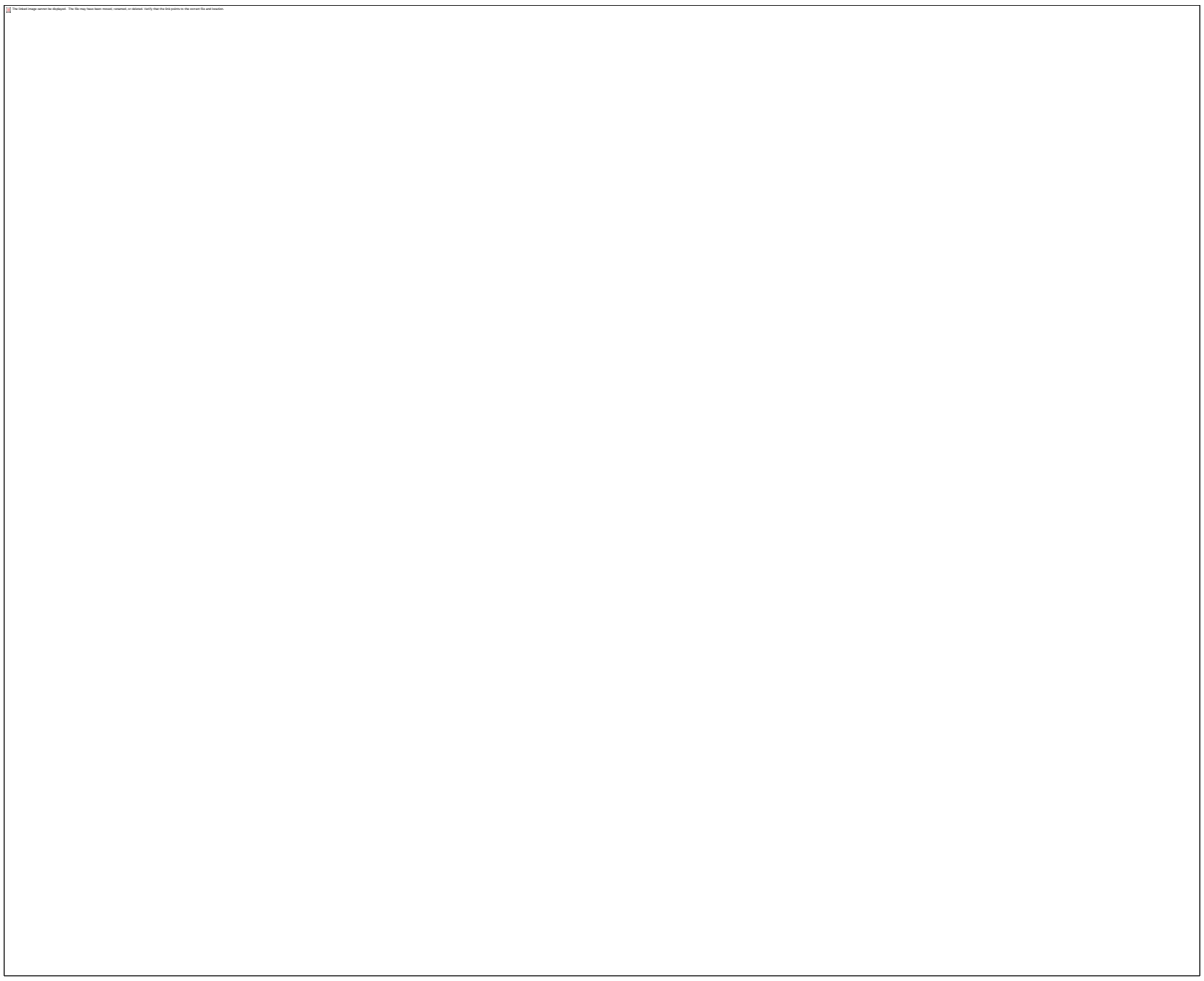
Here is the caller graph for this function:

1.10.4.15 FlowField energy (const FlowField & *u*, ChebyCoeff & *U*)

References energy().

```
868                                          {
869   FlowField e; energy(u,U,e); return e;
870 }
```

Here is the call graph for this function:



1.10.4.16 FlowField energy (const FlowField & *u*)

References energy().

```
865                                 {
866   FlowField e; energy(u,e); return e;
867 }
```

Here is the call graph for this function:

1.10.4.17 void grad (const FlowField & *f*, FlowField & *grad_f*)

References FlowField::a(), FlowField::b(), cferror(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), i3j(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

Referenced by PPEDNS::advance(), convectionNL(), grad(), gradient(), PPEDNS::PPEDNS(), PPEDNS::push(), skewsymmetricNL(), skewsymmetricNL_GPU(), and skewsymmetricNL_THREAD().

```
280                                     {
281
282   FlowField& f = (FlowField&) f_;
283   fieldstate sxz = f.xzstate();
284   fieldstate sy  = f.ystate();
285   f.makeSpectral();
286
287   Real Lx = f.Lx();
288   Real Lz = f.Lz();
289
290   // compute gradf(nx,ny,nz,i) = df(nx,ny,nz,0)/dx_i for scalar-valued f
291   if (f.Nd() == 1) {
292     if (!f.geomCongruent(gradf) || gradf.vectorDim() != 3)
293       gradf.resize(f.Nx(), f.Ny(), f.Nz(), 3, f.Lx(), f.Lz(), f.a(), f.b());
294     else
295       gradf.setToZero();
296     gradf.setState(Spectral, Spectral);
297
298     int Mx = f.Mx();
299     int My = f.My();
300     int Mz = f.Mz();
301     int kxmax = f.kxmax();
302     int kzmax = f.kzmax();
303
304     // Assign df/dx to Df(i) and df/dz to Df(i)
305     for (int my=0; my<My; ++my)
306       for (int mx=0; mx<Mx; ++mx) {
307         int kx = f.kx(mx);
308         Complex cxkx(0.0, 2.0*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
309         for (int mz=0; mz<Mz; ++mz) {
310           int kz = f.kz(mz);
311           Complex czkz(0.0, 2.0*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
```

```
312         Complex fval = f.cmplx(mx,my,mz,0);
313         gradf.cmplx(mx,my,mz,0) = fval*cxkx;
314         gradf.cmplx(mx,my,mz,2) = fval*czkz;
315       }
316     }
317
318   // Assign df/dy to Df(i,1)
319   ComplexChebyCoeff g(f.Ny(),  f.a(), f.b(), Spectral);
320   ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
321   for (int mx=0; mx<Mx; ++mx)
322     for (int mz=0; mz<Mz; ++mz) {
323       for (int my=0; my<My; ++my)
324         g.set(my, f.cmplx(mx,my,mz,0));
325       diff(g,gy);
326       for (int my=0; my<My; ++my)
327         gradf.cmplx(mx,my,mz,1) = gy[my];
328     }
329 }
330 else if (f.Nd() == 3) {
331   if (!f.geomCongruent(gradf) || gradf.vectorDim() != 9)
332     gradf.resize(f.Nx(), f.Ny(), f.Nz(), 9, f.Lx(), f.Lz(), f.a(), f.b());
333   else
334     gradf.setToZero();
335   gradf.setState(Spectral, Spectral);
336
337   int Mx = f.Mx();
338   int My = f.My();
339   int Mz = f.Mz();
340   int Nd = 3;
341   int kxmax = f.kxmax();
342   int kzmax = f.kzmax();
343
344   // Assign df_i/dx to Df(i,0) and df_i/dz to Df(i,2)
345   for (int i=0; i<Nd; ++i)
346     for (int my=0; my<My; ++my)
347       for (int mx=0; mx<Mx; ++mx) {
348         int kx = f.kx(mx);
349         Complex cxkx(0.0, 2.0*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
350         for (int mz=0; mz<Mz; ++mz) {
351           int kz = f.kz(mz);
352           Complex czkz(0.0, 2.0*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
353           Complex fval = f.cmplx(mx,my,mz,i);
354           gradf.cmplx(mx,my,mz,i3j(i,0)) = fval*cxkx;
355           gradf.cmplx(mx,my,mz,i3j(i,2)) = fval*czkz;
356         }
357       }
```

```
358
359   // Assign df_i/dy to Df(i,1)
360   ComplexChebyCoeff g(f.Ny(),  f.a(), f.b(), Spectral);
361   ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
362   for (int i=0; i<Nd; ++i)
363     for (int mx=0; mx<Mx; ++mx)
364       for (int mz=0; mz<Mz; ++mz) {
365         for (int my=0; my<My; ++my)
366           g.set(my, f.cmplx(mx,my,mz,i));
367         diff(g,gy);
368         for (int my=0; my<My; ++my)
369           gradf.cmplx(mx,my,mz,i3j(i,1)) = gy[my];
370       }
371   }
372   else
373     cferror("grad(FlowField f, FlowField grad_f) : f must be 1d or 3d");
374
375   f.makeState(sxz, sy);
376   return;
377 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.10.4.18 FlowField grad (const FlowField & *f*)

References grad().

```
849 {FlowField g;  grad(f,g); return g;}
```

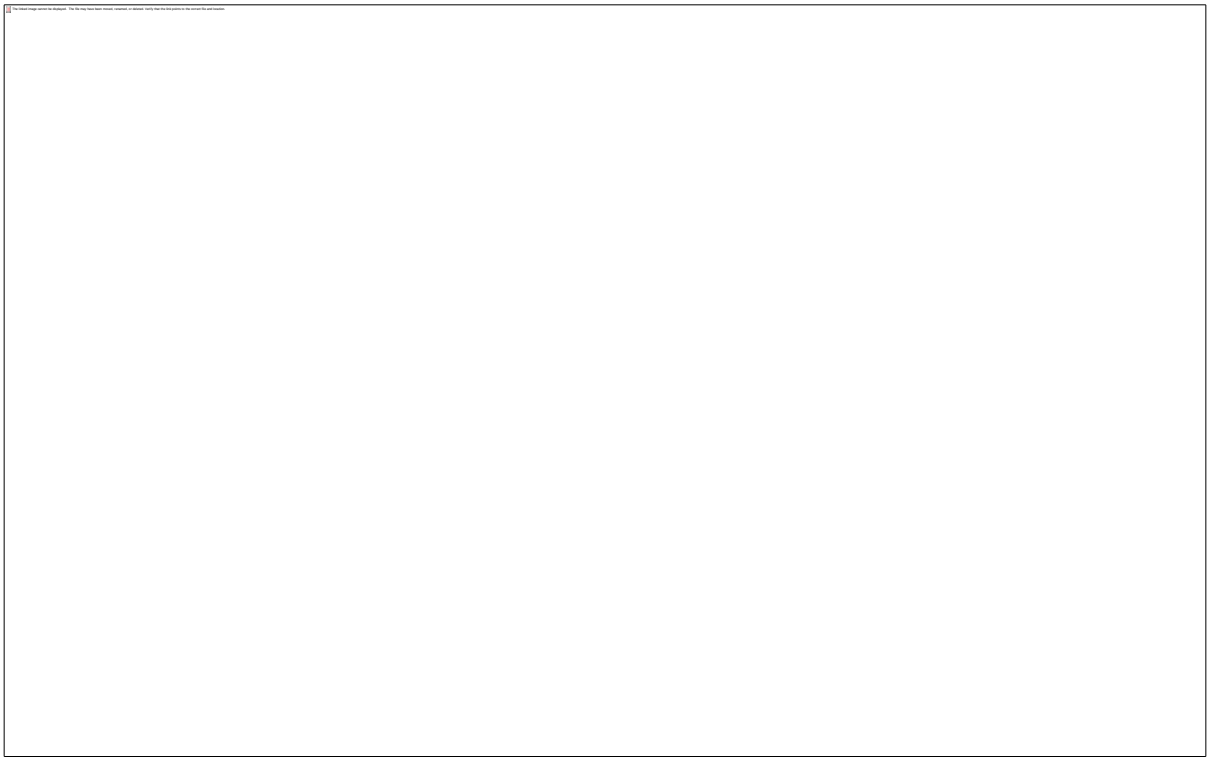
Here is the call graph for this function:

1.10.4.19 int i3j (int *i*, int *j*) [inline]

Referenced by convectionNL(), div(), grad(), skewsymmetricNL_task4::Run(), skewsymmetricNL_task3::Run(), skewsymmetricNL_task::Run(), skewsymmetricNL(), skewsymmetricNL_GPU(), and skewsymmetricNL_THREAD().

```
137 {return 3*i + j;}
```

Here is the caller graph for this function:



1.10.4.20 void lapl (const FlowField & *f*, FlowField & *lapl_f*)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), diff2(), FlowField::kx(), FlowField::kz(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, square(), FlowField::xzstate(), and FlowField::ystate().

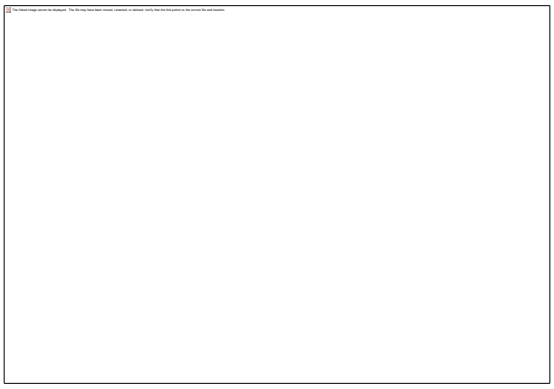
Referenced by lapl(), laplacian(), PressureSolver::verify(), and PoissonSolver::verify().

```
379                                     {
380   FlowField& f = (FlowField&) f_;
381   fieldstate sxz = f.xzstate();
```

```
382  fieldstate sy  = f.ystate();
383  f.makeSpectral();
384
385  if (!f.congruent(laplf))
386    laplf.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
387  else
388    laplf.setToZero();
389  laplf.setState(Spectral, Spectral);
390
391  // compute gradf(nx,ny,nz,i) = df(nx,ny,nz,0)/dx_i for scalar-valued f
392  Real Lx = f.Lx();
393  Real Lz = f.Lz();
394  int Mx = f.Mx();
395  int My = f.My();
396  int Mz = f.Mz();
397  int Nd = f.Nd();
398
399  ComplexChebyCoeff g(f.Ny(),  f.a(), f.b(), Spectral);
400  ComplexChebyCoeff gyy(f.Ny(), f.a(), f.b(), Spectral);
401
402  Real az = -1.0*square(2.0*pi/Lz);
403  for (int i=0; i<Nd; ++i) {
404    // Assign lapl f_i = (d^2/dx^2 + d^2/dz^2) f_i
405    for (int my=0; my<My; ++my)
406      for (int mx=0; mx<Mx; ++mx) {
407        int kx = f.kx(mx);
408        Complex cx = -1.0*square(2.0*pi*kx/Lx);
409        for (int mz=0; mz<Mz; ++mz)
410          laplf.cmplx(mx,my,mz,i) =
411            (cx + az*square(f.kz(mz)))*f.cmplx(mx,my,mz,i);
412      }
413
414    // Add d^2/dy^2 f_i on to previous result
415    for (int mx=0; mx<Mx; ++mx)
416      for (int mz=0; mz<Mz; ++mz) {
417        for (int my=0; my<My; ++my)
418          g.set(my, f.cmplx(mx,my,mz,i));
419        diff2(g,gyy);
420        for (int my=0; my<My; ++my)
421          laplf.cmplx(mx,my,mz,i) += gyy[my];
422      }
423  }
424  f.makeState(sxz,sy);
425 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.10.4.21 FlowField lapl (const FlowField & *f*)

References lapl().

```
850 {FlowField g;  lapl(f,g); return g;}
```

Here is the call graph for this function:

1.10.4.22 void linearizedNL (const FlowField & *u*, const ChebyCoeff & *U*, FlowField & *f*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), FlowField::kx(), FlowField::kxmax(), FlowField::Lx(), FlowField::Lz(), ChebyCoeff::makePhysical(), FlowField::makeSpectral(), ChebyCoeff::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Mx(), FlowField::Mz(), ChebyCoeff::N(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

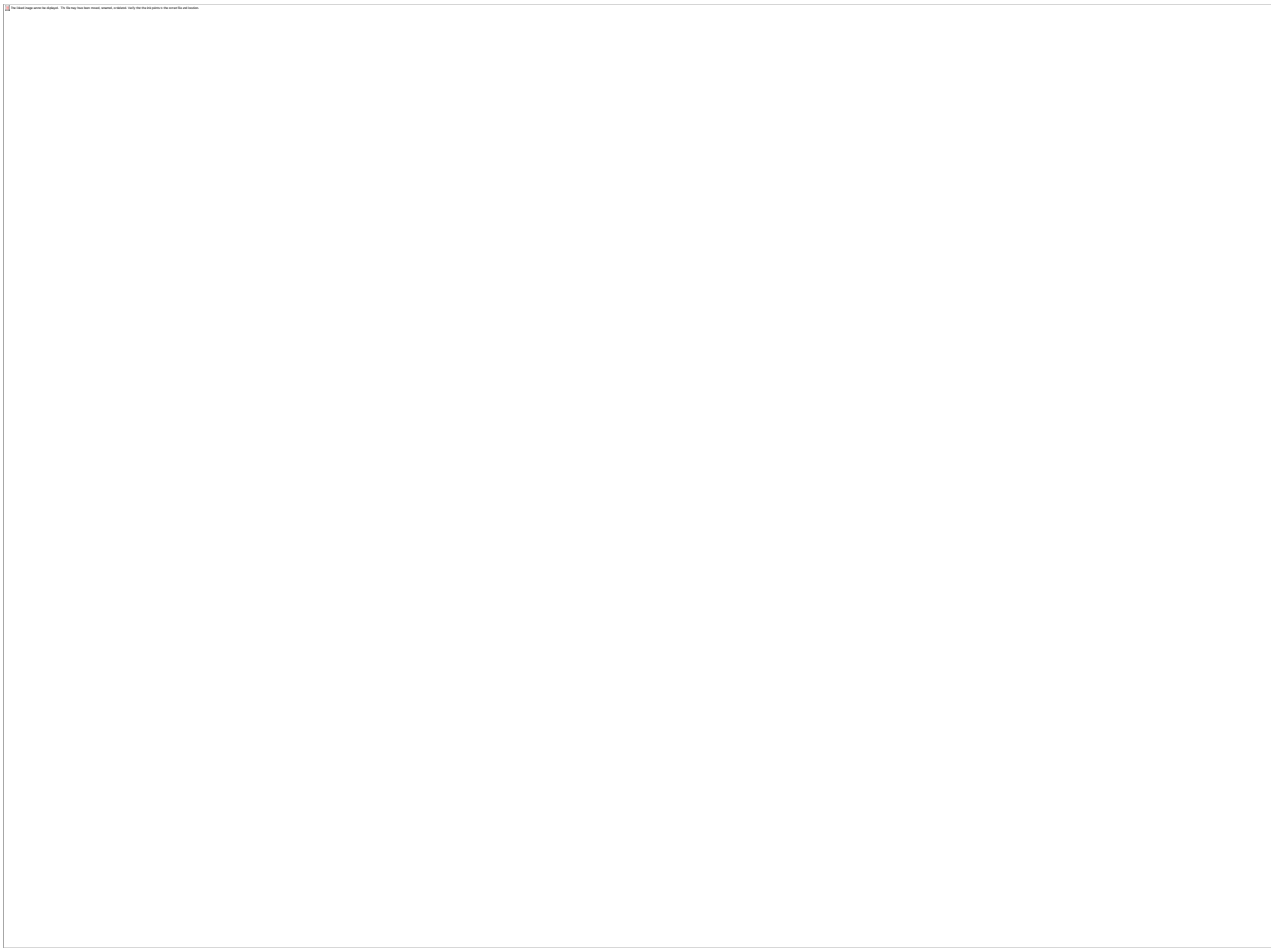
Referenced by navierstokesNL().

```
1530                                                    {
1531   FlowField& u = (FlowField&) u_;
1532   ChebyCoeff& U = (ChebyCoeff&) U_;
1533
1534   assert(u.Nd() == 3);
1535   assert(U.N() == u.Ny());
1536   assert(U.a() == u.a() && U.b() == u.b());
1537
1538   fieldstate uxzstate = u.xzstate();
1539   fieldstate uystate  = u.ystate();
1540   fieldstate Ustate   = U.state();
1541
1542   if (!u.geomCongruent(f) || f.Nd() != 3)
1543     f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
1544   else
1545     f.setToZero();
1546   f.setState(Spectral, Physical);
1547
1548   ChebyTransform trans(U.N());
1549   U.makeSpectral(trans);
1550   ChebyCoeff Uy;
1551   diff(U,Uy);
1552   U.makePhysical(trans);
1553   Uy.makePhysical(trans);
1554
1555   Complex cu;
1556   Complex cv;
1557   Complex cw;
1558
1559   int Mx = u.Mx();
1560   int Ny = u.Ny();
1561   int Mz = u.Mz();
```

```
1562   Real Lx = u.Lx();
1563   int kxmax = u.kxmax();
1564
1565   u.makeState(Spectral, Physical);
1566   for (int ny=0; ny<Ny; ++ny)
1567     for (int mx=0; mx<Mx; ++mx) {
1568       int kx = u.kx(mx);
1569       Complex d_dx(0.0, 2*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
1570       for (int mz=0; mz<Mz; ++mz) {
1571         cu = u.cmplx(mx,ny,mz, 0);
1572         cv = u.cmplx(mx,ny,mz, 1);
1573         cw = u.cmplx(mx,ny,mz, 2);
1574
1575         f.cmplx(mx,ny,mz, 0) = U[ny]*d_dx*cu + cv*Uy[ny];
1576         f.cmplx(mx,ny,mz, 1) = U[ny]*d_dx*cv;
1577         f.cmplx(mx,ny,mz, 2) = U[ny]*d_dx*cw;
1578       }
1579     }
1580
1581   f.makeSpectral();
1582   u.makeState(uxzstate,uystate);
1583   U.makeState(Ustate);
1584   return;
1585 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.10.4.23 void matlabSave (FlowField & *f*, int *i*, std::string & *filename*)

1.10.4.24 void navierstokesNL (const FlowField & *u*, const ChebyCoeff & *U*, FlowField & *f*, FlowField & *tmp*, NonlinearMethod & *method*)

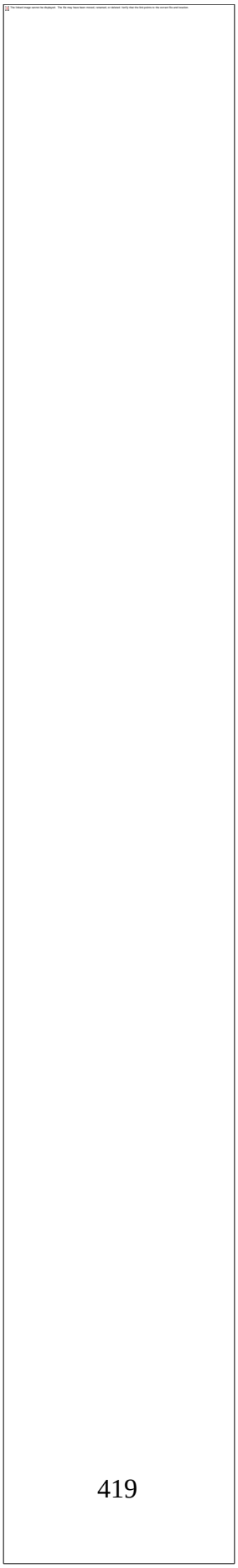
References Alternating, Alternating_, cferror(), Convection, convectionNL(), CPU, Divergence, divergenceNL(), cmndline::GetInstance(), cmndline::getRun(), GPU, Linearized, linearizedNL(), Rotational, rotationalNL(), SkewSymmetric, skewsymmetricNL(), skewsymmetricNL_GPU(), skewsymmetricNL_THREAD(), and THREAD.

Referenced by PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), MultistepDNS::MultistepDNS(),

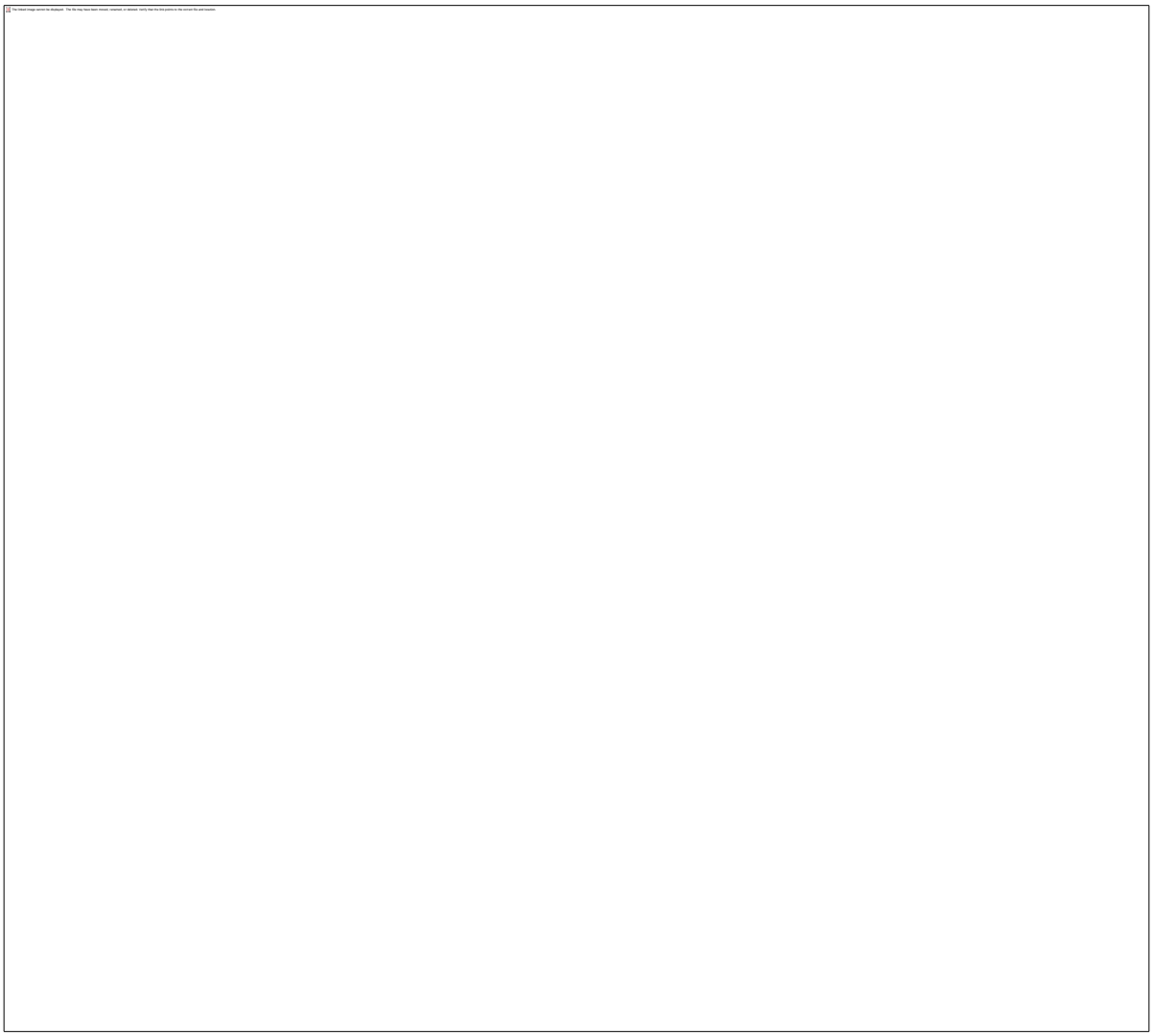
PPEDNS::PPEDNS(), PPEDNS::push(), CNABstyleDNS::push(), MultistepDNS::push(), PressureSolver::solve(), and PressureSolver::verify().

```
873                                         {
874
875 //IG
876   cmndline *cmd = cmndline::GetInstance();
877   //u.print();
878   //cmd->show();
879   //cout << "Run=" << cmd->getRun() << endl;
880   HowToRun howToRun = cmd->getRun();
881 //IG-e
882
883
884   switch (method) {
885   case Rotational:
886     rotationalNL(u,U,f,tmp);
887     break;
888   case Convection:
889     convectionNL(u,U,f,tmp);
890     break;
891   case SkewSymmetric:
892     if(howToRun == CPU) skewsymmetricNL(u,U,f,tmp);
893     if(howToRun == GPU) skewsymmetricNL_GPU(u,U,f,tmp);
894     if(howToRun == THREAD) skewsymmetricNL_THREAD(u,U,f,tmp);
895     break;
896   case Divergence:
897     divergenceNL(u,U,f,tmp);
898     break;
899   case Alternating:
900     divergenceNL(u,U,f,tmp);
901     method = Alternating_;
902     break;
903   case Alternating_:
904     convectionNL(u,U,f,tmp);
905     method = Alternating;
906     break;
907   case Linearized:
908     linearizedNL(u,U,f); break;
909   default:
910     cferror("navierstokesNL(method, u,U,f,tmp,parity) : unknown method");
911   }
912 }
```

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.

Here is the caller graph for this function:



1.10.4.25 void norm (const FlowField & *f*, FlowField & *norm_f*)

References FlowField::a(), FlowField::b(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), square(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
457                                     {
458   FlowField& f = (FlowField&) f_;
459   fieldstate sxz = f.xzstate();
460   fieldstate sy  = f.ystate();
```

```
461   f.makePhysical();
462
463   if (!f.geomCongruent(normf) || normf.vectorDim() != 1)
464     normf.resize(f.Nx(), f.Ny(), f.Nz(), 1, f.Lx(), f.Lz(), f.a(), f.b());
465   else
466     normf.setToZero();
467   normf.setState(Physical, Physical);
468
469   // compute gradf(nx,ny,nz,i) = df(nx,ny,nz,0)/dx_i for scalar-valued f
470   int Nx = f.Nx();
471   int Ny = f.Ny();
472   int Nz = f.Nz();
473   int Nd = f.Nd();
474
475   for (int ny=0; ny<Ny; ++ny)
476     for (int nx=0; nx<Nx; ++nx)
477       for (int nz=0; nz<Nz; ++nz) {
478         Real nrm2 = 0.0;
479         for (int i=0; i<Nd; ++i)
480           nrm2 += square(f(nx,ny,nz,i));
481         normf(nx,ny,nz,0) = sqrt(nrm2);
482       }
483
484   normf.makeSpectral();
485   f.makeState(sxz, sy);
486   return;
487 }
```

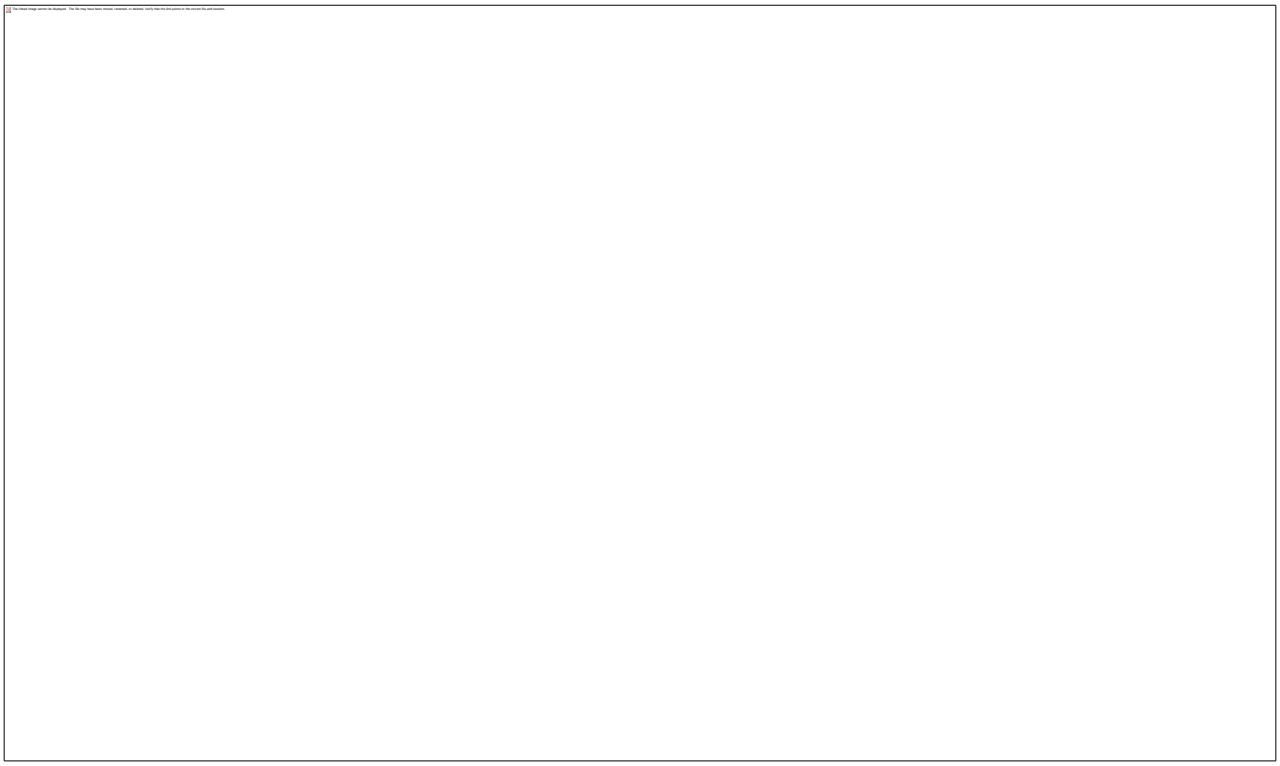
Here is the call graph for this function:

1.10.4.26 FlowField norm (const FlowField & *f*)

References norm().

```
852 {FlowField g;  norm(f,g); return g;}
```

Here is the call graph for this function:



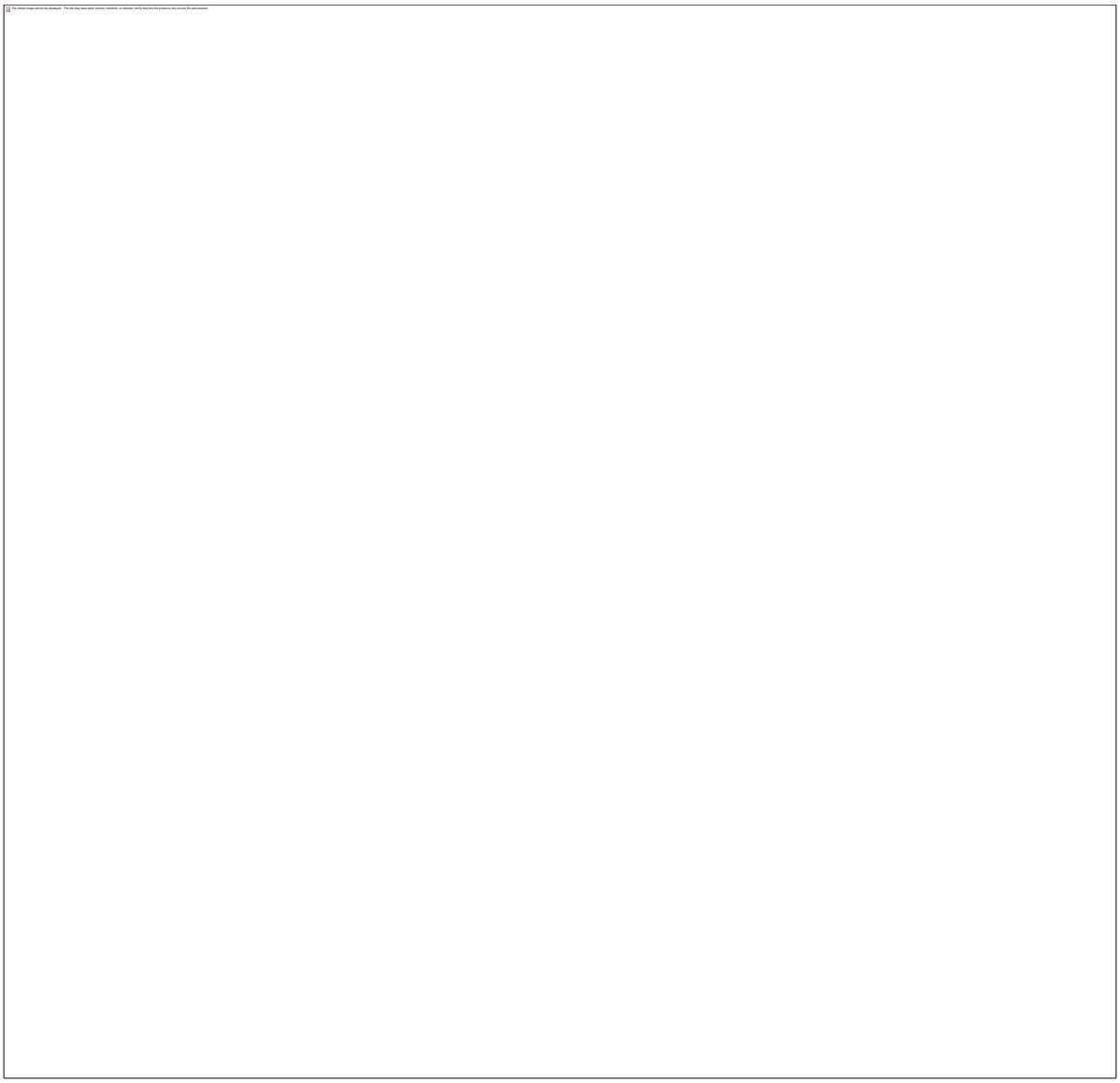
1.10.4.27 void norm2 (const FlowField & *f*, FlowField & *norm2_f*)

References FlowField::a(), FlowField::b(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, square(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
427                                     {
428   FlowField& f = (FlowField&) f_;
429   fieldstate sxz = f.xzstate();
430   fieldstate sy  = f.ystate();
431   f.makeState(Spectral, Spectral);
432
```

```
433   if (!f.geomCongruent(norm2f) || norm2f.vectorDim() != 1)
434     norm2f.resize(f.Nx(), f.Ny(), f.Nz(), 1, f.Lx(), f.Lz(), f.a(), f.b());
435   else
436     norm2f.setToZero();
437   norm2f.setState(Physical, Physical);
438
439   f.makePhysical();
440
441   int Nx = f.Nx();
442   int Ny = f.Ny();
443   int Nz = f.Nz();
444   int Nd = f.Nd();
445
446   for (int i=0; i<Nd; ++i)
447     for (int ny=0; ny<Ny; ++ny)
448       for (int nx=0; nx<Nx; ++nx)
449         for (int nz=0; nz<Nz; ++nz)
450           norm2f(nx,ny,nz,0) += square(f(nx,ny,nz,i));
451
452   f.makeState(sxz,sy);
453   norm2f.makeSpectral();
454   return;
455 }
```

Here is the call graph for this function:



1.10.4.28 FlowField norm2 (const FlowField & *f*)

References norm2().

```
853 {FlowField g; norm2(f,g); return g;}
```

Here is the call graph for this function:

1.10.4.29 std::ostream& operator<< (std::ostream & *os*, NonlinearMethod *nonl*)

References Alternating, Alternating_, Convection, Divergence, Linearized, Rotational, and SkewSymmetric.

```
1656                                                {
1657   string s;
1658   switch(nonl) {
1659   case Rotational: s="Rotational"; break;
1660   case Convection: s="Convection"; break;
1661   case Divergence: s="Divergence"; break;
1662   case SkewSymmetric: s="SkewSymmetric"; break;
1663   case Alternating: s="Alternating"; break;
1664   case Alternating_: s="Alternating_"; break;
1665   case Linearized: s="Linearized"; break;
1666   default: s = "Invalid NonlinearMethod: please submit bug report";
1667   }
1668   os << s;
1669   return os;
1670 }
```

1.10.4.30 void outer (const FlowField & *f*, const FlowField & *g*, FlowField & *fg*)

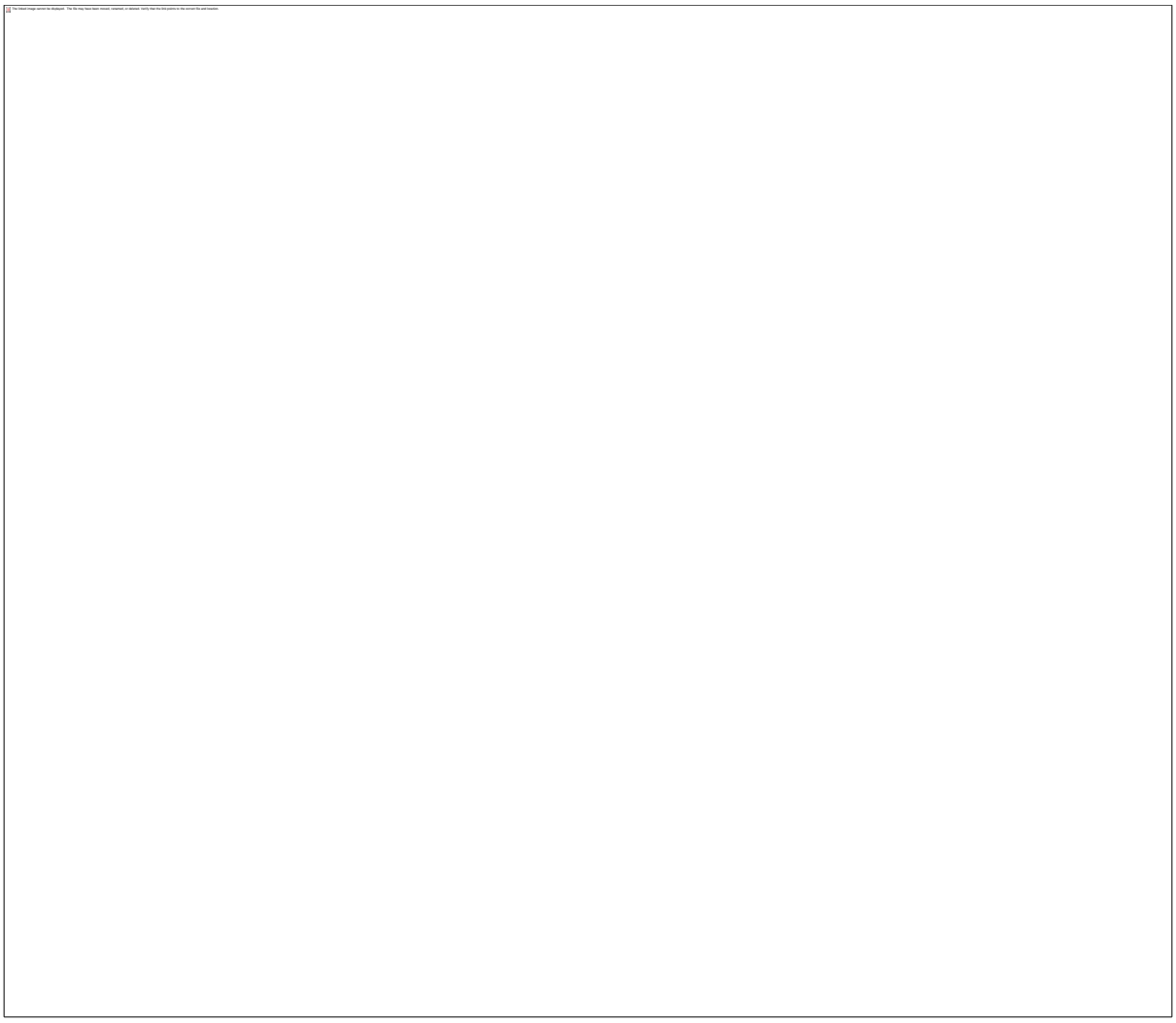
References FlowField::a(), FlowField::b(), FlowField::geomCongruent(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), FlowField::makeSpectral(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), FlowField::xzstate(), and FlowField::ystate().

Referenced by divergenceNL(), and outer().

```
552                                                {
553   FlowField& f = (FlowField&) f_;
554   FlowField& g = (FlowField&) g_;
555   fieldstate fxz = f.xzstate();
556   fieldstate fy  = f.ystate();
557   fieldstate gxz = g.xzstate();
```

```
558   fieldstate gy  = g.ystate();
559
560   assert(f.geomCongruent(g));
561   f.makePhysical();
562   g.makePhysical();
563
564   if (!f.geomCongruent(fg) || fg.Nd() != f.Nd()*g.Nd())
565     fg.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd()*g.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
566   else
567     fg.setToZero();
568   fg.setState(Physical, Physical);
569
570   int Nx = f.Nx();
571   int Ny = f.Ny();
572   int Nz = f.Nz();
573   int fd = f.Nd();
574   int gd = g.Nd();
575
576   for (int i=0; i<fd; ++i)
577     for (int j=0; j<gd; ++j) {
578       int ij = i*gd + j;   // generalized form of i3j == i*3 + j
579       for (int ny=0; ny<Ny; ++ny)
580         for (int nx=0; nx<Nx; ++nx)
581           for (int nz=0; nz<Nz; ++nz)
582             fg(nx,ny,nz,ij) = f(nx,ny,nz,i)*g(nx,ny,nz,j);
583     }
584
585   f.makeState(fxz, fy);
586   g.makeState(gxz, gy);
587   fg.makeSpectral();
588   return;
589 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.10.4.31 FlowField outer (const FlowField & *f*, const FlowField & *g*)

References outer().

```
856                                                 {
857   FlowField fg; outer(f,g,fg); return fg;
858 }
```

Here is the call graph for this function:

1.10.4.32 void rotationalNL (const FlowField & *u*, const ChebyCoeff & *U*, FlowField & *f*, FlowField & *tmp*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::cmplx(), curl(), diff(), FlowField::geomCongruent(), FlowField::kx(), FlowField::kxmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), ChebyCoeff::makePhysical(), ChebyCoeff::makeSpectral(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), ChebyCoeff::N(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

Referenced by navierstokesNL().

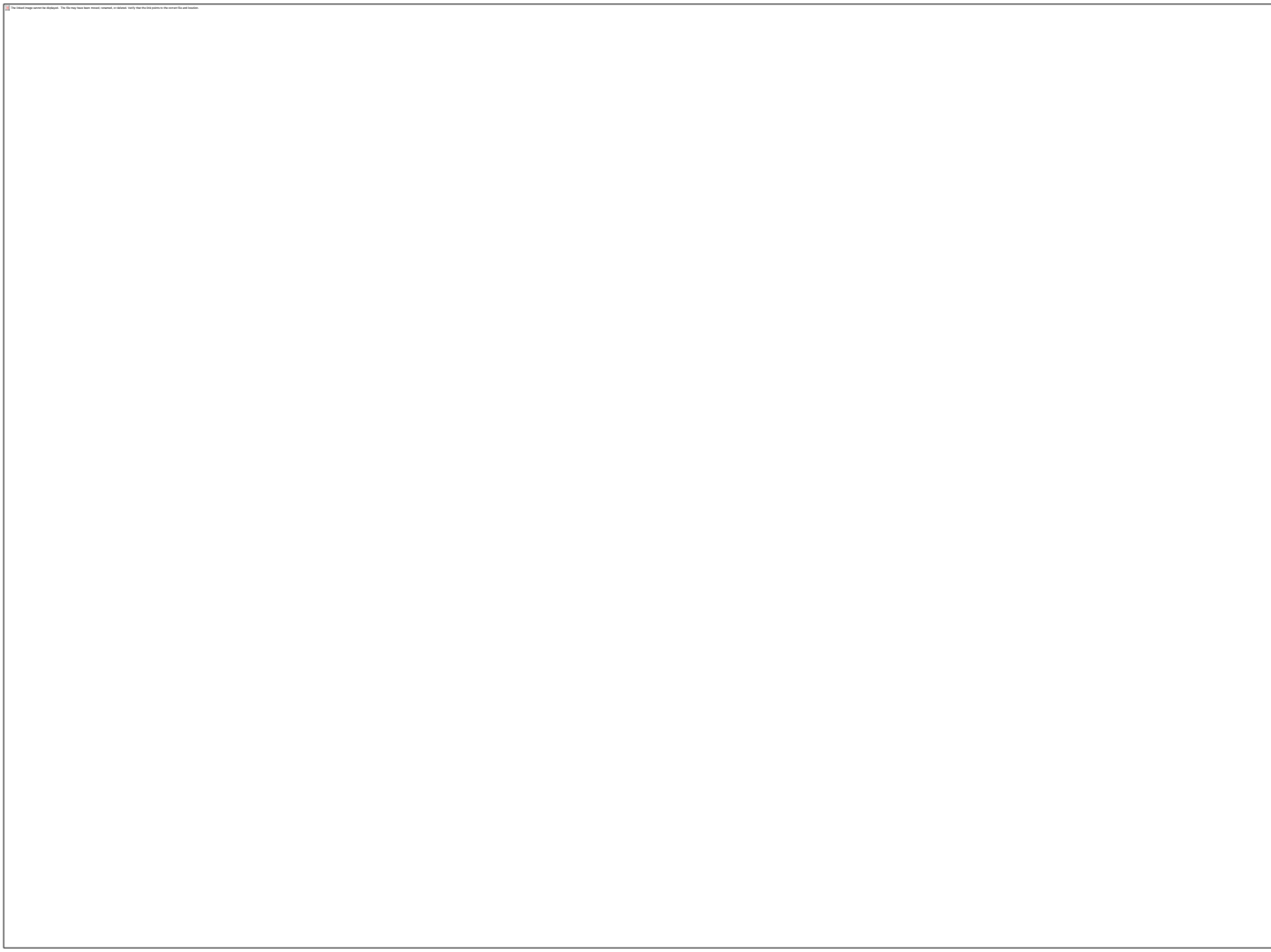
```
925                             {
926
927   FlowField& u = (FlowField&) u_;
928   ChebyCoeff& U = (ChebyCoeff&) U_;
929   FlowField& vort = tmp;
930
931   assert(u.Nd() == 3);
932   assert(U.numModes() == u.Ny());
933   assert(U.a() == u.a() && U.b() == u.b());
934   fieldstate uxzstate = u.xzstate();
935   fieldstate uystate  = u.ystate();
936   fieldstate Ustate   = U.state();
937
938   if (!u.geomCongruent(f) || f.Nd() != 3)
939     f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
940   else
941     f.setToZero();
942   f.setState(Physical, Physical);
943
944   if (!u.geomCongruent(vort) || vort.Nd() < 3)
945     vort.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
946
947   ChebyTransform trans(U.N());
948   u.makeSpectral();
949   U.makeSpectral(trans);
950   ChebyCoeff Uy = diff(U);
951
952   curl(u, vort);
953
954   u.makeState(Spectral, Physical);;
955   vort.makeState(Spectral, Physical);
```

```
956  f.setState(Spectral, Physical);
957  U.makePhysical(trans);
958  Uy.makePhysical(trans);
959
960  Complex uc;
961  Complex vc;
962  Complex wc;
963
964  int Mx = u.Mx();
965  int My = u.My();
966  int Mz = u.Mz();
967  int kxmax = f.kxmax();
968
969  for (int ny=0; ny<My; ++ny) {
970    Real rU  = U[ny];
971    Real rUy = Uy[ny];
972    for (int mx=0; mx<Mx; ++mx) {
973      int kx = u.kx(mx);
974      Complex d_dx(0.0, 2*pi*kx/u.Lx()*zero_last_mode(kx,kxmax,1));
975      for (int mz=0; mz<Mz; ++mz) {
976        uc = u.cmplx(mx,ny,mz, 0);
977        vc = u.cmplx(mx,ny,mz, 1);
978        wc = u.cmplx(mx,ny,mz, 2);
979
980        f.cmplx(mx,ny,mz, 0) = rU*d_dx*uc + vc*rUy;
981        f.cmplx(mx,ny,mz, 1) = rU*d_dx*vc;
982        f.cmplx(mx,ny,mz, 2) = rU*d_dx*wc;
983      }
984    }
985  }
986
987  vort.makePhysical();
988  u.makePhysical();
989  f.makePhysical();
990
991  int Nx = u.Nx();
992  int Ny = u.Ny();
993  int Nz = u.Nz();
994
995  Real omega_x;
996  Real omega_y;
997  Real omega_z;
998  Real ru;
999  Real rv;
1000   Real rw;
1001
```

```
1002   // Loop should be efficient if we get 9 cache lines.
1003   // Add in omega x u so that
1004   // f = omega x U + Omega x u + omega x u
1005   for (int ny=0; ny<Ny; ++ny) {
1006     //Real Uy = Ubasey[ny];
1007     for (int nx=0; nx<Nx; ++nx) {
1008       for (int nz=0; nz<Nz; ++nz) {
1009         ru = u(nx,ny,nz, 0);
1010         rv = u(nx,ny,nz, 1);
1011         rw = u(nx,ny,nz, 2);
1012         omega_x = vort(nx,ny,nz, 0);
1013         omega_y = vort(nx,ny,nz, 1);
1014         omega_z = vort(nx,ny,nz, 2);
1015
1016         f(nx,ny,nz, 0) += omega_y*rw - omega_z*rv;
1017         f(nx,ny,nz, 1) += omega_z*ru - omega_x*rw;
1018         f(nx,ny,nz, 2) += omega_x*rv - omega_y*ru;
1019       }
1020     }
1021   }
1022   f.makeSpectral();
1023
1024   // Possible extra transforms if entry state was Physical
1025   u.makeState(uxzstate, uystate);
1026   U.makeState(Ustate, trans);
1027
1028   return;
1029 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.10.4.33 void skewsymmetricNL (const FlowField & *u*, const ChebyCoeff & *U*, FlowField & *f*, FlowField & *tmp*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), grad(), i3j(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), ChebyCoeff::makeSpectral(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

Referenced by navierstokesNL().

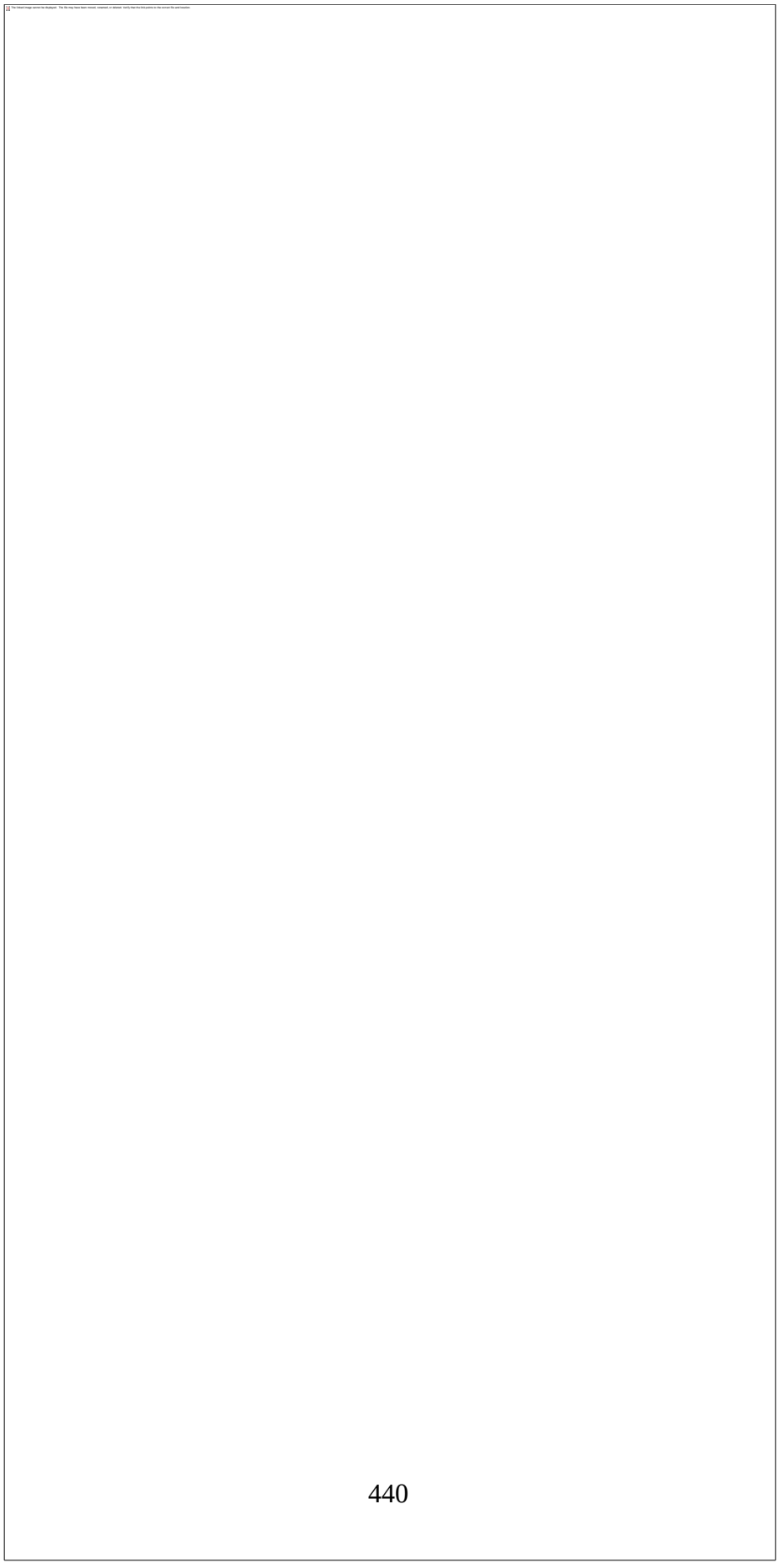
```
1130                                                                {
1131
1132  FlowField& u = (FlowField&) u_;
1133  ChebyCoeff& U = (ChebyCoeff&) U_;
1134
1135  assert(u.Nd() == 3);
1136  assert(U.numModes() == u.Ny());
1137  assert(U.a() == u.a() && U.b() == u.b());
1138
1139  fieldstate uxzstate = u.xzstate();
1140  fieldstate uystate  = u.ystate();
1141  fieldstate Ustate   = U.state();
1142
1143  if (!u.geomCongruent(f) || f.Nd() != 3)
1144    f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
1145  else
1146    f.setToZero();
1147  f.setState(Physical, Physical);
1148
1149  if (!u.geomCongruent(tmp) || tmp.Nd() < 9)
1150    tmp.resize(u.Nx(), u.Ny(), u.Nz(), 9, u.Lx(), u.Lz(), u.a(), u.b());
1151  else
1152    tmp.setToZero();
1153
1154  // Possible extra transforms if entry states are Physical
1155  u.makeSpectral();
1156  U.makeSpectral();
1157  u += U;
1158
1159  //
==========================================================================
==
1160  // Compute 1/2 u dotgrad u. (where u is set to former u + U)
1161
1162  FlowField& grad_u = tmp;
1163  grad(u, grad_u);
1164
1165  grad_u.makePhysical();
1166  u.makePhysical();
1167
1168  int Nx = u.Nx();
1169  int Ny = u.Ny();
1170  int Nz = u.Nz();
1171
1172
```

```
1173  // Accumulate 1/2 u_j du_i/dx_j in f_i
1174  clock_t t = clock();
1175  for (int i=0; i<3; ++i)
1176    for (int j=0; j<3; ++j) {
1177      int ij = i3j(i,j);
1178      for (int ny=0; ny<Ny; ++ny)
1179        for (int nx=0; nx<Nx; ++nx)
1180          for (int nz=0; nz<Nz; ++nz)
1181            f(nx,ny,nz,i) += 0.5*u(nx,ny,nz,j)*grad_u(nx,ny,nz,ij);
1182    }
1183
1184   t = clock() -t;
1185  printf("IG: first task cpu  time = %f  \n",((float)t)/CLOCKS_PER_SEC);
1186  //exit(0);
1187
1188
1189  //
=================================================================
1190  // II. Add grad dot (u u) to f. Spell out loops because div(uu, f)
1191  // would overwrite results already in f (and changing order of div
1192  // and convec calculations would require an extra transform)
1193
1194  FlowField& uu = tmp;
1195
1196  //outer(u,u,uu);
1197   t = clock();
1198
1199  for (int ny=0; ny<Ny; ++ny) {
1200    for (int nx=0; nx<Nx; ++nx)
1201      for (int nz=0; nz<Nz; ++nz) {
1202        Real u0 = u(nx,ny,nz,0);
1203        Real u1 = u(nx,ny,nz,1);
1204        Real u2 = u(nx,ny,nz,2);
1205        uu(nx,ny,nz,0) = u0*u0;
1206        uu(nx,ny,nz,1) = tmp(nx,ny,nz,3) = u0*u1;
1207        uu(nx,ny,nz,2) = tmp(nx,ny,nz,6) = u0*u2;
1208        uu(nx,ny,nz,4) = u1*u1;
1209        uu(nx,ny,nz,5) = tmp(nx,ny,nz,7) = u1*u2;
1210        uu(nx,ny,nz,8) = u2*u2;
1211      }
1212  }
1213
1214   t = clock() -t;
1215  printf("IG: second task cpu  time = %f  \n",((float)t)/CLOCKS_PER_SEC);
1216
1217  uu.makeSpectral();
```

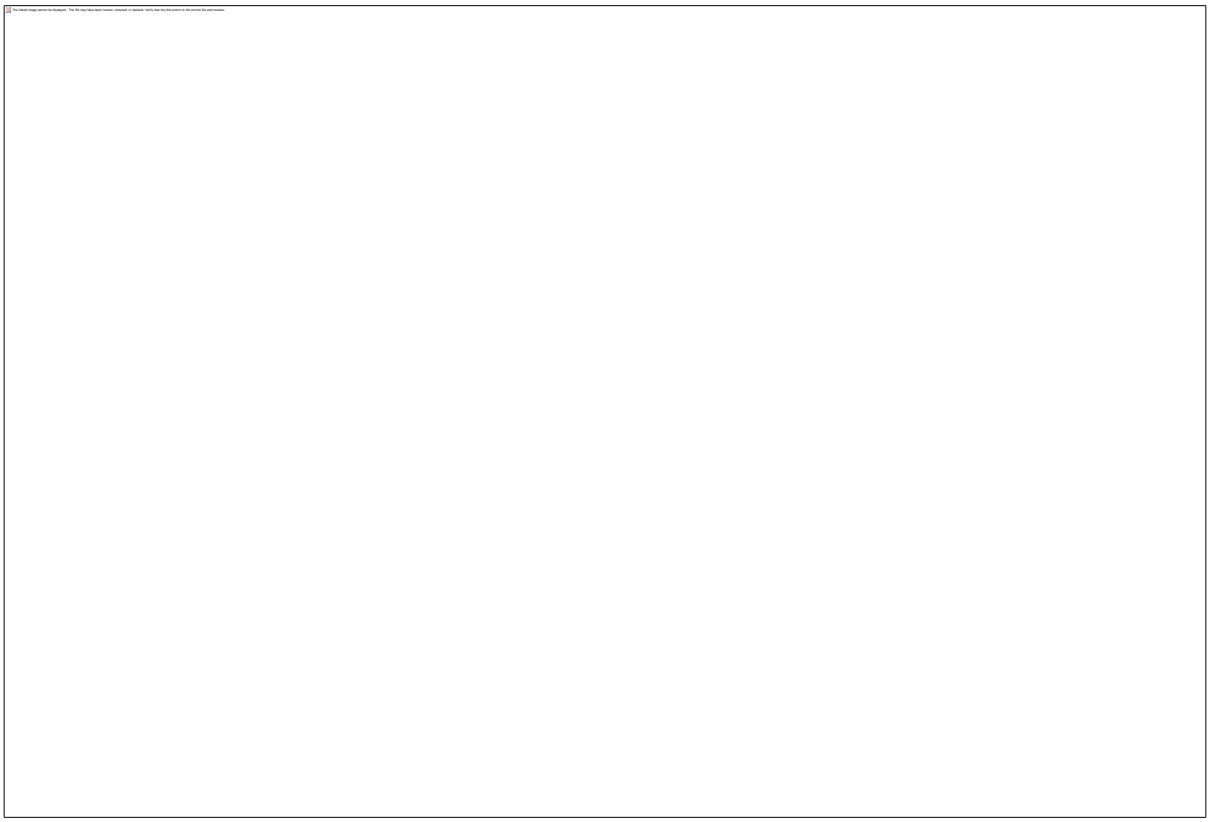
```
1218  f.makeSpectral();
1219
1220  int Mx = u.Mx();
1221  int My = u.My();
1222  int Mz = u.Mz();
1223  int kxmax = u.kxmax();
1224  int kzmax = u.kzmax();
1225  Real Lx = u.Lx();
1226  Real Lz = u.Lz();
1227
1228  ComplexChebyCoeff tmpProfile(My, u.a(), u.b(), Spectral);
1229  ComplexChebyCoeff tmpProfile_y(My, u.a(), u.b(), Spectral);
1230
1231  // Now set f_i += d/dx_j (u_i u_j)
1232   t = clock();
1233
1234  for (int i=0; i<3; ++i) {
1235    int i0 = i3j(i,0);
1236    int i1 = i3j(i,1);
1237    int i2 = i3j(i,2);
1238
1239    // Add in du_i/dx and du_i/dz, that is, d/dx_j (u_i u_j) for j=0,2
1240    for (int my=0; my<My; ++my)
1241      for (int mx=0; mx<Mx; ++mx) {
1242        int kx = u.kx(mx);
1243        Complex d_dx(0.0, 2*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
1244        for (int mz=0; mz<Mz; ++mz) {
1245          int kz = u.kz(mz);
1246          Complex d_dz(0.0, 2*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
1247          f.cmplx(mx,my,mz,i)
1248            += 0.5*(d_dx*uu.cmplx(mx,my,mz,i0)+d_dz*uu.cmplx(mx,my,mz,i2));
1249        }
1250      }
1251    // Add in du_i/dy, that is d/dx_j (u_i u_j) for j=1
1252    for (int mx=0; mx<Mx; ++mx)
1253      for (int mz=0; mz<Mz; ++mz) {
1254        for (int my=0; my<My; ++my)
1255          tmpProfile.set(my, uu.cmplx(mx,my,mz,i1));
1256        diff(tmpProfile, tmpProfile_y);
1257        for (int my=0; my<My; ++my)
1258          f.cmplx(mx,my,mz,i) += 0.5*tmpProfile_y[my]; // j=1
1259      }
1260  }
1261
1262   t = clock() -t;
1263  printf("IG: thirt  task cpu  time = %f  \n",((float)t)/CLOCKS_PER_SEC);
```

```
1264
1265   u.makeSpectral();
1266   U.makeSpectral();
1267   u -= U;
1268
1269   // Possible extra transforms if entry states are Physical
1270   u.makeState(uxzstate, uystate);
1271   U.makeState(Ustate);
1272
1273   //exit(0);
1274   return;
1275 }
```

Here is the call graph for this function:


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.

Here is the caller graph for this function:



1.10.4.34 void skewsymmetricNL_GPU (const FlowField & *u*, const ChebyCoeff & *U*, FlowField & *f*, FlowField & *tmp*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::cmplx(), diff(), FlowField::geomCongruent(), grad(), i3j(), FlowField::kx(), FlowField::kxmax(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), ChebyCoeff::makeSpectral(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, pi, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

Referenced by navierstokesNL().

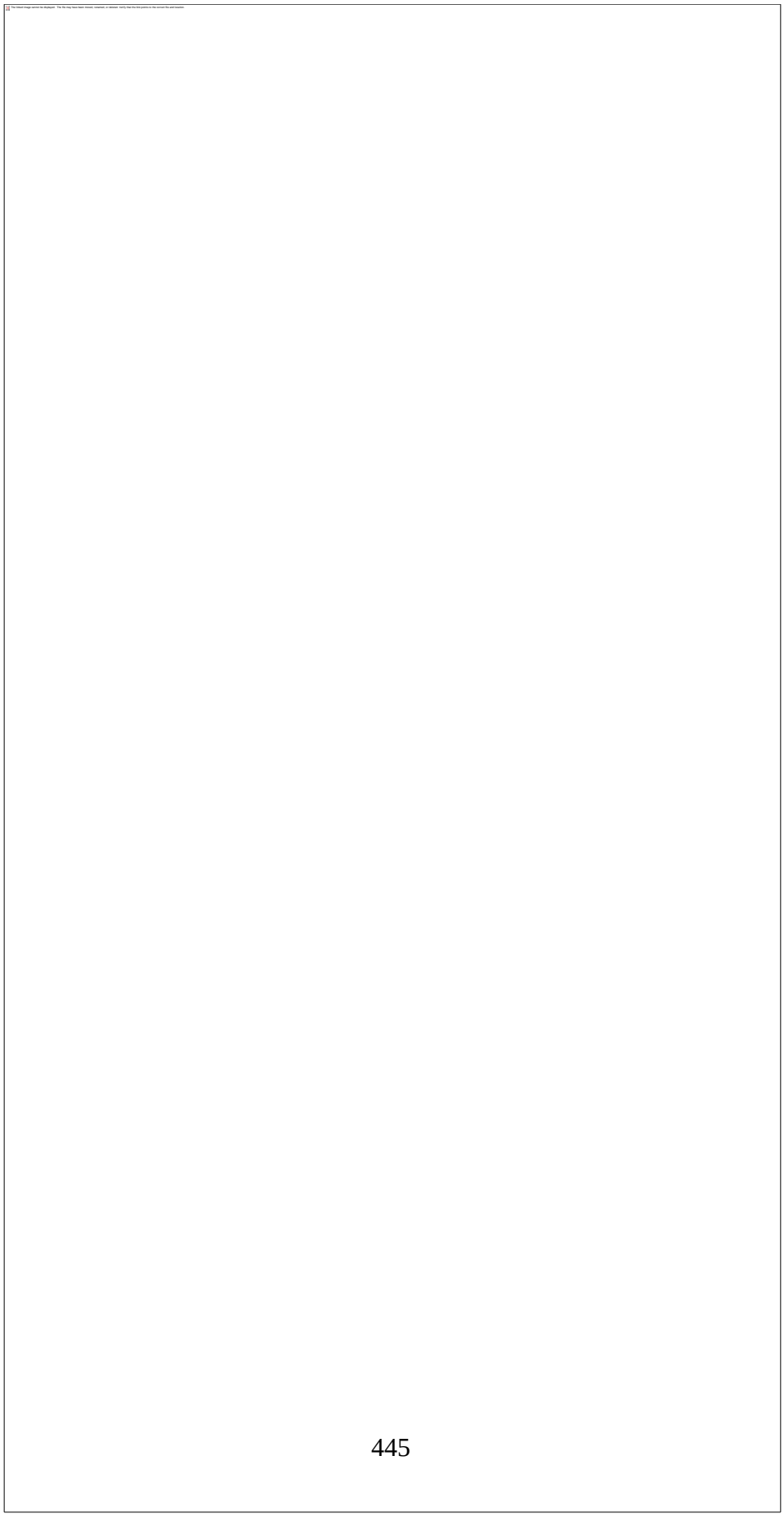
```
1277                                                                    {
1278
1279   FlowField& u = (FlowField&) u_;
1280   ChebyCoeff& U = (ChebyCoeff&) U_;
1281
1282   assert(u.Nd() == 3);
```

```
1283  assert(U.numModes() == u.Ny());
1284  assert(U.a() == u.a() && U.b() == u.b());
1285
1286  fieldstate uxzstate = u.xzstate();
1287  fieldstate uystate  = u.ystate();
1288  fieldstate Ustate   = U.state();
1289
1290  if (!u.geomCongruent(f) || f.Nd() != 3)
1291    f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
1292  else
1293    f.setToZero();
1294  f.setState(Physical, Physical);
1295
1296  if (!u.geomCongruent(tmp) || tmp.Nd() < 9)
1297    tmp.resize(u.Nx(), u.Ny(), u.Nz(), 9, u.Lx(), u.Lz(), u.a(), u.b());
1298  else
1299    tmp.setToZero();
1300
1301  // Possible extra transforms if entry states are Physical
1302  u.makeSpectral();
1303  U.makeSpectral();
1304  u += U;
1305
1306  //
==========================================================================
==
1307  // Compute 1/2 u dotgrad u. (where u is set to former u + U)
1308
1309  FlowField& grad_u = tmp;
1310  grad(u, grad_u);
1311
1312  grad_u.makePhysical();
1313  u.makePhysical();
1314
1315  int Nx = u.Nx();
1316  int Ny = u.Ny();
1317  int Nz = u.Nz();
1318
1319  // Accumulate 1/2 u_j du_i/dx_j in f_i
1320  for (int i=0; i<3; ++i)
1321    for (int j=0; j<3; ++j) {
1322      int ij = i3j(i,j);
1323      for (int ny=0; ny<Ny; ++ny)
1324        for (int nx=0; nx<Nx; ++nx)
1325          for (int nz=0; nz<Nz; ++nz)
1326            f(nx,ny,nz,i) += 0.5*u(nx,ny,nz,j)*grad_u(nx,ny,nz,ij);
```

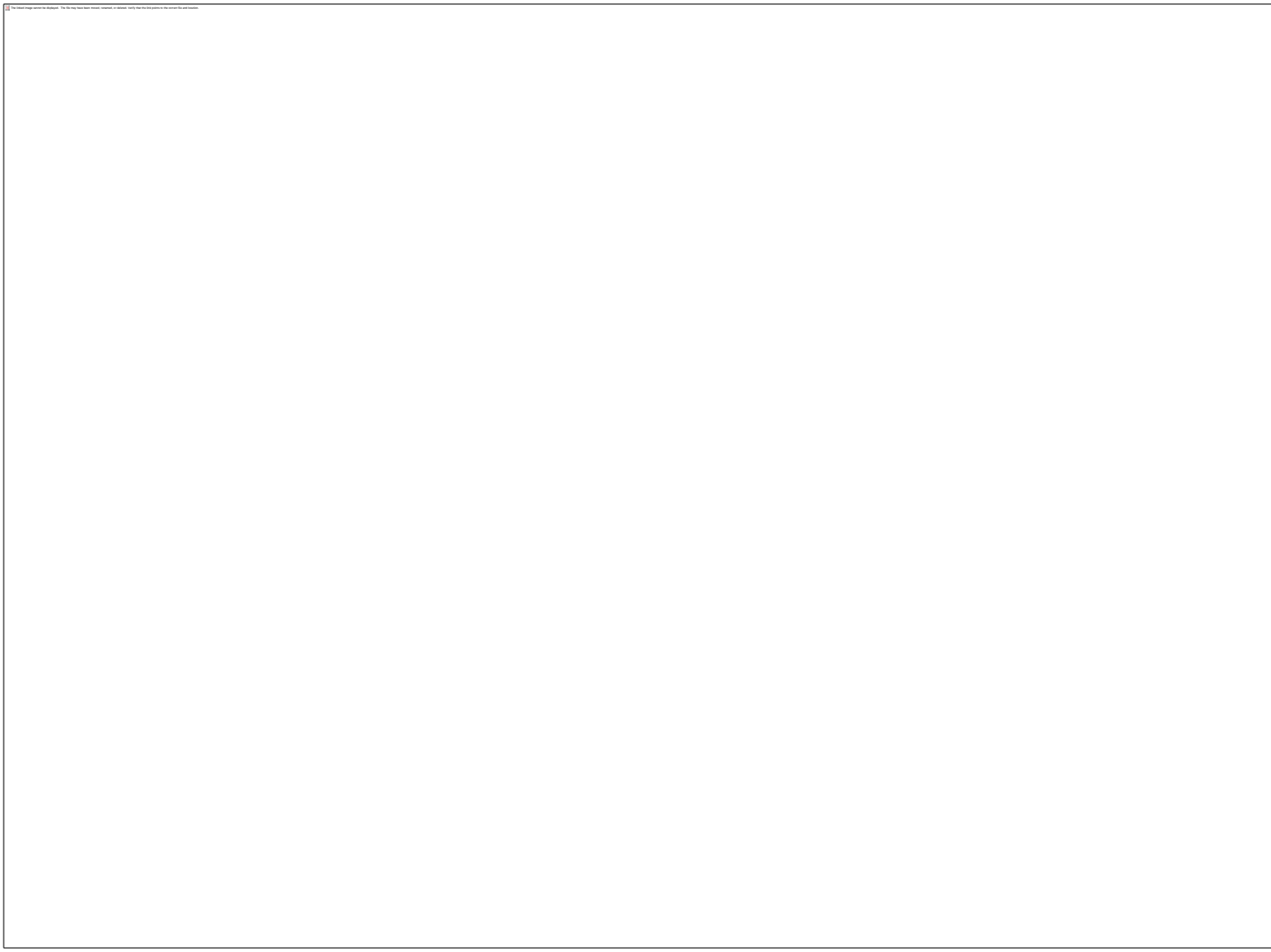
```
1327    }
1328
1329  //
=================================================================
1330  // II. Add grad dot (u u) to f. Spell out loops because div(uu, f)
1331  // would overwrite results already in f (and changing order of div
1332  // and convec calculations would require an extra transform)
1333
1334  FlowField& uu = tmp;
1335
1336  //outer(u,u,uu);
1337
1338  for (int ny=0; ny<Ny; ++ny) {
1339    for (int nx=0; nx<Nx; ++nx)
1340      for (int nz=0; nz<Nz; ++nz) {
1341        Real u0 = u(nx,ny,nz,0);
1342        Real u1 = u(nx,ny,nz,1);
1343        Real u2 = u(nx,ny,nz,2);
1344        uu(nx,ny,nz,0) = u0*u0;
1345        uu(nx,ny,nz,1) = tmp(nx,ny,nz,3) = u0*u1;
1346        uu(nx,ny,nz,2) = tmp(nx,ny,nz,6) = u0*u2;
1347        uu(nx,ny,nz,4) = u1*u1;
1348        uu(nx,ny,nz,5) = tmp(nx,ny,nz,7) = u1*u2;
1349        uu(nx,ny,nz,8) = u2*u2;
1350      }
1351  }
1352
1353  uu.makeSpectral();
1354  f.makeSpectral();
1355
1356  int Mx = u.Mx();
1357  int My = u.My();
1358  int Mz = u.Mz();
1359  int kxmax = u.kxmax();
1360  int kzmax = u.kzmax();
1361  Real Lx = u.Lx();
1362  Real Lz = u.Lz();
1363
1364  ComplexChebyCoeff tmpProfile(My, u.a(), u.b(), Spectral);
1365  ComplexChebyCoeff tmpProfile_y(My, u.a(), u.b(), Spectral);
1366
1367  // Now set f_i += d/dx_j (u_i u_j)
1368  for (int i=0; i<3; ++i) {
1369    int i0 = i3j(i,0);
1370    int i1 = i3j(i,1);
1371    int i2 = i3j(i,2);
```

```
1372
1373    // Add in du_i/dx and du_i/dz, that is, d/dx_j (u_i u_j) for j=0,2
1374    for (int my=0; my<My; ++my)
1375      for (int mx=0; mx<Mx; ++mx) {
1376        int kx = u.kx(mx);
1377        Complex d_dx(0.0, 2*pi*kx/Lx*zero_last_mode(kx,kxmax,1));
1378        for (int mz=0; mz<Mz; ++mz) {
1379          int kz = u.kz(mz);
1380          Complex d_dz(0.0, 2*pi*kz/Lz*zero_last_mode(kz,kzmax,1));
1381          f.cmplx(mx,my,mz,i)
1382            += 0.5*(d_dx*uu.cmplx(mx,my,mz,i0)+d_dz*uu.cmplx(mx,my,mz,i2));
1383        }
1384      }
1385    // Add in du_i/dy, that is d/dx_j (u_i u_j) for j=1
1386    for (int mx=0; mx<Mx; ++mx)
1387      for (int mz=0; mz<Mz; ++mz) {
1388        for (int my=0; my<My; ++my)
1389          tmpProfile.set(my, uu.cmplx(mx,my,mz,i1));
1390        diff(tmpProfile, tmpProfile_y);
1391        for (int my=0; my<My; ++my)
1392          f.cmplx(mx,my,mz,i) += 0.5*tmpProfile_y[my]; // j=1
1393      }
1394  }
1395
1396  u.makeSpectral();
1397  U.makeSpectral();
1398  u -= U;
1399
1400  // Possible extra transforms if entry states are Physical
1401  u.makeState(uxzstate, uystate);
1402  U.makeState(Ustate);
1403
1404  return;
1405 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



1.10.4.35 void skewsymmetricNL_THREAD (const FlowField & *u*, const ChebyCoeff & *U*, FlowField & *f*, FlowField & *tmp*)

References FlowField::a(), ChebyCoeff::a(), FlowField::b(), ChebyCoeff::b(), FlowField::cmplx(), diff(), ThreadPool::executeTask(), FlowField::geomCongruent(), ThreadPool::GetInstance(), grad(), i3j(), FlowField::kxmax(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makePhysical(), ChebyCoeff::makeSpectral(), FlowField::makeSpectral(), ChebyCoeff::makeState(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), ChebyCoeff::numModes(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, ChebyCoeff::state(), FlowField::xzstate(), and FlowField::ystate().

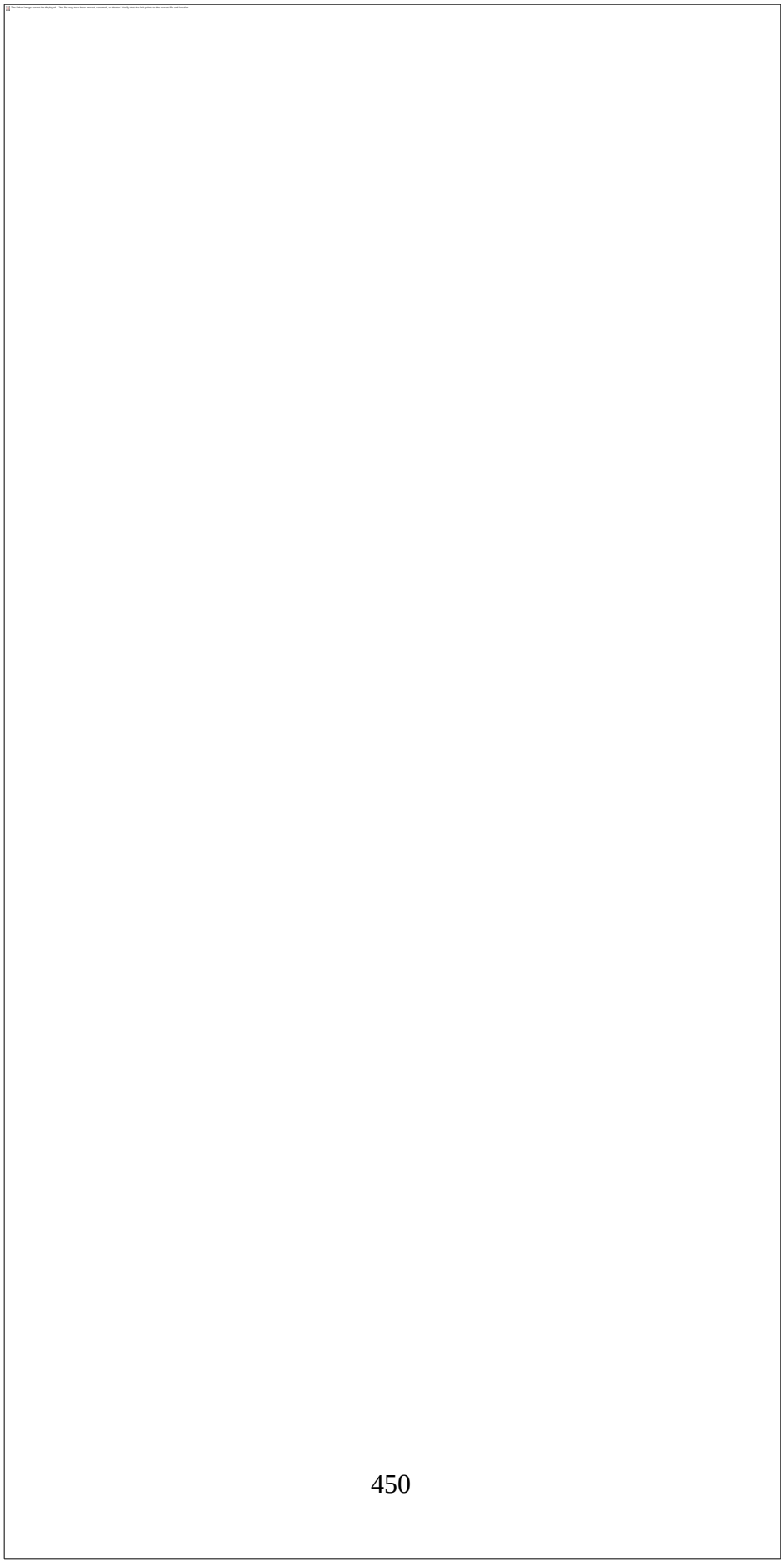
Referenced by navierstokesNL().

```
1408 {
1409   ThreadPool *pool = ThreadPool::GetInstance();
1410
1411   FlowField& u = (FlowField&) u_;
1412   ChebyCoeff& U = (ChebyCoeff&) U_;
1413
1414   assert(u.Nd() == 3);
1415   assert(U.numModes() == u.Ny());
1416   assert(U.a() == u.a() && U.b() == u.b());
1417
1418   fieldstate uxzstate = u.xzstate();
1419   fieldstate uystate  = u.ystate();
1420   fieldstate Ustate   = U.state();
1421
1422   if (!u.geomCongruent(f) || f.Nd() != 3)
1423     f.resize(u.Nx(), u.Ny(), u.Nz(), 3, u.Lx(), u.Lz(), u.a(), u.b());
1424   else
1425     f.setToZero();
1426   f.setState(Physical, Physical);
1427
1428   if (!u.geomCongruent(tmp) || tmp.Nd() < 9)
1429     tmp.resize(u.Nx(), u.Ny(), u.Nz(), 9, u.Lx(), u.Lz(), u.a(), u.b());
1430   else
1431     tmp.setToZero();
1432
1433   // Possible extra transforms if entry states are Physical
1434   u.makeSpectral();
1435   U.makeSpectral();
1436   u += U;
1437
1438   //
==========================================================================
==
1439   // Compute 1/2 u dotgrad u. (where u is set to former u + U)
1440
1441   FlowField& grad_u = tmp;
1442   grad(u, grad_u);
1443
1444   grad_u.makePhysical();
1445   u.makePhysical();
1446
1447
1448
1449
1450
```

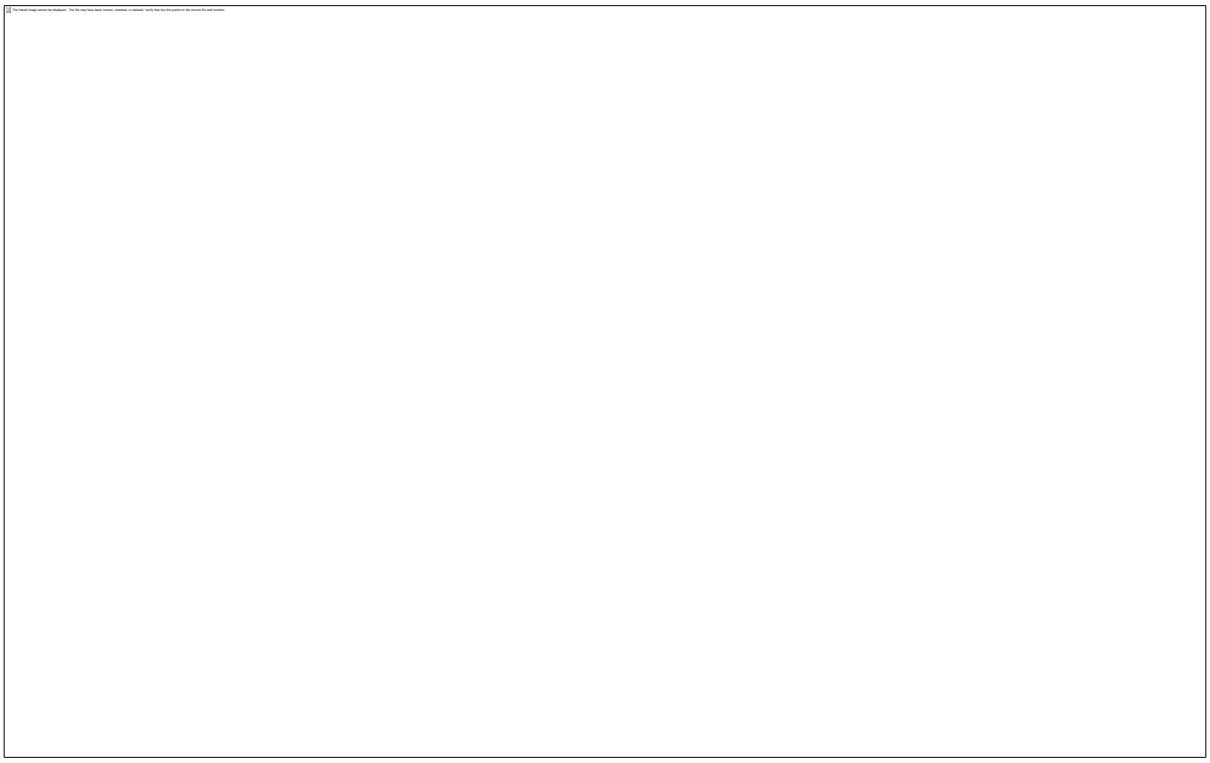
```
1451
1452        //  clock_t tm = clock();
1453        //struct timespec start, finish;
1454        //double elapsed;
1455        //clock_gettime(CLOCK_MONOTONIC, &start);
1456
1457       pool->executeTask( new skewsymmetricNL_task(u, f, grad_u));
1458
1459     //   clock_gettime(CLOCK_MONOTONIC, &finish);
1460     //         elapsed = (finish.tv_sec - start.tv_sec);
1461     //   elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
1462
1463      //  cout << "total time="<<elapsed<<endl;
1464
1465       //   tm = clock() -tm;
1466       //   printf("IG: total time = %f  \n",((float)tm)/CLOCKS_PER_SEC);
1467
1468
1469  //
==========================================================================
1470  // II. Add grad dot (u u) to f. Spell out loops because div(uu, f)
1471  // would overwrite results already in f (and changing order of div
1472  // and convec calculations would require an extra transform)
1473
1474  FlowField& uu = tmp;
1475
1476  //outer(u,u,uu);
1477
1478
1479       pool->executeTask( new skewsymmetricNL_task2(u,uu, tmp));
1480      //uu.print();
1481      //exit(0);
1482
1483  uu.makeSpectral();
1484  f.makeSpectral();
1485
1486  int Mx = u.Mx();
1487  int My = u.My();
1488  int Mz = u.Mz();
1489  int kxmax = u.kxmax();
1490  int kzmax = u.kzmax();
1491  Real Lx = u.Lx();
1492  Real Lz = u.Lz();
1493
1494  ComplexChebyCoeff tmpProfile(My, u.a(), u.b(), Spectral);
1495  ComplexChebyCoeff tmpProfile_y(My, u.a(), u.b(), Spectral);
```

```
1496
1497   pool->executeTask(new skewsymmetricNL_task3(u,uu,f,kxmax,kzmax,Lx,Lz,Mx,My,Mz));
1498  // pool->executeTask(new skewsymmetricNL_task4(u,uu,f,tmpProfile,tmpProfile_y,kxmax,kzmax,Lx,Lz,Mx,My,Mz));
1499
1500   // Now set f_i += d/dx_j (u_i u_j)
1501
1502   for (int i=0; i<3; ++i) {
1503     //int i0 = i3j(i,0);
1504     int i1 = i3j(i,1);
1505     //int i2 = i3j(i,2);
1506     // Add in du_i/dy, that is d/dx_j (u_i u_j) for j=1
1507     for (int mx=0; mx<Mx; ++mx)
1508       for (int mz=0; mz<Mz; ++mz) {
1509         for (int my=0; my<My; ++my)
1510           tmpProfile.set(my, uu.cmplx(mx,my,mz,i1));
1511         diff(tmpProfile, tmpProfile_y);
1512         for (int my=0; my<My; ++my)
1513           f.cmplx(mx,my,mz,i) += 0.5*tmpProfile_y[my]; // j=1
1514       }
1515   }
1516
1517
1518   u.makeSpectral();
1519   U.makeSpectral();
1520   u -= U;
1521
1522   // Possible extra transforms if entry states are Physical
1523   u.makeState(uxzstate, uystate);
1524   U.makeState(Ustate);
1525
1526 }
```

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.

Here is the caller graph for this function:



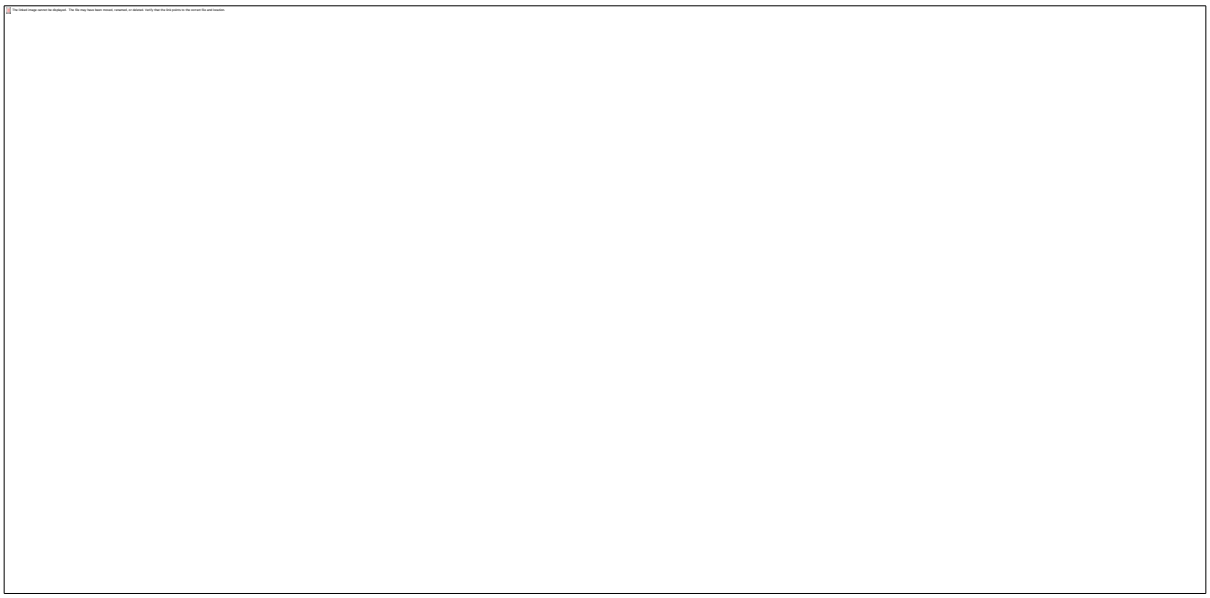
1.10.4.36 void up2q (const FlowField & *u*, const FlowField & *p*, FlowField & *q*, NonlinearMethod *nonlin_method*)

References FlowField::makePhysical(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Rotational, square(), FlowField::xzstate(), and FlowField::ystate().

```
1622                                {
1623
1624   assert(u_.Nd() == 3);
1625   assert(q.Nd() == 1);
1626   assert(p.Nd() == 1);
1627
1628   q = p;
1629   if (nonl_method != Rotational)
1630     return;
1631
1632   FlowField& u = (FlowField&) u_;
1633   fieldstate uxzstate = u.xzstate();
1634   fieldstate uystate = u.ystate();
1635
```

```
1636   u.makePhysical();
1637   q.makePhysical();
1638
1639   int Nx=u_.Nx();
1640   int Ny=u_.Ny();
1641   int Nz=u_.Nz();
1642
1643   // Add 1/2 u dot u to q
1644   for (int ny=0; ny<Ny; ++ny) {
1645     for (int nx=0; nx<Nx; ++nx)
1646       for (int nz=0; nz<Nz; ++nz)
1647         q(nx,ny,nz,0) += 0.5*(square(u(nx,ny,nz,0)) +
1648                               square(u(nx,ny,nz,1)) +
1649                               square(u(nx,ny,nz,2)));
1650   }
1651   u.makeState(uxzstate, uystate);
1652   q.makeState(p.xzstate(), p.ystate());
1653   return;
1654 }
```

Here is the call graph for this function:

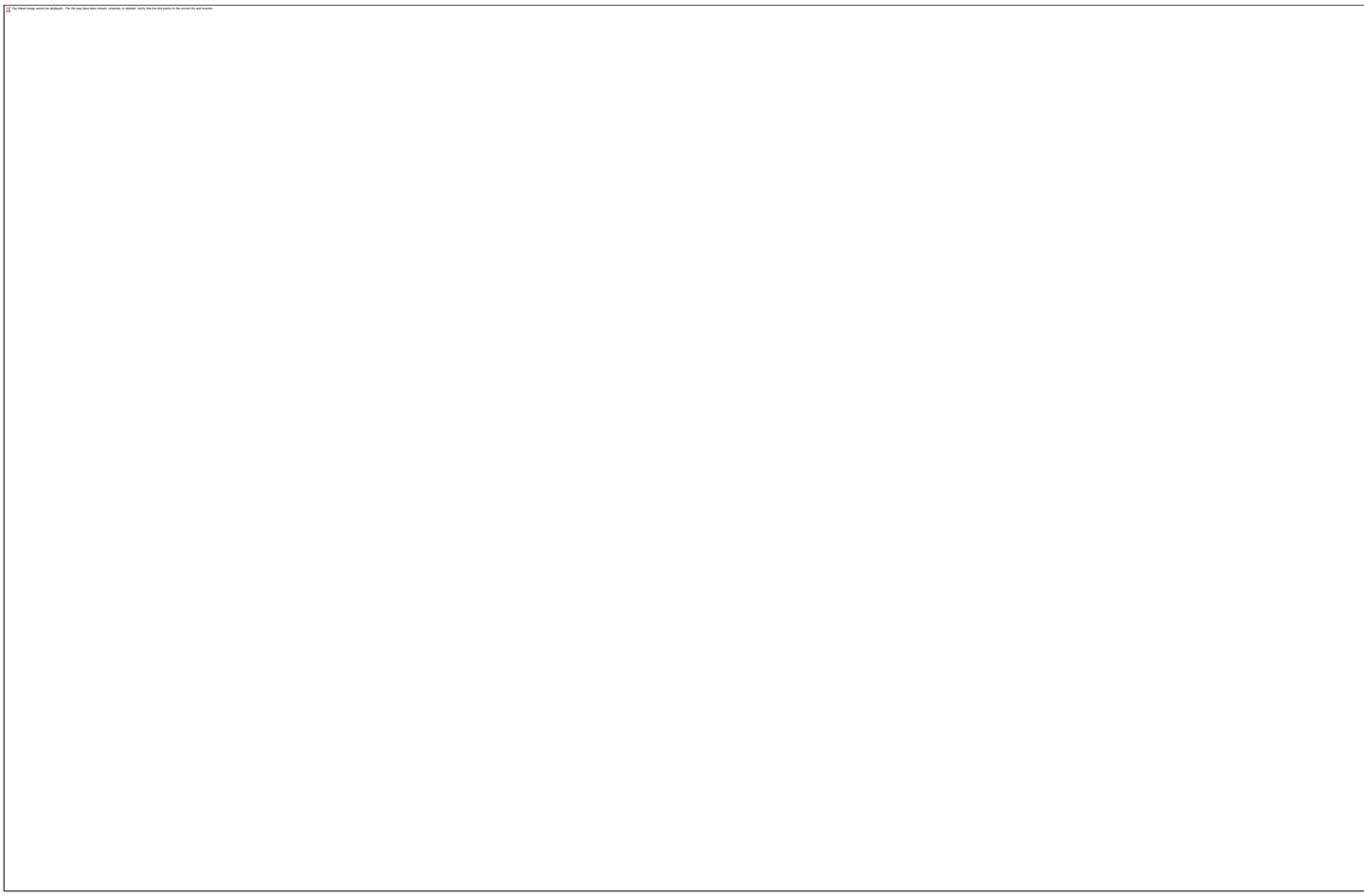


1.10.4.37 void uq2p (const FlowField & *u*, const FlowField & *q*, FlowField & *p*, NonlinearMethod *nonlin_method*)

References FlowField::makePhysical(), FlowField::makeState(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Rotational, square(), FlowField::xzstate(), and FlowField::ystate().

```
1588                                {
1589
1590   assert(u_.Nd() == 3);
1591   assert(q.Nd() == 1);
1592   assert(p.Nd() == 1);
1593
1594   p = q;
1595   if (nonl_method != Rotational)
1596     return;
1597
1598   FlowField& u = (FlowField&) u_;
1599   fieldstate uxzstate = u.xzstate();
1600   fieldstate uystate = u.ystate();
1601
1602   u.makePhysical();
1603   p.makePhysical();
1604
1605   int Nx=u_.Nx();
1606   int Ny=u_.Ny();
1607   int Nz=u_.Nz();
1608
1609   // Remove 1/2 u dot u or 1/2 (u+U) dot (u+U) from p
1610   for (int ny=0; ny<Ny; ++ny)
1611     for (int nx=0; nx<Nx; ++nx)
1612       for (int nz=0; nz<Nz; ++nz)
1613         p(nx,ny,nz,0) -= 0.5*(square(u(nx,ny,nz,0)) +
1614                               square(u(nx,ny,nz,1)) +
1615                               square(u(nx,ny,nz,2)));
1616
1617   u.makeState(uxzstate, uystate);
1618   p.makeState(q.xzstate(), q.ystate());
1619 }
```

Here is the call graph for this function:



1.10.4.38 void xdiff (const FlowField & *f*, FlowField & *dfdx*, int *n* = 1)

References FlowField::a(), FlowField::b(), cferror(), FlowField::cmplx(), FlowField::congruent(), FlowField::kx(), FlowField::kxmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeSpectral_xz(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, pow(), FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

```
39                                      {
40   if (n==0)
41     return;
42
43   FlowField& f = (FlowField&) f_;
44   fieldstate sxz = f.xzstate();
45   fieldstate sy  = f.ystate();
46   f.makeSpectral_xz();
47
```

```
48  Real Lx = f.Lx();
49
50  if (!f.congruent(dfdx))
51    dfdx.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
52  else
53    dfdx.setToZero();
54  dfdx.setState(Spectral, sy);
55
56  Complex rot(0.0, 0.0);
57  switch(n % 4) {
58  case 0: rot = Complex( 1.0, 0.0); break;
59  case 1: rot = Complex( 0.0, 1.0); break;
60  case 2: rot = Complex(-1.0, 0.0); break;
61  case 3: rot = Complex( 0.0,-1.0); break;
62  default: cferror("xdiff(f, dfdx, n) : impossible: n % 4 > 4 !!"); break;
63  }
64
65  int Nd = f.Nd();
66  int Mx = f.Mx();
67  int My = f.My();
68  int Mz = f.Mz();
69  int kxmax = f.kxmax();
70
71  for (int i=0; i<Nd; ++i)
72    for (int my=0; my<My; ++my)
73      for (int mx=0; mx<Mx; ++mx) {
74        int kx = f.kx(mx);
75        Complex cx = rot*(pow(2*pi*kx/Lx, n)*zero_last_mode(kx,kxmax,n));
76        for (int mz=0; mz<Mz; ++mz)
77          dfdx.cmplx(mx,my,mz,i) = cx*f.cmplx(mx,my,mz,i);
78      }
79
80  f.makeState(sxz,sy);
81  dfdx.makeSpectral();
82  return;
83 }
```

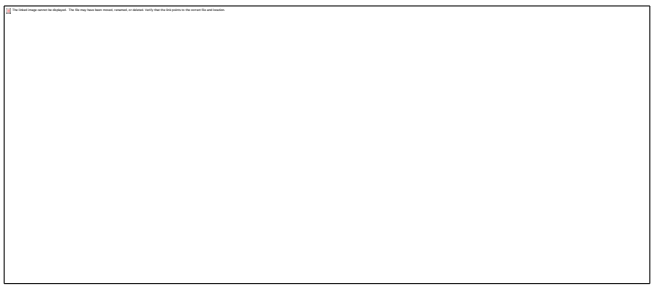
Here is the call graph for this function:

1.10.4.39 FlowField xdiff (const FlowField & *f*, int *n* = 1)

References xdiff().

```
825                                {
826   FlowField g;
827   xdiff(f,g,n);
828   return g;
829 }
```

Here is the call graph for this function:



1.10.4.40 void ydiff (const FlowField & *f*, FlowField & *dfdy*, int *n* = 1)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), diff(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeSpectral_y(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, swap(), FlowField::xzstate(), and FlowField::ystate().

```
85                                    {
86   if (n==0)
87     return;
88
89   FlowField& f = (FlowField&) f_;
90   fieldstate sxz = f.xzstate();
91   fieldstate sy  = f.ystate();
92   f.makeSpectral_y();
93
94   if (!f.congruent(dfdy))
95     dfdy.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
96   else
97     dfdy.setToZero();
98   dfdy.setState(sxz, Spectral);
99
100   int Nd = f.Nd();
101   if (sxz == Spectral) {
```

```
102   int Mx = f.Mx();
103   int My = f.My();
104   int Mz = f.Mz();
105
106   ComplexChebyCoeff  g(f.Ny(), f.a(), f.b(), Spectral);
107   ComplexChebyCoeff gy(f.Ny(), f.a(), f.b(), Spectral);
108
109   for (int i=0; i<Nd; ++i)
110     for (int mx=0; mx<Mx; ++mx)
111       for (int mz=0; mz<Mz; ++mz) {
112         for (int my=0; my<My; ++my)
113           g.set(my, f.cmplx(mx,my,mz,i));
114
115         // differentiate n times and leave derivative in *g
116         for (int k=0; k<n; ++k) {
117           diff(g,gy);
118           swap(g,gy);
119         }
120         for (int my=0; my<My; ++my)
121           dfdy.cmplx(mx,my,mz,i) = g[my];
122       }
123   }
124   else {
125     int Nx = f.Nx();
126     int Ny = f.Ny();
127     int Nz = f.Nz();
128     ChebyCoeff g(Ny,  f.a(), f.b(), Spectral);
129     ChebyCoeff gy(Ny, f.a(), f.b(), Spectral);
130
131     for (int i=0; i<Nd; ++i)
132       for (int nx=0; nx<Nx; ++nx)
133         for (int nz=0; nz<Nz; ++nz) {
134           for (int ny=0; ny<Ny; ++ny)
135             g[ny] = f(nx,ny,nz,i);
136
137           // differentiate n times and leave derivative in g
138           for (int k=0; k<n; ++k) {
139             diff(g,gy);
140             swap(g,gy);
141           }
142           for (int ny=0; ny<Ny; ++ny)
143             dfdy(nx,ny,nz,i) = g[ny];
144         }
145     }
146
147   f.makeState(sxz, sy);
```

```
148   dfdy.makeSpectral();
149 }
```

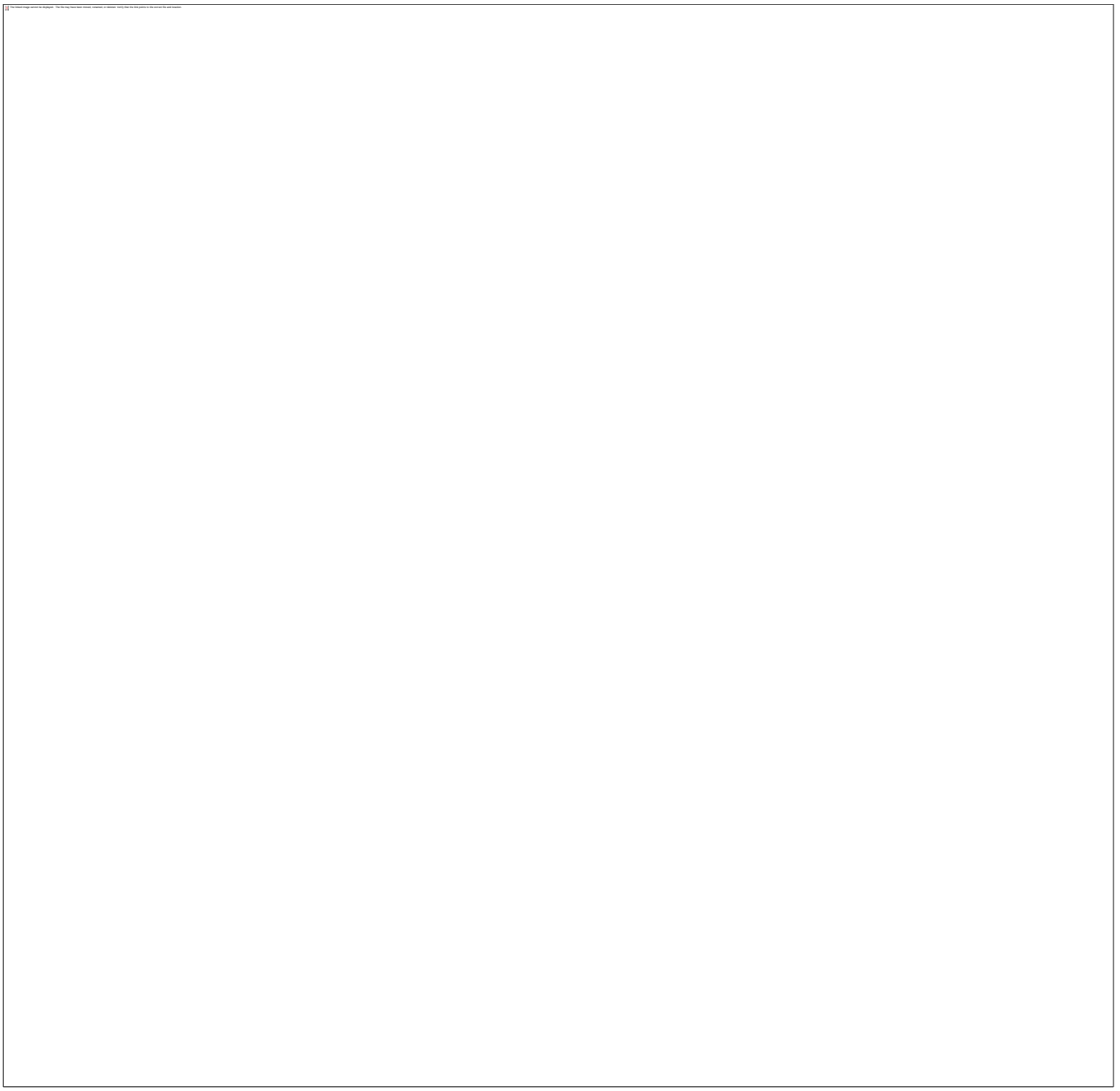
Here is the call graph for this function:

1.10.4.41 FlowField ydiff (const FlowField & *f*, int *n* = 1)

References ydiff().

```
831                                 {
832   FlowField g;
833   ydiff(f,g,n);
834   return g;
835 }
```

Here is the call graph for this function:



1.10.4.42 void zdiff (const FlowField & *f*, FlowField & *dfdz*, int *n* = 1)

References FlowField::a(), FlowField::b(), cferror(), FlowField::cmplx(), FlowField::congruent(), FlowField::kz(), FlowField::kzmax(), FlowField::Lx(), FlowField::Lz(), FlowField::makeSpectral(), FlowField::makeSpectral_xz(), FlowField::makeState(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), pi, pow(),

FlowField::resize(), FlowField::setState(), FlowField::setToZero(), Spectral, FlowField::xzstate(), FlowField::ystate(), and zero_last_mode().

```
152                                     {
153   if (n==0)
154     return;
155
156   FlowField& f = (FlowField&) f_;
157   fieldstate sxz = f.xzstate();
158   fieldstate sy  = f.ystate();
159   f.makeSpectral_xz();
160
161   // compute gradf(nx,ny,nz,i) = df(nx,ny,nz,0)/dx_i for scalar-valued f
162   if (!f.congruent(dfdz))
163     dfdz.resize(f.Nx(), f.Ny(), f.Nz(), f.Nd(), f.Lx(), f.Lz(), f.a(), f.b());
164   else
165     dfdz.setToZero();
166   dfdz.setState(Spectral, sy);
167
168   Complex rot(0.0, 0.0);
169   switch(n % 4) {
170   case 0: rot = Complex( 1.0, 0.0); break;
171   case 1: rot = Complex( 0.0, 1.0); break;
172   case 2: rot = Complex(-1.0, 0.0); break;
173   case 3: rot = Complex( 0.0,-1.0); break;
174   default: cferror("zdiff(f, dfdz, n) : impossible: n % 4 > 4 !!"); break;
175   }
176
177   int Nd = f.Nd();
178   int Mx = f.Mx();
179   int My = f.My();
180   int Mz = f.Mz();
181   int kzmax = f.kzmax();
182   Real Lz = f.Lz();
183
184   Real az = 2*pi/Lz;
185   for (int i=0; i<Nd; ++i)
186     for (int my=0; my<My; ++my)
187       for (int mx=0; mx<Mx; ++mx)
188         for (int mz=0; mz<Mz; ++mz) {
189           int kz = f.kz(mz);
190           Complex cz = rot*(pow(az*kz, n)*zero_last_mode(kz,kzmax,n));
191           dfdz.cmplx(mx,my,mz,i) = cz*f.cmplx(mx,my,mz,i);
192         }
193
194   f.makeState(sxz,sy);
195   dfdz.makeSpectral();
```

```
196   return;
197 }
```

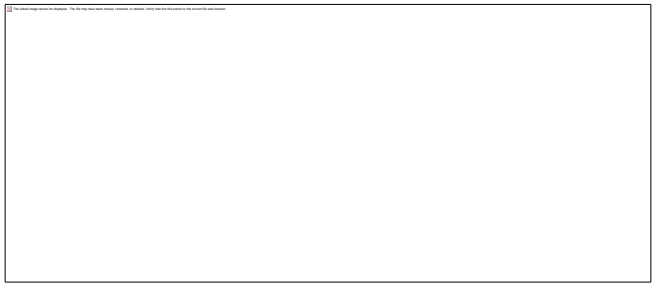
Here is the call graph for this function:

1.10.4.43 FlowField zdiff (const FlowField & *f*, int *n* = 1)

References zdiff().

```
837                                {
838   FlowField g;
839   zdiff(f,g,n);
840   return g;
841 }
```

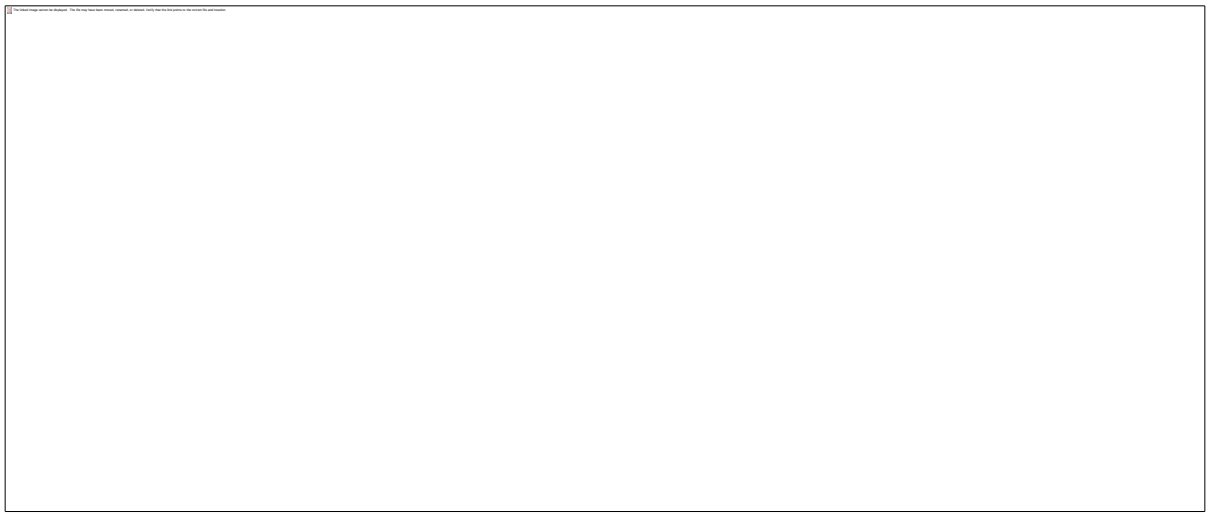
Here is the call graph for this function:



1.10.4.44

1.11 /_cuda/channelflow-0.9.22/channelflow/dns.cpp File Reference

#include "channelflow/dns.h"
#include <stdlib.h>
Include dependency graph for dns.cpp:



1.11.1 Functions

- void * start_thread (void *arg)
- void changeBaseFlow (const ChebyCoeff &ubase0, const FlowField &ufluc0, const FlowField &q0arg, const ChebyCoeff &ubase1, FlowField &u1, FlowField &q1)
- ostream & operator<< (ostream &os, Dealiasing d)
- ostream & operator<< (ostream &os, MeanConstraint m)
- ostream & operator<< (ostream &os, TimeStepMethod t)
- ostream & operator<< (ostream &os, Verbosity v)
- ostream & operator<< (ostream &os, DNSFlags &flags)

1.11.2 Variables

- const Real EPSILON = 1e-12

1.11.3 Function Documentation

1.11.3.1 void changeBaseFlow (const ChebyCoeff & *ubase0*, const FlowField & *ufluc0*, const FlowField & *q0arg*, const ChebyCoeff & *ubase1*, FlowField & *u1*, FlowField & *q1*)

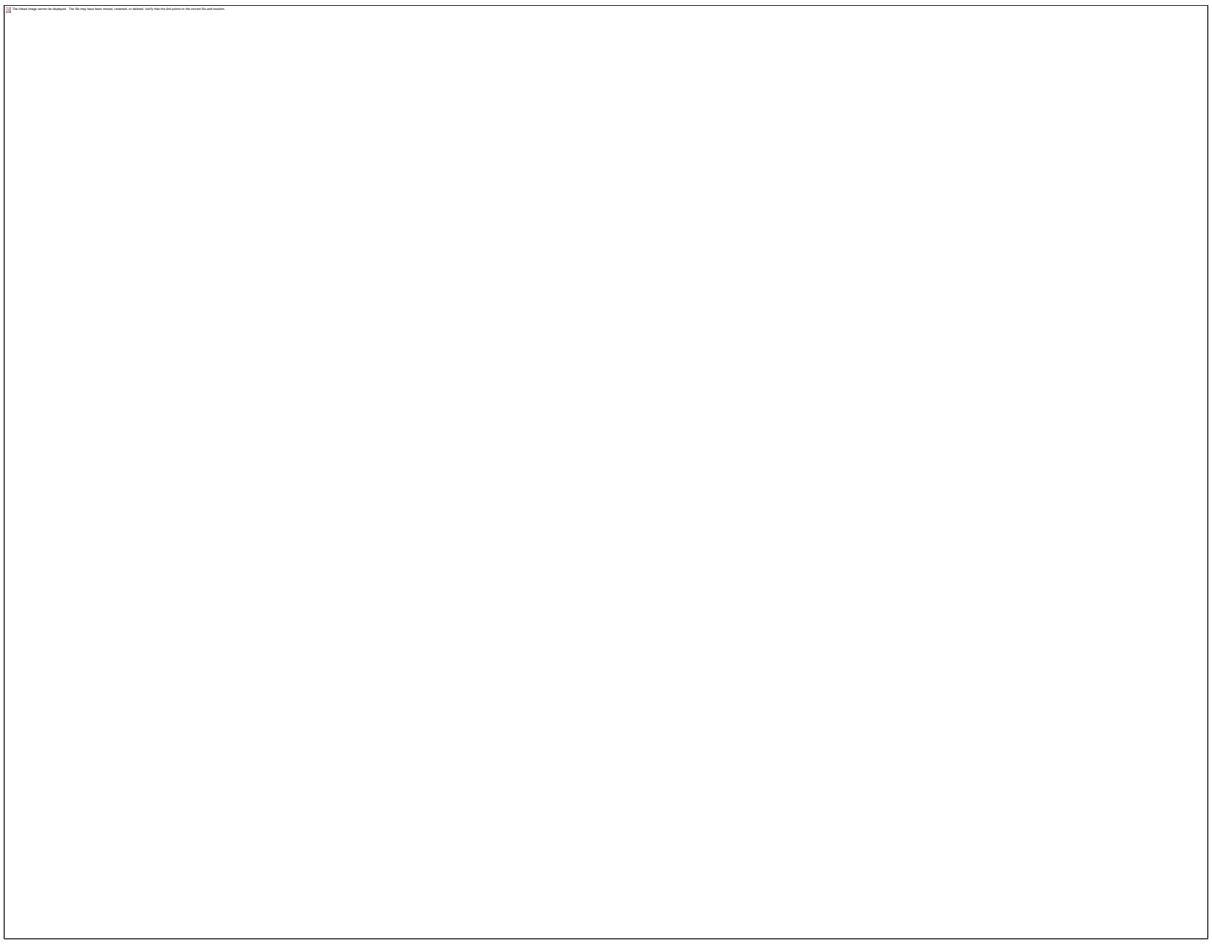
References FlowField::addProfile(), ChebyCoeff::makePhysical(), FlowField::makePhysical(), ChebyCoeff::makeState(), FlowField::makeState(), ChebyCoeff::numModes(), FlowField::numXgridpts(), FlowField::numYgridpts(),

FlowField::numZgridpts(), Physical, Spectral, square(), ChebyCoeff::state(), FlowField::xzstate(), and FlowField::ystate().

```
2414                                                    {
2415  ChebyCoeff& U0 = (ChebyCoeff&) ubase0;
2416  fieldstate U0state = U0.state();
2417
2418  ChebyCoeff& U1 = (ChebyCoeff&) ubase1;
2419  fieldstate U1state = U1.state();
2420
2421  FlowField& u0 = (FlowField&) ufluc0;
2422  fieldstate u0xzstate = u0.xzstate();
2423  fieldstate u0ystate = u0.ystate();
2424
2425  FlowField& q0 = (FlowField&) q0arg;
2426  fieldstate q0xzstate = q0.xzstate();
2427  fieldstate q0ystate = q0.ystate();
2428
2429  int Nx=u0.numXgridpts();
2430  int Ny=u0.numYgridpts();
2431  int Nz=u0.numZgridpts();
2432
2433  u1 = u0; // want u1 FPF
2434  u1.makeState(Spectral, Physical);
2435  u0.makePhysical();
2436  q0.makePhysical();
2437  q1 = q0; // want q1 physical
2438
2439  // At this point
2440  // u1 == utot - U0
2441  // q1 == p + 1/2 u0 dot u0
2442
2443  // Remove 1/2 u0 dot u0 from q1
2444  for (int ny=0; ny<Ny; ++ny)
2445    for (int nx=0; nx<Nx; ++nx)
2446      for (int nz=0; nz<Nz; ++nz)
2447        q1(nx,ny,nz,0) -= 0.5*(square(u0(nx,ny,nz,0)) +
2448                               square(u0(nx,ny,nz,1)) +
2449                               square(u0(nx,ny,nz,2)));
2450  // At this point
2451  // u1 == utot - U0
2452  // q1 == p
2453
2454  ChebyTransform t(U0.numModes());
2455  U0.makePhysical(t);
2456  U1.makePhysical(t);
2457
```

```
2458   // Add U0-U1 to u1
2459   ChebyCoeff delta_U(U0);
2460   delta_U -= U1;
2461   u1.addProfile(delta_U);
2462   u1.makePhysical();
2463
2464   // At this point
2465   // u1 == utot - U1
2466   // q1 == p
2467
2468   // Add 1/2 u1 dot u1 to q1
2469   for (int ny=0; ny<Ny; ++ny)
2470     for (int nx=0; nx<Nx; ++nx)
2471       for (int nz=0; nz<Nz; ++nz)
2472         q1(nx,ny,nz,0) += 0.5*(square(u1(nx,ny,nz,0)) +
2473                                square(u1(nx,ny,nz,1)) +
2474                                square(u1(nx,ny,nz,2)));
2475   // At this point
2476   // u1 == utot - U1
2477   // q1 == p + 1/2 u1 dot u1
2478   // et, voila
2479
2480   U0.makeState(U0state,t);
2481   U1.makeState(U1state,t);
2482   u0.makeState(u0xzstate, u0ystate);
2483   q0.makeState(q0xzstate, q0ystate);
2484   u1.makeState(u0xzstate, u0ystate);
2485   q1.makeState(q0xzstate, q0ystate);
2486 }
```

Here is the call graph for this function:



1.11.3.2 ostream& operator<< (ostream & *os*, DNSFlags & *flags*)

References DNSFlags::constraint, DNSFlags::dealiasing, DNSFlags::initstepping, DNSFlags::nonlinearity, DNSFlags::taucorrection, DNSFlags::timestepping, and DNSFlags::verbosity.

```
2576                                        {
2577   string s(", ");
2578   string tau = (flags.taucorrection) ? "TauCorrection" : "NoTauCorrection";
2579   os << flags.timestepping << s
2580      << flags.initstepping << s
2581      << flags.nonlinearity << s
2582      << flags.dealiasing << s
2583      << flags.constraint << s
2584      << tau << s
2585      << flags.verbosity;
2586   return os;
2587 }
```

1.11.3.3 ostream& operator<< (ostream & *os*, Verbosity *v*)

References PrintAll, PrintTicks, PrintTime, Silent, and VerifyTauSolve.

```
2562                                    {
2563   string s;
2564   switch(v) {
2565   case Silent: s="Silent"; break;
2566   case PrintTicks: s="PrintTicks"; break;
2567   case PrintTime: s="PrintTime"; break;
2568   case VerifyTauSolve: s="VerifyTauSolve"; break;
2569   case PrintAll: s="PrintAll"; break;
2570   default: s="Invalid Verbosity value: please submit bug report";
2571   }
2572   os << s;
2573   return os;
2574 }
```

1.11.3.4 ostream& operator<< (ostream & *os*, TimeStepMethod *t*)

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

```
2545                                    {
2546   string s;
2547   switch(t) {
2548   case CNFE1: s="CNFE1"; break;
2549   case CNAB2: s="CNAB2"; break;
2550   case CNRK2: s="CNRK2"; break;
2551   case SMRK2: s="SMRK2"; break;
2552   case SBDF1: s="SBDF1"; break;
2553   case SBDF2: s="SBDF2"; break;
2554   case SBDF3: s="SBDF3"; break;
2555   case SBDF4: s="SBDF4"; break;
2556   case PPEQ2: s="PPEQ2"; break;
2557   default: s="Invalid TimeStepMethod value: please submit bug report";
2558   }
2559   os << s;
2560   return os;
2561 }
```

1.11.3.5 ostream& operator<< (ostream & *os*, MeanConstraint *m*)

References BulkVelocity, and PressureGradient.

```
2534                                    {
```

```
2535   string s;
2536   switch(m) {
2537   case PressureGradient: s="PressureGradient"; break;
2538   case BulkVelocity: s="BulkVelocity"; break;
2539   default: s="Invalid MeanConstraint value: please submit bug report";
2540   }
2541   os << s;
2542   return os;
2543 }
```

1.11.3.6 ostream& operator<< (ostream & *os*, Dealiasing *d*)

References DealiasXYZ, DealiasXZ, DealiasY, and NoDealiasing.

```
2521                                    {
2522   string s;
2523   switch(d) {
2524   case NoDealiasing: s="NoDealiasing"; break;
2525   case DealiasXZ: s="DealiasXZ"; break;
2526   case DealiasY: s="DealiasY"; break;
2527   case DealiasXYZ: s="DealiasXYZ"; break;
2528   default: s="Invalid Dealiasing value: please submit bug report";
2529   }
2530   os << s;
2531   return os;
2532 }
```

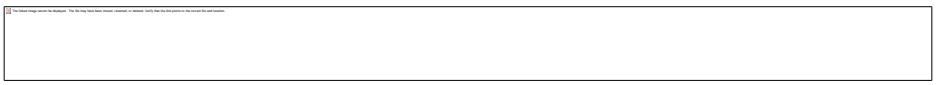
1.11.3.7 void* start_thread (void * *arg*)

References ThreadPool::Run().

Referenced by ThreadPool::CreateThreads().

```
406 {
407   ThreadPool* tp = (ThreadPool*) arg;
408   tp->Run();
409   return NULL;
410 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

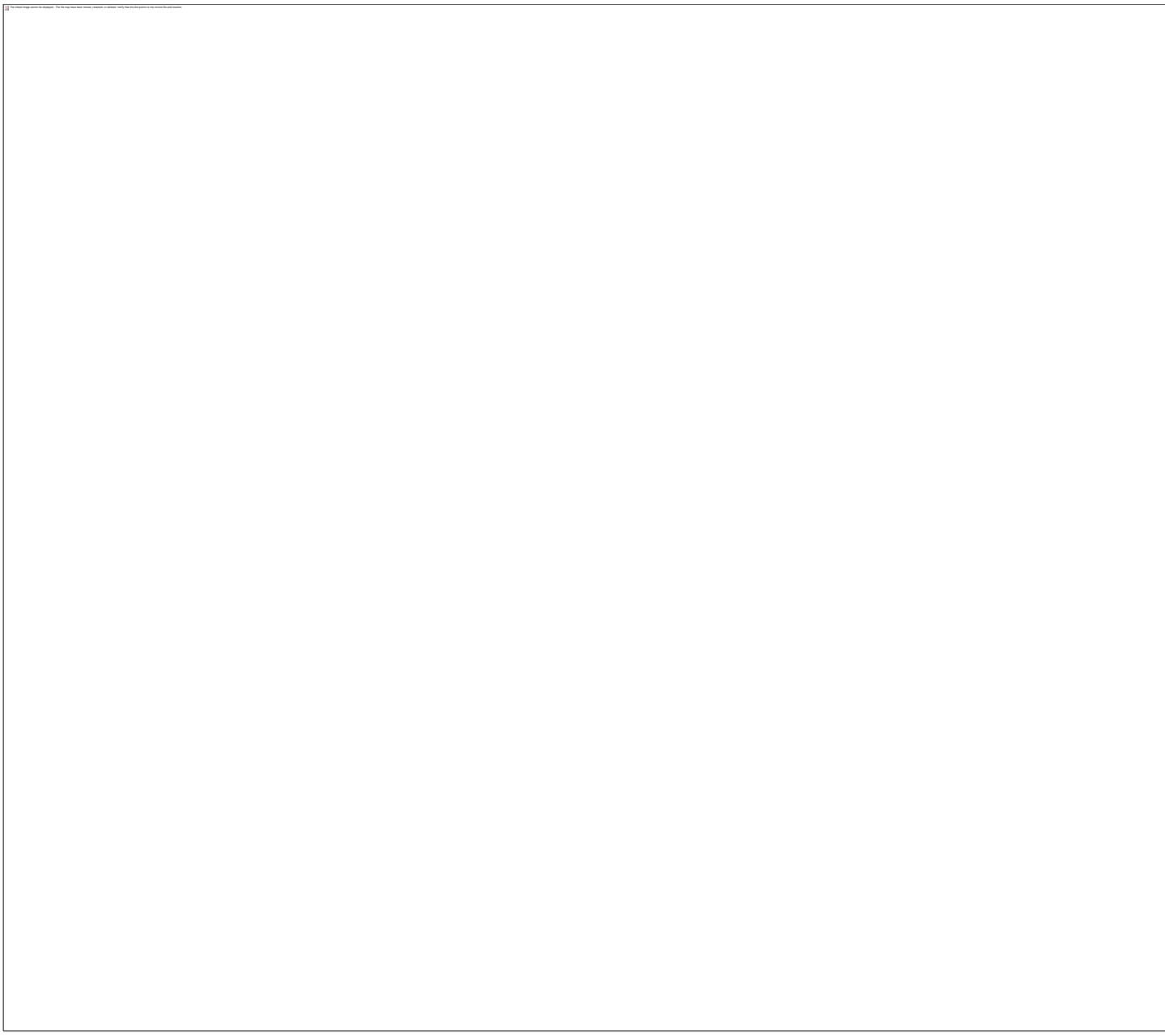
1.11.4 Variable Documentation

1.11.4.1 const Real EPSILON = 1e-12

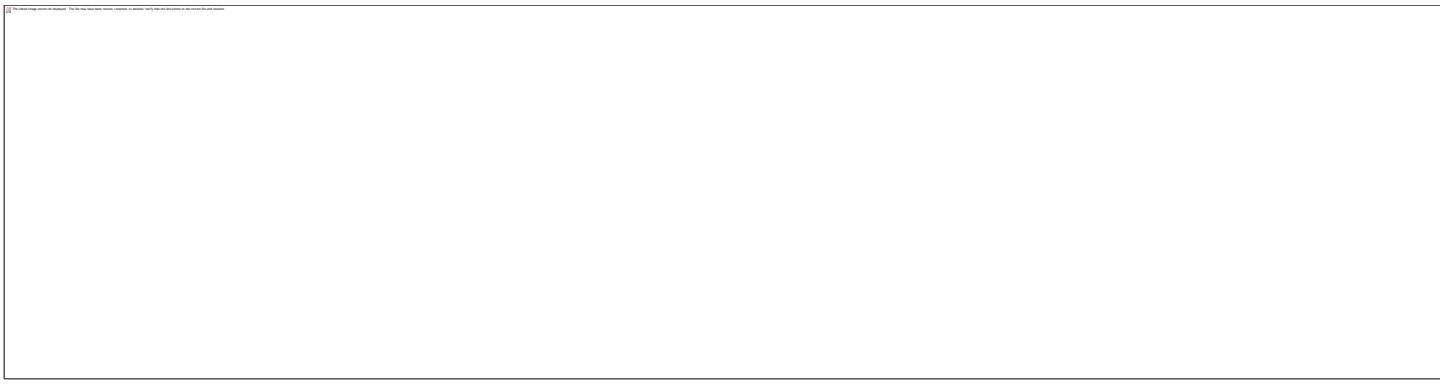
1.11.4.2

1.12 /_cuda/channelflow-0.9.22/channelflow/dns.h File Reference

#include "channelflow/mathdefs.h"
#include "channelflow/vector.h"
#include "channelflow/chebyshev.h"
#include "channelflow/array.h"
#include "channelflow/flowfield.h"
#include "channelflow/diffops.h"
#include "channelflow/tausolver.h"
#include <pthread.h>
#include <string>
#include <vector>
#include <map>
#include <list>
#include <algorithm>
#include <queue>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "channelflow/poissonsolver.h"
Include dependency graph for dns.h:



This graph shows which files directly or indirectly include this file:



1.12.1 Classes

- class cmndline
- class Attributes
- class BaseTask
- class skewsymmetricNL_task
- class skewsymmetricNL_task2
- class skewsymmetricNL_task3
- class skewsymmetricNL_task4
- class Limits
- class ThreadPool
- class DNSFlags
- class TimeStep
- class DNS
- class DNSAlgorithm
- class MultistepDNS
- class RungeKuttaDNS
- class CNABstyleDNS
- class PPEDNS

1.12.2 Enumerations

- enum Dealiasing { NoDealiasing, DealiasXZ, DealiasY, DealiasXYZ }
- enum TimeStepMethod
  { CNFE1, CNAB2, CNRK2, SMRK2, SBDF1, SBDF2, SBDF3, SBDF4, PPEQ2 }
- enum MeanConstraint { PressureGradient, BulkVelocity }
- enum Verbosity { Silent, PrintTicks, PrintTime, VerifyTauSolve, PrintAll }
- enum HowToRun { CPU, GPU, THREAD }

1.12.3 Functions

- std::ostream & operator<< (std::ostream &os, Dealiasing d)
- std::ostream & operator<< (std::ostream &os, MeanConstraint m)
- std::ostream & operator<< (std::ostream &os, TimeStepMethod t)

- std::ostream & operator<< (std::ostream &os, Verbosity v)
- std::ostream & operator<< (std::ostream &os, DNSFlags &flags)
- void changeBaseFlow (const ChebyCoeff &U0, const FlowField &u0, const FlowField &q0, const ChebyCoeff &U1, FlowField &u1, FlowField &q1)

1.12.4 Enumeration Type Documentation

1.12.4.1 enum Dealiasing

Enumerator:

NoDealiasing
DealiasXZ
DealiasY
DealiasXYZ

```
55 {NoDealiasing, DealiasXZ, DealiasY, DealiasXYZ};
```

1.12.4.2 enum HowToRun

Enumerator:

CPU
GPU
THREAD

```
68 {CPU,GPU,THREAD};
```

1.12.4.3 enum MeanConstraint

Enumerator:

PressureGradient
BulkVelocity

```
58 {PressureGradient, BulkVelocity};
```

1.12.4.4 enum TimeStepMethod

Enumerator:

CNFE1
CNAB2
CNRK2

SMRK2
SBDF1
SBDF2
SBDF3
SBDF4
PPEQ2

```
56                {CNFE1, CNAB2, CNRK2, SMRK2,
57                 SBDF1, SBDF2, SBDF3, SBDF4, PPEQ2};
```

1.12.4.5 enum Verbosity

Enumerator:

Silent
PrintTicks
PrintTime
VerifyTauSolve
PrintAll

```
59 {Silent, PrintTicks, PrintTime, VerifyTauSolve, PrintAll};
```

1.12.5 Function Documentation

1.12.5.1 void changeBaseFlow (const ChebyCoeff & *U0*, const FlowField & *u0*, const FlowField & *q0*, const ChebyCoeff & *U1*, FlowField & *u1*, FlowField & *q1*)

References FlowField::addProfile(), ChebyCoeff::makePhysical(), FlowField::makePhysical(), ChebyCoeff::makeState(), FlowField::makeState(), ChebyCoeff::numModes(), FlowField::numXgridpts(), FlowField::numYgridpts(), FlowField::numZgridpts(), Physical, Spectral, square(), ChebyCoeff::state(), FlowField::xzstate(), and FlowField::ystate().

```
2414                                                    {
2415   ChebyCoeff& U0 = (ChebyCoeff&) ubase0;
2416   fieldstate U0state = U0.state();
2417
2418   ChebyCoeff& U1 = (ChebyCoeff&) ubase1;
2419   fieldstate U1state = U1.state();
2420
2421   FlowField& u0 = (FlowField&) ufluc0;
2422   fieldstate u0xzstate = u0.xzstate();
2423   fieldstate u0ystate = u0.ystate();
2424
```

```
2425  FlowField& q0 = (FlowField&) q0arg;
2426  fieldstate q0xzstate = q0.xzstate();
2427  fieldstate q0ystate = q0.ystate();
2428
2429  int Nx=u0.numXgridpts();
2430  int Ny=u0.numYgridpts();
2431  int Nz=u0.numZgridpts();
2432
2433  u1 = u0; // want u1 FPF
2434  u1.makeState(Spectral, Physical);
2435  u0.makePhysical();
2436  q0.makePhysical();
2437  q1 = q0; // want q1 physical
2438
2439  // At this point
2440  // u1 == utot - U0
2441  // q1 == p + 1/2 u0 dot u0
2442
2443  // Remove 1/2 u0 dot u0 from q1
2444  for (int ny=0; ny<Ny; ++ny)
2445    for (int nx=0; nx<Nx; ++nx)
2446      for (int nz=0; nz<Nz; ++nz)
2447        q1(nx,ny,nz,0) -= 0.5*(square(u0(nx,ny,nz,0)) +
2448                               square(u0(nx,ny,nz,1)) +
2449                               square(u0(nx,ny,nz,2)));
2450  // At this point
2451  // u1 == utot - U0
2452  // q1 == p
2453
2454  ChebyTransform t(U0.numModes());
2455  U0.makePhysical(t);
2456  U1.makePhysical(t);
2457
2458  // Add U0-U1 to u1
2459  ChebyCoeff delta_U(U0);
2460  delta_U -= U1;
2461  u1.addProfile(delta_U);
2462  u1.makePhysical();
2463
2464  // At this point
2465  // u1 == utot - U1
2466  // q1 == p
2467
2468  // Add 1/2 u1 dot u1 to q1
2469  for (int ny=0; ny<Ny; ++ny)
2470    for (int nx=0; nx<Nx; ++nx)
```

```
2471      for (int nz=0; nz<Nz; ++nz)
2472        q1(nx,ny,nz,0) += 0.5*(square(u1(nx,ny,nz,0)) +
2473                               square(u1(nx,ny,nz,1)) +
2474                               square(u1(nx,ny,nz,2)));
2475   // At this point
2476   // u1 == utot - U1
2477   // q1 == p + 1/2 u1 dot u1
2478   // et, voila
2479
2480   U0.makeState(U0state,t);
2481   U1.makeState(U1state,t);
2482   u0.makeState(u0xzstate, u0ystate);
2483   q0.makeState(q0xzstate, q0ystate);
2484   u1.makeState(u0xzstate, u0ystate);
2485   q1.makeState(q0xzstate, q0ystate);
2486 }
```

Here is the call graph for this function:

1.12.5.2 std::ostream& operator<< (std::ostream & *os*, DNSFlags & *flags*)

References DNSFlags::constraint, DNSFlags::dealiasing, DNSFlags::initstepping, DNSFlags::nonlinearity, DNSFlags::taucorrection, DNSFlags::timestepping, and DNSFlags::verbosity.

```
2576                                                {
2577   string s(", ");
2578   string tau = (flags.taucorrection) ? "TauCorrection" : "NoTauCorrection";
2579   os << flags.timestepping << s
2580      << flags.initstepping << s
2581      << flags.nonlinearity << s
2582      << flags.dealiasing << s
2583      << flags.constraint << s
2584      << tau << s
2585      << flags.verbosity;
2586   return os;
2587 }
```

1.12.5.3 std::ostream& operator<< (std::ostream & *os*, Verbosity *v*)

References PrintAll, PrintTicks, PrintTime, Silent, and VerifyTauSolve.

```
2562                                                {
2563   string s;
2564   switch(v) {
2565   case Silent: s="Silent"; break;
2566   case PrintTicks: s="PrintTicks"; break;
2567   case PrintTime: s="PrintTime"; break;
2568   case VerifyTauSolve: s="VerifyTauSolve"; break;
2569   case PrintAll: s="PrintAll"; break;
2570   default: s="Invalid Verbosity value: please submit bug report";
2571   }
2572   os << s;
2573   return os;
2574 }
```

1.12.5.4 std::ostream& operator<< (std::ostream & *os*, TimeStepMethod *t*)

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

```
2545                                                {
2546   string s;
```

```
2547   switch(t) {
2548   case CNFE1: s="CNFE1"; break;
2549   case CNAB2: s="CNAB2"; break;
2550   case CNRK2: s="CNRK2"; break;
2551   case SMRK2: s="SMRK2"; break;
2552   case SBDF1: s="SBDF1"; break;
2553   case SBDF2: s="SBDF2"; break;
2554   case SBDF3: s="SBDF3"; break;
2555   case SBDF4: s="SBDF4"; break;
2556   case PPEQ2: s="PPEQ2"; break;
2557   default: s="Invalid TimeStepMethod value: please submit bug report";
2558   }
2559   os << s;
2560   return os;
2561 }
```

1.12.5.5 std::ostream& operator<< (std::ostream & *os*, MeanConstraint *m*)

References BulkVelocity, and PressureGradient.

```
2534                                        {
2535   string s;
2536   switch(m) {
2537   case PressureGradient: s="PressureGradient"; break;
2538   case BulkVelocity: s="BulkVelocity"; break;
2539   default: s="Invalid MeanConstraint value: please submit bug report";
2540   }
2541   os << s;
2542   return os;
2543 }
```

1.12.5.6 std::ostream& operator<< (std::ostream & *os*, Dealiasing *d*)

References DealiasXYZ, DealiasXZ, DealiasY, and NoDealiasing.

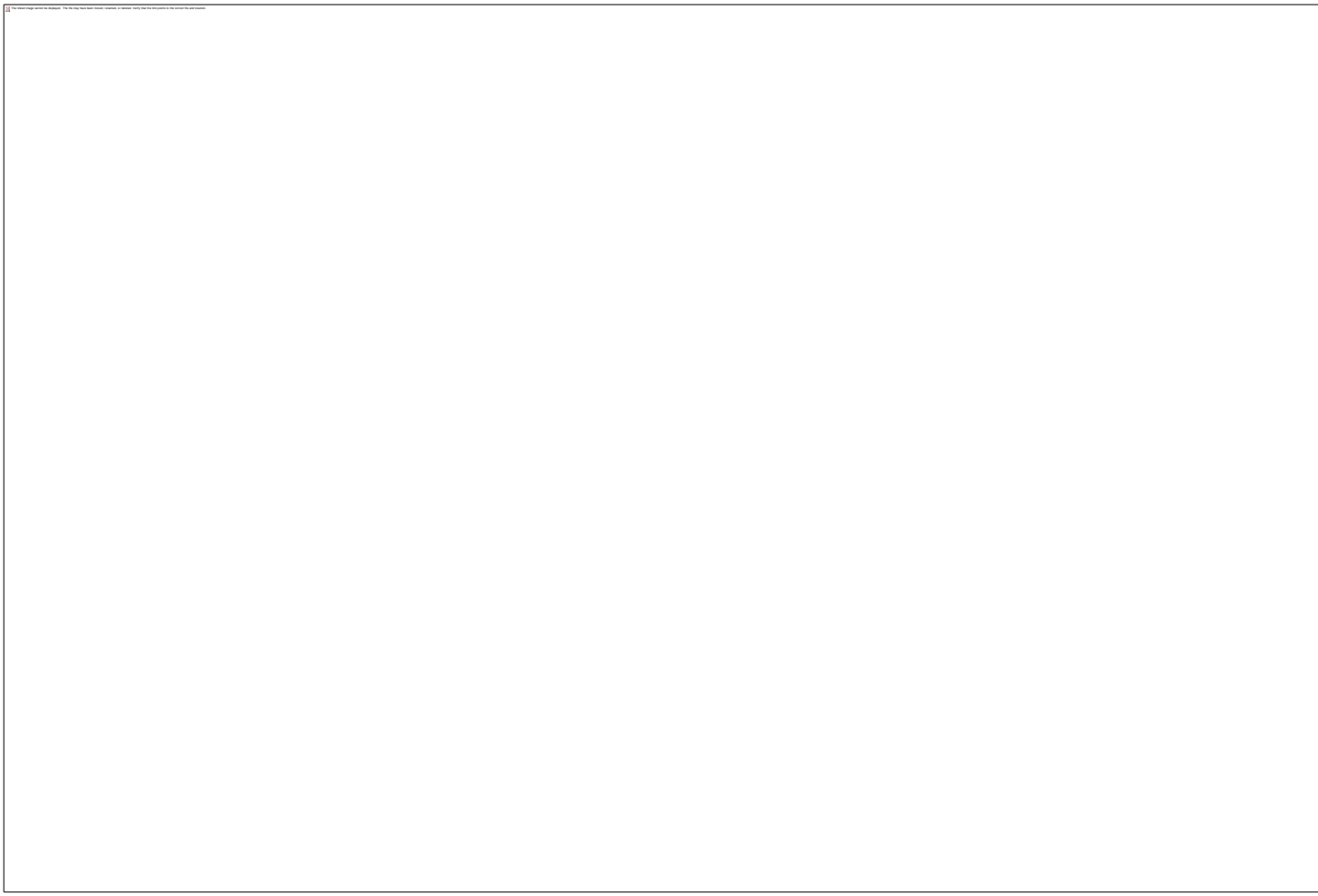
```
2521                                        {
2522   string s;
2523   switch(d) {
2524   case NoDealiasing: s="NoDealiasing"; break;
2525   case DealiasXZ: s="DealiasXZ"; break;
2526   case DealiasY: s="DealiasY"; break;
2527   case DealiasXYZ: s="DealiasXYZ"; break;
2528   default: s="Invalid Dealiasing value: please submit bug report";
2529   }
2530   os << s;
2531   return os;
```

```
2532 }
```

1.12.5.7

1.13 /_cuda/channelflow-0.9.22/channelflow/flowfield.cpp File Reference

#include <fstream>
#include <iomanip>
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include "channelflow/flowfield.h"
Include dependency graph for flowfield.cpp:



1.13.1 Functions

- void swap (FlowField &f, FlowField &g)
- Real bcNorm2 (const FlowField &f, bool normalize)
- Real bcDist2 (const FlowField &f, const FlowField &g, bool normalize)
- Real bcNorm (const FlowField &f, bool normalize)
- Real bcDist (const FlowField &f, const FlowField &g, bool normalize)
- Real divNorm (const FlowField &f, bool normalize)
- Real divNorm2 (const FlowField &f, bool normalize)

- Real divDist2 (const FlowField &f, const FlowField &g, bool normalize)
- Real divDist (const FlowField &f, const FlowField &g, bool normalize)
- Real L1Norm (const FlowField &u, bool normalize)
- Real L1Dist (const FlowField &f, const FlowField &g, bool normalize)
- Real LinfNorm (const FlowField &u)
- Real LinfDist (const FlowField &f, const FlowField &g)
- Real L2Dist (const FlowField &u, const FlowField &v, bool normalize)
- Real L2Dist2 (const FlowField &u, const FlowField &v, bool normalize)
- Real L2Norm (const FlowField &u, bool normalize)
- Real L2Norm2 (const FlowField &u, bool normalize)
- Real L2InnerProduct (const FlowField &f, const FlowField &g, bool normalize)
- Real L2Norm2 (const FlowField &u, int kxmax, int kzmax, bool normalize)
- Real L2Norm (const FlowField &u, int kxmax, int kzmax, bool normalize)
- Real L2Dist (const FlowField &u, const FlowField &v, int kxmax, int kzmax, bool normalize)
- Real L2Dist2 (const FlowField &u, const FlowField &v, int kxmax, int kzmax, bool normalize)
- Complex L2InnerProduct (const FlowField &u, const BasisFunc &phi, bool normalize)
- void randomUprofile (ComplexChebyCoeff &u, Real mag, Real decay)
- void randomVprofile (ComplexChebyCoeff &v, Real mag, Real decay)
- void chebyUprofile (ComplexChebyCoeff &u, int n, Real decay)
- void chebyVprofile (ComplexChebyCoeff &v, int n, Real decay)
- void randomProfile (ComplexChebyCoeff &u, ComplexChebyCoeff &v, ComplexChebyCoeff &w, int kx, int kz, Real Lx, Real Lz, Real mag, Real decay)
- void chebyProfile (ComplexChebyCoeff &u, ComplexChebyCoeff &v, ComplexChebyCoeff &w, int un, int vn, int kx, int kz, Real Lx, Real Lz, Real decay)
- void assignOrrSommField (FlowField &u, FlowField &P, Real t, Real Reynolds, Complex omega, const ComplexChebyCoeff &ueig, const ComplexChebyCoeff &veig, const ComplexChebyCoeff &peig)

1.13.2 Variables

- const Real EPSILON = 1e-4

1.13.3 Function Documentation

1.13.3.1 void assignOrrSommField (FlowField & *u*, FlowField & *P*, Real *t*, Real *Reynolds*, Complex *omega*, const ComplexChebyCoeff & *ueig*, const ComplexChebyCoeff & *veig*, const ComplexChebyCoeff & *peig*)

References FlowField::chebyfft_y(), FlowField::cmplx(), exp(), I(), FlowField::irealfft_xz(), FlowField::Mx(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::realfft_xz(), FlowField::setState(), FlowField::setToZero(), Spectral, square(), and FlowField::y().

```
2442                                        {
2443
2444   int Ny=u.Ny();
2445
2446   // Reconstruct velocity field (Poisseuille plus OS perturbation) from
2447   // y-profile of (kx,kz) == (1,0) Spectral mode of pertubation (ueig & veig).
2448   u.setState(Spectral, Physical);
2449   u.setToZero();
2450   Complex c = exp((-t*omega)*I);
2451   int n=u.Mx()-1;
2452   int ny; // MSVC++ FOR-SCOPE BUG
2453   for (ny=0; ny<Ny; ++ny) {
2454     Complex uc = c*ueig[ny];
2455     Complex vc = c*veig[ny];
2456     u.cmplx(0, ny, 0, 0) = Complex(1.0-square(u.y(ny)));
2457     u.cmplx(1, ny, 0, 0) = uc;
2458     u.cmplx(1, ny, 0, 1) = vc;
2459     u.cmplx(n, ny, 0, 0) = conj(uc);
2460     u.cmplx(n, ny, 0, 1) = conj(vc);
2461   }
2462
2463   // Assign pressure perturbation p to P field.
2464   P.setState(Spectral, Physical);
2465   P.setToZero();
2466   for (ny=0; ny<Ny; ++ny) {
2467     Complex pc = c*peig[ny];
2468     P.cmplx(1, ny, 0, 0) = pc;
2469     P.cmplx(n, ny, 0, 0) = conj(pc);
2470   }
2471
2472   // Add velocity contrib to get modified pressure P = p + 1/2 |u|^2
2473   u.irealfft_xz();
2474   P.irealfft_xz();
2475
2476   for (ny=0; ny<Ny; ++ny)
2477     for (int nx=0; nx<u.Nx(); ++nx)
2478       for (int nz=0; nz<u.Nz(); ++nz)
2479         P(nx,ny,nz,0) += 0.5*(square(u(nx,ny,nz,0)) +
2480                               square(u(nx,ny,nz,1)) +
2481                               square(u(nx,ny,nz,2)));
2482
2483   u.realfft_xz();
2484   P.realfft_xz();
2485   u.chebyfft_y();
2486   P.chebyfft_y();
2487 }
```

Here is the call graph for this function:

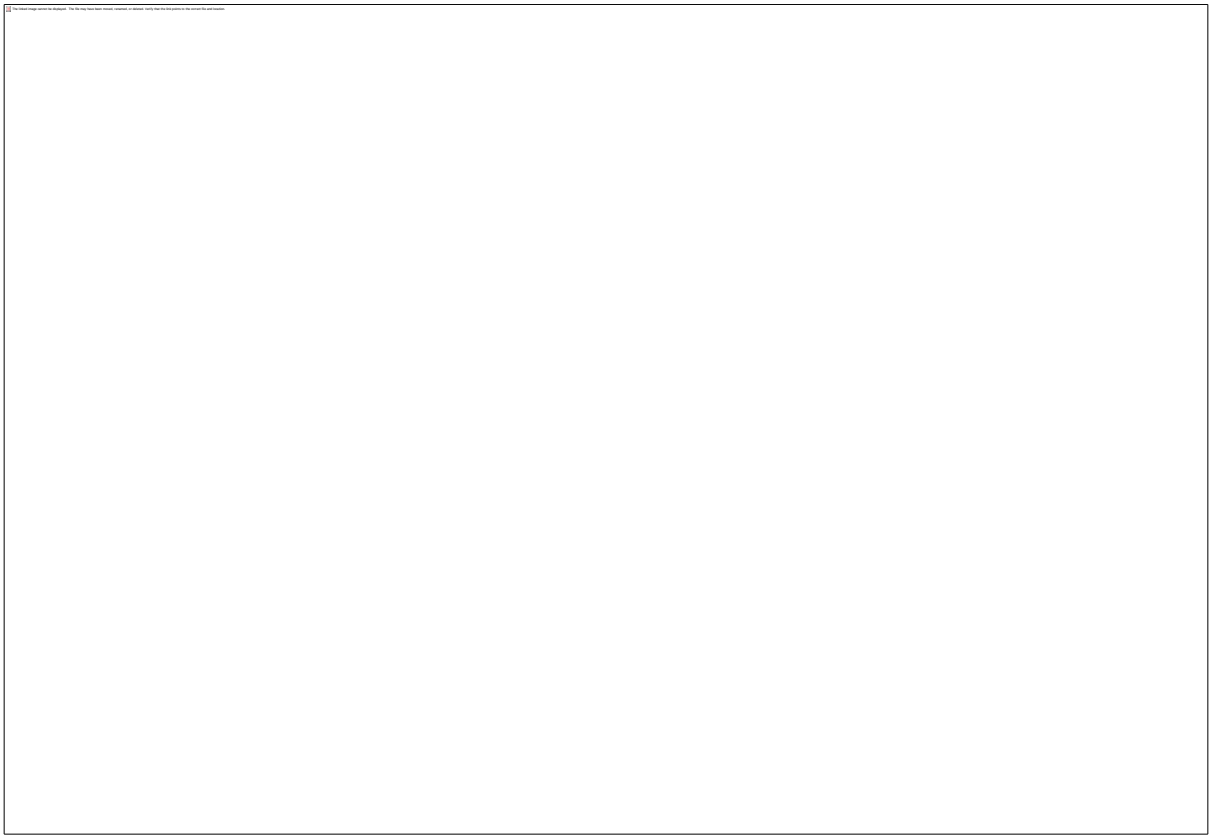
1.13.3.2 Real bcDist (const FlowField & *f*, const FlowField & *g*, bool *normalize*)

References bcDist2().

```
1755                                                    {
1756   return sqrt(bcDist2(f,g, normalize));
```

```
1757 }
```

Here is the call graph for this function:



1.13.3.3 Real bcDist2 (const FlowField & *f*, const FlowField & *g*, bool *normalize*)

References FlowField::a(), abs2(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), diff(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), Spectral, FlowField::xzstate(), and FlowField::ystate().

```
1724                                                    {
1725   assert(f.congruent(g));
1726   assert(f.xzstate() == Spectral && g.xzstate() == Spectral);
1727   //assert(f.ystate() == Spectral && g.ystate() == Spectral);
1728
1729   Real bc2 = 0.0;
1730   int Nd = f.Nd();
1731   int Mx = f.Mx();
1732   int My = f.My();
1733   int Mz = f.Mz();
1734   ComplexChebyCoeff diff(My, f.a(), f.b(), f.ystate());
```

```
1735
1736   for (int i=0; i<Nd; ++i)
1737     for (int mx=0; mx<Mx; ++mx) {
1738       for (int mz=0; mz<Mz; ++mz) {
1739         for (int my=0; my<My; ++my)
1740           diff.set(my, f.cmplx(mx,my,mz,i)-g.cmplx(mx,my,mz,i));
1741         bc2 += abs2(diff.eval_a());
1742         bc2 += abs2(diff.eval_b());
1743       }
1744     }
1745   if (!normalize)
1746     bc2 *= f.Lx()*f.Lz();
1747   return bc2;
1748 }
```

Here is the call graph for this function:

1.13.3.4 Real bcNorm (const FlowField & *f*, bool *normalize*)

References bcNorm2().

```
1751                                        {
1752   return sqrt(bcNorm2(f, normalize));
1753 }
```

Here is the call graph for this function:

1.13.3.5 Real bcNorm2 (const FlowField & *f*, bool *normalize*)

References FlowField::a(), abs2(), FlowField::b(), FlowField::cmplx(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Ny(), Spectral, FlowField::xzstate(), and FlowField::ystate().

```
1703                                        {
1704   assert(f.xzstate() == Spectral);
1705   Real bc2 = 0.0;
1706   int Mx = f.Mx();
1707   int My = f.My();
1708   int Mz = f.Mz();
1709   ComplexChebyCoeff prof(f.Ny(), f.a(), f.b(), f.ystate());
1710   for (int i=0; i<f.Nd(); ++i)
1711     for (int mx=0; mx<Mx; ++mx) {
1712       for (int mz=0; mz<Mz; ++mz) {
1713         for (int my=0; my<My; ++my)
1714           prof.set(my, f.cmplx(mx,my,mz,i));
1715         bc2 += abs2(prof.eval_a());
1716         bc2 += abs2(prof.eval_b());
1717       }
1718     }
1719   if (!normalize)
1720     bc2 *= f.Lx()*f.Lz();
1721   return bc2;
1722 }
```

Here is the call graph for this function:

1.13.3.6 void chebyProfile (ComplexChebyCoeff & *u*, ComplexChebyCoeff & *v*, ComplexChebyCoeff & *w*, int *un*, int *vn*, int *kx*, int *kz*, Real *Lx*, Real *Lz*, Real *decay*)

References ComplexChebyCoeff::a(), abs(), ComplexChebyCoeff::b(), chebyUprofile(), chebyVprofile(), diff(), div(), divNorm(), EPSILON, ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), I(), ComplexChebyCoeff::im, L2Norm(), ComplexChebyCoeff::length(), ComplexChebyCoeff::numModes(), pi, ComplexChebyCoeff::setState(), ComplexChebyCoeff::setToZero(), ChebyCoeff::setToZero(), and Spectral.

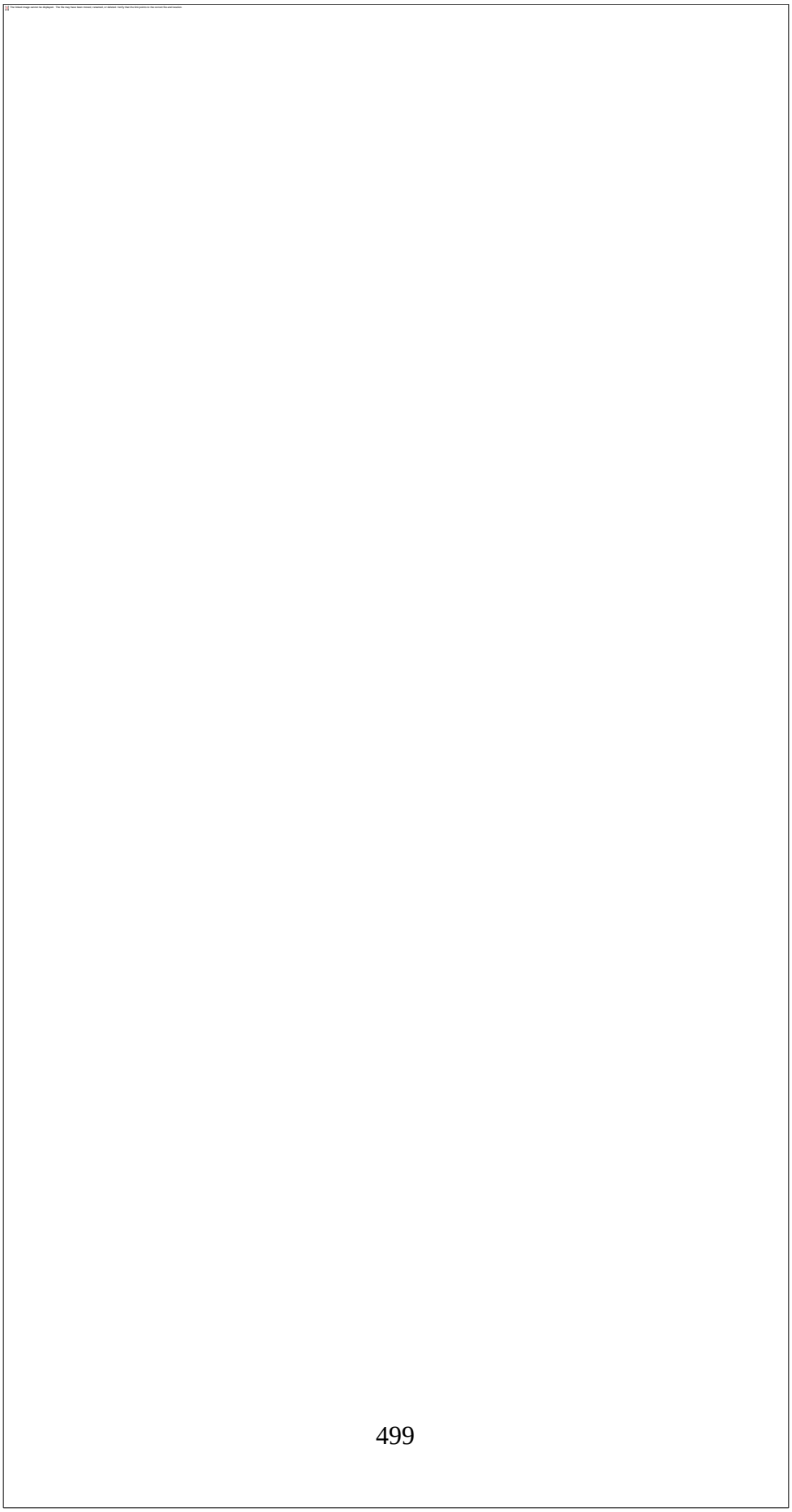
```
2333
{
2334
2335   int N = u.length();
2336   ChebyTransform trans(N);
2337   u.setState(Spectral);
2338   v.setState(Spectral);
2339   w.setState(Spectral);
```

```
2340  //u.setToZero();
2341  //v.setToZero();
2342  //w.setToZero();
2343  if (kx == 0 && kz == 0) {
2344    chebyUprofile(u, un, decay);
2345    chebyUprofile(w, vn, decay); // yes, vn
2346    u.im.setToZero();
2347    w.im.setToZero();
2348    v.setToZero();
2349  }
2350  else {
2351    // Other kx,kz cases are based on a random v(y).
2352
2353    chebyVprofile(v, vn, decay);
2354    ComplexChebyCoeff vy = diff(v);
2355
2356    if (kx == 0) {
2357      chebyUprofile(u, un, decay);
2358      u.im.setToZero();
2359      w = vy;
2360      w *= -Lz/((2*pi*kz)*I);
2361    }
2362    else if (kz == 0) {
2363      chebyUprofile(w, un, decay); // yes, un
2364      w.im.setToZero();
2365      u = vy;
2366      u *= -Lx/((2*pi*kx)*I);
2367    }
2368    else {
2369
2370      ComplexChebyCoeff v0(v);
2371      ComplexChebyCoeff v1(v);
2372      chebyVprofile(v0, vn, decay);
2373      chebyVprofile(v1, vn, decay);
2374
2375      ComplexChebyCoeff v0y = diff(v0);
2376      ComplexChebyCoeff v1y = diff(v1);
2377
2378      // Finally, the general case, where kx, kz != 0 and u,v,w are nonzero
2379      // Set a random u(y)
2380      ComplexChebyCoeff u0(v.numModes(), v.a(), v.b(), Spectral);
2381      ComplexChebyCoeff w0(v.numModes(), v.a(), v.b(), Spectral);
2382      ComplexChebyCoeff u1(v.numModes(), v.a(), v.b(), Spectral);
2383      ComplexChebyCoeff w1(v.numModes(), v.a(), v.b(), Spectral);
2384
2385      chebyUprofile(u0, un, decay);
```

```
2386
2387      // Calculate w0 from div u0 == u0x + v0y + w0z == 0.
2388      ComplexChebyCoeff u0x(u0);
2389      u0x *= (2*pi*kx/Lx)*I;
2390      w0 = v0y;
2391      w0 += u0x;
2392      w0 *= -Lz/((2*pi*kz)*I);       // Set w = -Lz/(2*pi*I*kz) * (ux + vy);
2393
2394
2395      //randomUprofile(w1, mag, decay);
2396
2397      // Calculate u0 from div u0 == u0x + v0y + w0z == 0.
2398      ComplexChebyCoeff w1z(w1);
2399      w1z *= (2*pi*kz/Lz)*I;
2400      u1 = v1y;
2401      u1 += w1z;
2402      u1 *= -Lx/((2*pi*kx)*I);       // Set w = -Lz/(2*pi*I*kz) * (ux + vy);
2403
2404      u = u0;
2405      v = v0;
2406      w = w0;
2407      u += u1;
2408      v += v1;
2409      w += w1;
2410    }
2411  }
2412
2413  // Check divergence
2414  ComplexChebyCoeff ux(u);
2415  ux *= (2*pi*kx/Lx)*I;
2416  ComplexChebyCoeff wz(w);
2417  wz *= (2*pi*kz/Lz)*I;
2418  ComplexChebyCoeff vy = diff(v);
2419
2420  ComplexChebyCoeff div(ux);
2421  div += vy;
2422  div += wz;
2423
2424  Real divNorm = L2Norm(div);
2425  Real ubcNorm = abs(u.eval_a()) + abs(u.eval_b());
2426  Real vbcNorm = abs(v.eval_a()) + abs(v.eval_b());
2427  Real wbcNorm = abs(w.eval_a()) + abs(w.eval_b());
2428  assert(divNorm < EPSILON);
2429  assert(ubcNorm < EPSILON);
2430  assert(vbcNorm < EPSILON);
2431  assert(wbcNorm < EPSILON);
```

```
2432   // supress unused-variable compiler warnings...
2433   wbcNorm += divNorm + ubcNorm + vbcNorm;
2434
2435 }
```

Here is the call graph for this function:


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.

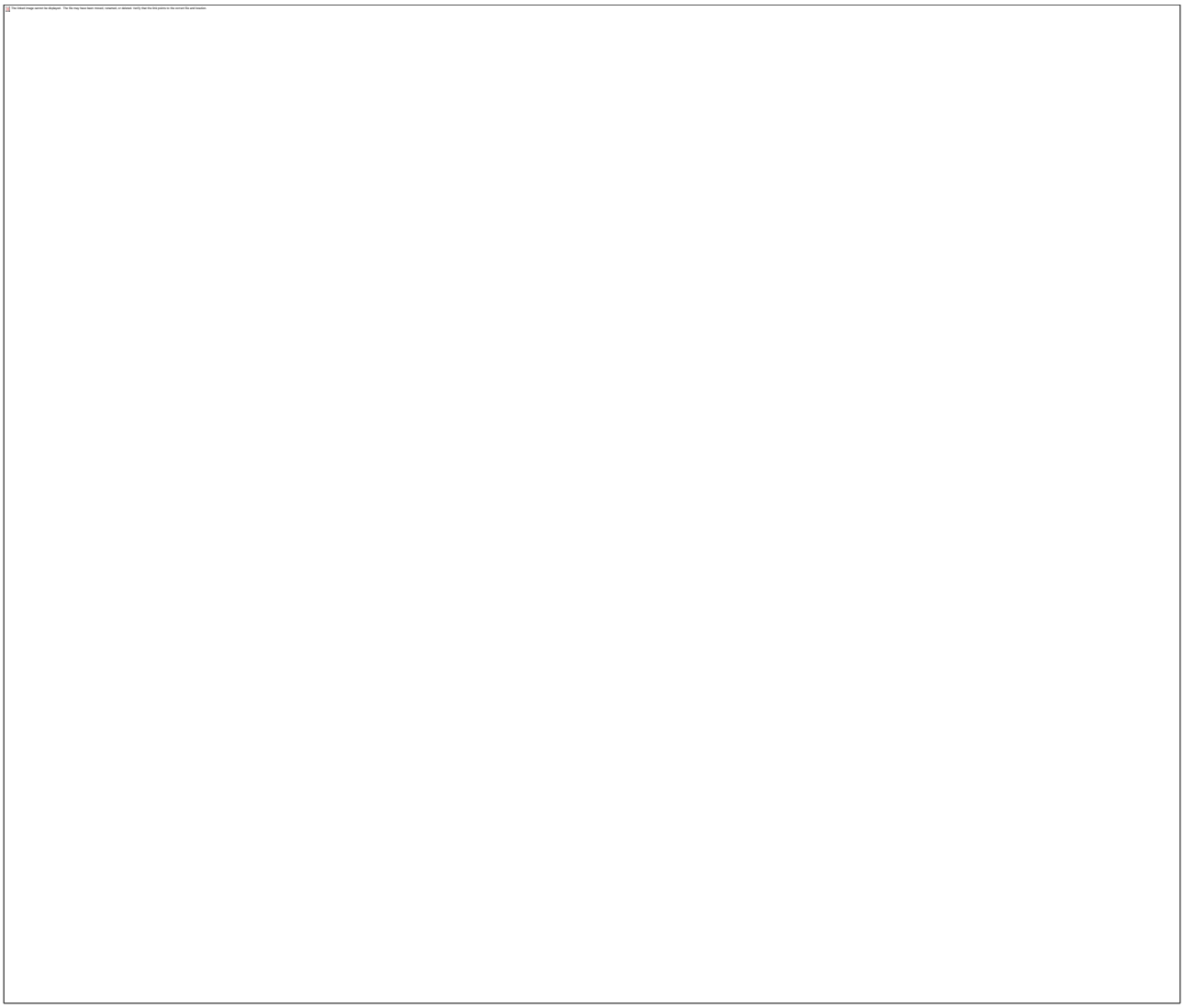
1.13.3.7 void chebyUprofile (ComplexChebyCoeff & *u*, int *n*, Real *decay*)

References abs(), EPSILON, ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), I(), ComplexChebyCoeff::length(), pi, pow(), randomReal(), ComplexChebyCoeff::set(), ComplexChebyCoeff::setState(), ComplexChebyCoeff::setToZero(), Spectral, and ComplexChebyCoeff::sub().

Referenced by chebyProfile().

```
2146                                                {
2147   // Set a random u(y)
2148   int N = u.length();
2149   u.setToZero();
2150   u.setState(Spectral);
2151   Real theta = pi*randomReal();
2152   u.set(n, (cos(theta) + I*sin(theta))*pow(decay,n));
2153
2154   ChebyTransform trans(N);
2155
2156   // Adjust u(y) so that u(+-1) == 0
2157   Complex u0 = (u.eval_b() + u.eval_a())/2.0;  // 2.0 is correct for genl a,b
2158   Complex u1 = (u.eval_b() - u.eval_a())/2.0;  // 2.0 is correct for genl a,b
2159   u.sub(0, u0);
2160   u.sub(1, u1);
2161   //cout << "random u(a) == " << u.eval_a() << endl;
2162   //cout << "random u(b) == " << u.eval_b() << endl;
2163   assert(abs(u.eval_b()) < EPSILON);
2164   assert(abs(u.eval_a()) < EPSILON);
2165 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.13.3.8 void chebyVprofile (ComplexChebyCoeff & *v*, int *n*, Real *decay*)

References diff(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), I(), pi, pow(), randomReal(), ComplexChebyCoeff::set(), ComplexChebyCoeff::setState(), ComplexChebyCoeff::setToZero(), Spectral, and ComplexChebyCoeff::sub().

Referenced by chebyProfile().

```
2167                                        {
2168   v.setToZero();
2169   v.setState(Spectral);
```

```
2170  Real theta = pi*randomReal();
2171  v.set(n, (cos(theta) + I*sin(theta))*pow(decay,n));
2172
2173  // Adjust v so that v(+-1) == v'(+/-1) == 0, by subtracting off
2174  // s0 T0(x) + s1 T1(x) + s2 T2(x) + s3 T3(x), with s's chosen to
2175  // have same BCs as v.
2176  ComplexChebyCoeff vy = diff(v);
2177
2178  Complex a = v.eval_a();
2179  Complex b = v.eval_b();
2180  Complex c = vy.eval_a();
2181  Complex d = vy.eval_b();
2182
2183  // The numercial coeffs are inverse of the matrix (values found with Maple)
2184  // T0(-1)  T1(-1)  T2(-1)  T3(-1)     s0      a
2185  // T0(1)   T1(1)   T2(1)   T3(1)      s1      b
2186  // T0'(-1) T1'(-1) T2'(-1) T3'(-1)    s2  ==  c
2187  // T0'(1)  T1'(1)  T2'(1)  T3'(1)     s3      d
2188
2189  // The above matrix is
2190  // 1  -1   1  -1
2191  // 1   1   1   1
2192  // 0   1  -4   9
2193  // 0   1   4   9
2194
2195  Complex s0 = 0.5*(a + b) + 0.125*(c - d);
2196  Complex s1 = 0.5625*(b - a) - 0.0625*(c + d);
2197  Complex s2 = 0.125*(d - c);
2198  Complex s3 = 0.0625*(a - b + c + d);
2199
2200  //ComplexChebyCoeff adj(v.numModes(), v.a(), v.b(), Spectral);
2201  //adj.set(0, s0);
2202  //adj.set(1, s1);
2203  //adj.set(2, s2);
2204  //adj.set(3, s3);
2205  //ComplexChebyCoeff adjy = diff(adj);
2206
2207  // Subtract off the coeffs
2208  v.sub(0, s0);
2209  v.sub(1, s1);
2210  v.sub(2, s2);
2211  v.sub(3, s3);
2212
2213  //diff(v,vy);
2214
2215  //cout << "random v(a)  == " << v.eval_a() << endl;
```

```
2216   //cout << "random v(b)  == " << v.eval_b()  << endl;
2217   //cout << "random v'(a) == " << vy.eval_a() << endl;
2218   //cout << "random v'(b) == " << vy.eval_b() << endl;
2219
2220 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.13.3.9 Real divDist (const FlowField & *f*, const FlowField & *g*, bool *normalize*)

References divDist2().

```
1790                                                    {
1791   return sqrt(divDist2(f,g,normalize));
1792 }
```

Here is the call graph for this function:

1.13.3.10 Real divDist2 (const FlowField & *f*, const FlowField & *g*, bool *normalize*)

References FlowField::congruent(), divDist2(), FlowField::Mx(), FlowField::Mz(), FlowField::Nd(), FlowField::profile(), Spectral, FlowField::xzstate(), and FlowField::ystate().

```
1776                                                    {
1777   assert(f.xzstate() == Spectral && f.ystate() == Spectral);
1778   assert(f.congruent(g));
1779   assert(f.Nd() == 3);
1780
1781   Real div2=0.0;
1782   int Mx = f.Mx();
1783   int Mz = f.Mz();
1784   for (int mx=0; mx<Mx; ++mx)
1785     for (int mz=0; mz<Mz; ++mz)
1786       div2 += divDist2(f.profile(mx,mz), g.profile(mx,mz), normalize);
1787   return div2;
1788 }
```

Here is the call graph for this function:

1.13.3.11 Real divNorm (const FlowField & *f*, bool *normalize*)

References divNorm2().

```
1759                                        {
1760   return sqrt(divNorm2(f,normalize));
1761 }
```

Here is the call graph for this function:

1.13.3.12 Real divNorm2 (const FlowField & *f*, bool *normalize*)

References divNorm2(), FlowField::Mx(), FlowField::Mz(), FlowField::Nd(), FlowField::profile(), Spectral, FlowField::xzstate(), and FlowField::ystate().

```
1763                                        {
1764   assert(f.xzstate() == Spectral && f.ystate() == Spectral);
1765   assert(f.Nd() == 3);
1766
1767   Real div2=0.0;
1768   int Mx = f.Mx();
1769   int Mz = f.Mz();
1770   BasisFunc prof;
1771   for (int mx=0; mx<Mx; ++mx)
1772     for (int mz=0; mz<Mz; ++mz)
1773       div2 += divNorm2(f.profile(mx,mz), normalize);
1774   return div2;
1775 }
```

Here is the call graph for this function:

1.13.3.13 Real L1Dist (const FlowField & *f*, const FlowField & *g*, bool *normalize*)

References FlowField::congruent(), L1Norm(), FlowField::Lx(), FlowField::Lz(), ChebyCoeff::makePhysical(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), FlowField::profile(), Re(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1821                                                {
1822   assert(f.congruent(g));
1823   fieldstate fxstate = f.xzstate();
1824   fieldstate gxstate = g.xzstate();
1825   fieldstate fystate = f.ystate();
1826   fieldstate gystate = g.ystate();
1827
1828   ((FlowField&) f).makePhysical_xz();
1829   ((FlowField&) g).makePhysical_xz();
1830
1831   Real sum = 0.0;
1832
1833   ChebyCoeff prof;
1834   ChebyTransform trans(f.Ny());
1835   for (int i=0; i<f.vectorDim(); ++i)
1836     for (int nx=0; nx<f.Nx(); ++nx)
1837       for (int nz=0; nz<f.Nz(); ++nz) {
1838         prof = Re(f.profile(nx,nz,i));
1839         prof -= Re(g.profile(nx,nz,i));
1840         prof.makePhysical(trans);
1841         sum += L1Norm(prof, normalize);
1842       }
1843
1844   sum /= f.Nx()*f.Nz();
1845
1846   if (!normalize)
1847     sum *= f.Lx()*f.Lz();
1848
1849   ((FlowField&) f).makeState(fxstate, fystate);
1850   ((FlowField&) g).makeState(gxstate, gystate);
1851
1852   return sum;
1853 }
```

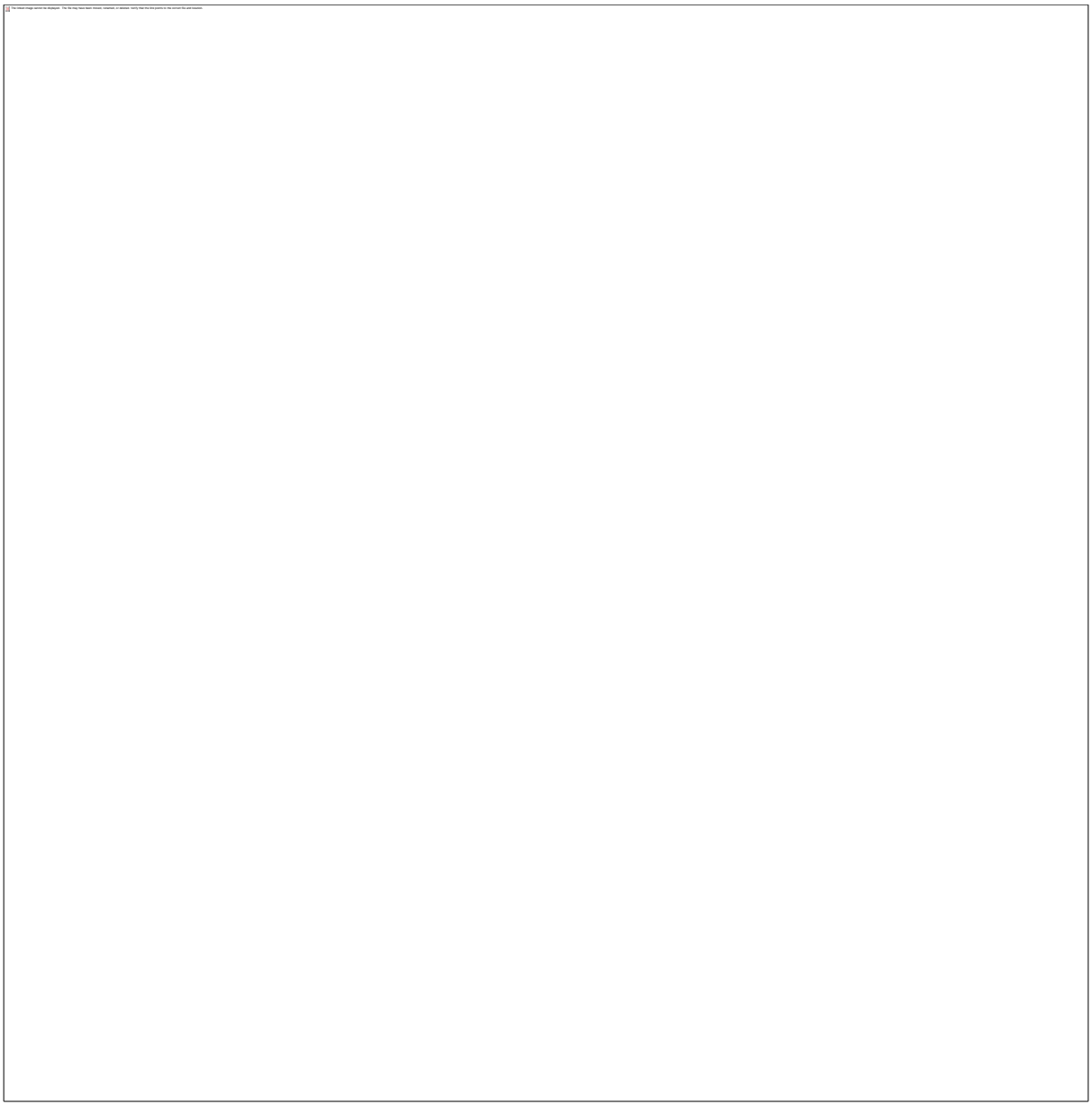
Here is the call graph for this function:

1.13.3.14 Real L1Norm (const FlowField & *u*, bool *normalize*)

References L1Norm(), FlowField::Lx(), FlowField::Lz(), ChebyCoeff::makePhysical(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), FlowField::profile(), Re(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1794                                    {
1795   fieldstate xstate = u.xzstate();
1796   fieldstate ystate = u.ystate();
1797   ((FlowField&) u).makePhysical_xz();
1798
1799   Real sum = 0.0;
1800
1801   ChebyCoeff prof;
1802   ChebyTransform trans(u.Ny());
1803   for (int i=0; i<u.vectorDim(); ++i)
1804     for (int nx=0; nx<u.Nx(); ++nx)
1805       for (int nz=0; nz<u.Nz(); ++nz) {
1806         prof = Re(u.profile(nx,nz,i));
1807         prof.makePhysical(trans);
1808         sum += L1Norm(prof);
1809       }
1810
1811   sum /= u.Nx()*u.Nz();
1812
1813   if (!normalize)
1814     sum *= u.Lx()*u.Lz();
1815
1816   ((FlowField&) u).makeState(xstate, ystate);
1817
1818   return sum;
1819 }
```

Here is the call graph for this function:



1.13.3.15 Real L2Dist (const FlowField & *u*, const FlowField & *v*, int *kxmax*, int *kzmax*, bool *normalize*)

References L2Dist2().

```
2010                                                                    {
2011   return sqrt(L2Dist2(u, v, kxmax, kzmax, normalize));
2012 }
```

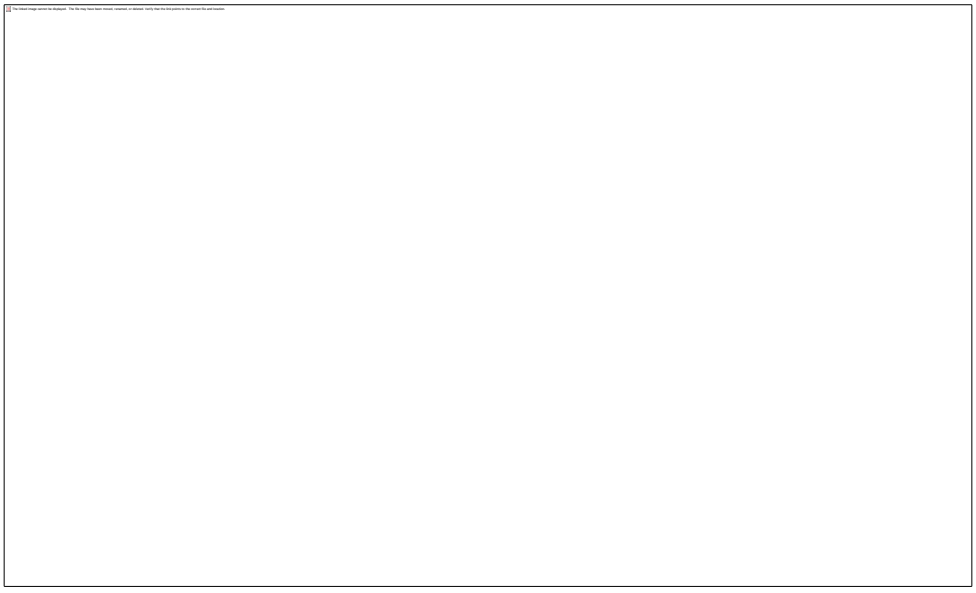
Here is the call graph for this function:

1.13.3.16 Real L2Dist (const FlowField & *u*, const FlowField & *v*, bool *normalize*)

References L2Dist2().

```
1906                                                {
1907   return sqrt(L2Dist2(u,v,normalize));
1908 }
```

Here is the call graph for this function:



1.13.3.17 Real L2Dist2 (const FlowField & *u*, const FlowField & *v*, int *kxmax*, int *kzmax*, bool *normalize*)

References FlowField::a(), abs(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), FlowField::kx(), FlowField::kz(), L2Norm2(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::Mz(), FlowField::Ny(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
2014                                                    {
2015   Real sum = 0.0;
2016   assert(u.congruent(v));
2017   assert(u.xzstate()==Spectral && u.ystate()==Spectral);
2018   ComplexChebyCoeff u_v(u.Ny(), u.a(), u.b(), Spectral);
2019   for (int i=0; i<u.vectorDim(); ++i)
2020     for (int mx=0; mx<u.Mx(); ++mx) {
2021       if (abs(u.kx(mx)) > kxmax)
2022         continue;
2023       int cz = 1;
2024       for (int mz=0; mz<u.Mz(); ++mz) {
2025         if (abs(u.kz(mz)) > kzmax)
2026           continue;
2027         for (int ny=0; ny<u.Ny(); ++ny)
2028           u_v.set(ny, u.cmplx(mx,ny,mz,i)-v.cmplx(mx,ny,mz,i));
2029         sum += cz*L2Norm2(u_v, normalize);
2030         cz = 2;
2031       }
2032     }
2033   if (!normalize)
2034     sum *= u.Lx()*u.Lz();
2035   return sum;
2036 }
```

Here is the call graph for this function:

1.13.3.18 Real L2Dist2 (const FlowField & *u*, const FlowField & *v*, bool *normalize*)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), L2Norm2(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::Mz(), FlowField::Ny(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1910                                                {
1911   Real sum = 0.0;
1912   assert(u.congruent(v));
1913   assert(u.xzstate()==Spectral && u.ystate()==Spectral);
1914   ComplexChebyCoeff u_v(u.Ny(), u.a(), u.b(), Spectral);
1915   for (int i=0; i<u.vectorDim(); ++i)
1916     for (int mx=0; mx<u.Mx(); ++mx) {
1917       int cz = 1;
1918       for (int mz=0; mz<u.Mz(); ++mz) {
1919         for (int ny=0; ny<u.Ny(); ++ny)
1920           u_v.set(ny, u.cmplx(mx,ny,mz,i)-v.cmplx(mx,ny,mz,i));
1921         sum += cz*L2Norm2(u_v, normalize);
1922         cz = 2;
1923       }
1924     }
1925   if (!normalize)
1926     sum *= u.Lx()*u.Lz();
1927   return sum;
1928 }
```

Here is the call graph for this function:

1.13.3.19 Complex L2InnerProduct (const FlowField & *u*, const BasisFunc & *phi*, bool *normalize*)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), BasisFunc::kx(), BasisFunc::kz(), L2InnerProduct(), FlowField::Lx(), FlowField::Lz(), FlowField::mx(), FlowField::mz(), FlowField::Nd(), FlowField::Ny(), Spectral, FlowField::xzstate(), and FlowField::ystate().

```
2039                                {
2040   assert(u.xzstate()==Spectral && u.ystate()==Spectral);
2041   assert(u.congruent(phi));
2042   int kx = phi.kx();
2043   int kz = phi.kz();
2044   int mx = u.mx(kx);
2045   int mz = u.mz(kz);
2046   int Ny = u.Ny();
2047   BasisFunc profile(Ny, kx, kz, u.Lx(), u.Lz(), u.a(), u.b(), Spectral);
2048
2049
2050   for (int i=0; i<u.Nd(); ++i)
2051     for (int ny=0; ny<Ny; ++ny)
2052       profile[i].set(ny, u.cmplx(mx,ny,mz, i));
2053
2054   return L2InnerProduct(profile, phi, normalize);
2055 }
```

Here is the call graph for this function:

1.13.3.20 Real L2InnerProduct (const FlowField & *f*, const FlowField & *g*, bool *normalize*)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), L2InnerProduct(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::Mz(), FlowField::Ny(), Re(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1954                                                    {
1955   assert(f.ystate() == Spectral);
1956   assert(g.ystate() == Spectral);
1957   assert(f.xzstate() == Spectral);
1958   assert(g.xzstate() == Spectral);
1959   assert(f.congruent(g));
1960
1961   Real sum = 0.0;
1962   ComplexChebyCoeff fprof(f.Ny(), f.a(), f.b(), Spectral);
1963   ComplexChebyCoeff gprof(g.Ny(), g.a(), g.b(), Spectral);
1964   for (int i=0; i<f.vectorDim(); ++i)
1965     for (int mx=0; mx<f.Mx(); ++mx) {
1966       Real cz = 1.0; // cz = 2 for kz>0 to take account of kz<0 ghost modes
1967       for (int mz=0; mz<f.Mz(); ++mz) {
1968         for (int ny=0; ny<f.Ny(); ++ny) {
1969           fprof.set(ny, f.cmplx(mx,ny,mz,i));
1970           gprof.set(ny, g.cmplx(mx,ny,mz,i));
1971         }
1972         sum += cz*Re(L2InnerProduct(fprof, gprof, normalize));
1973         cz = 2.0;
1974       }
1975     }
1976   if (!normalize)
1977     sum *= f.Lx()*f.Lz();
1978   return sum;
1979 }
```

Here is the call graph for this function:

1.13.3.21 Real L2Norm (const FlowField & *u*, int *kxmax*, int *kzmax*, bool *normalize*)

References L2Norm2().

```
2007                                                {
```

```
2008   return sqrt(L2Norm2(u, kxmax, kzmax, normalize));
2009 }
```

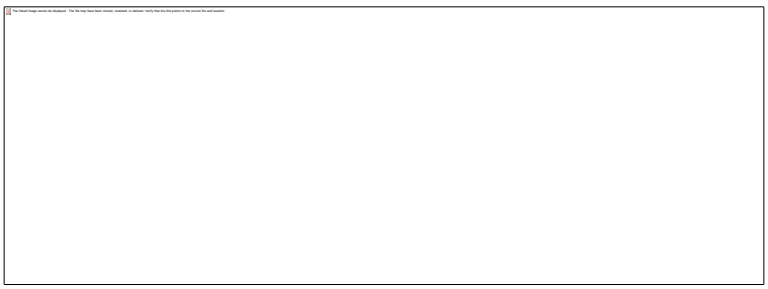
Here is the call graph for this function:

1.13.3.22 Real L2Norm (const FlowField & *u*, bool *normalize*)

References L2Norm2().

```
1930                                        {
1931   return sqrt(L2Norm2(u, normalize));
1932 }
```

Here is the call graph for this function:



1.13.3.23 Real L2Norm2 (const FlowField & *u*, int *kxmax*, int *kzmax*, bool *normalize*)

References FlowField::a(), abs(), FlowField::b(), FlowField::cmplx(), FlowField::kx(), FlowField::kz(), L2Norm2(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::Mz(), FlowField::Ny(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1981                                                {
1982   assert(u.ystate()==Spectral);
1983   assert(u.xzstate()==Spectral);
1984   Real sum = 0.0;
1985
```

```
1986   // EFFICIENCY: would be better with smarter looping, but prob not worth trouble.
1987   ComplexChebyCoeff prof(u.Ny(), u.a(), u.b(), Spectral);
1988   for (int i=0; i<u.vectorDim(); ++i)
1989     for (int mx=0; mx<u.Mx(); ++mx) {
1990       if (abs(u.kx(mx)) > kxmax)
1991         continue;
1992       Real cz = 1.0; // cz = 2 for kz>0 to take account of kz<0 ghost modes
1993       for (int mz=0; mz<u.Mz(); ++mz) {
1994         if (abs(u.kz(mz)) > kzmax)
1995           continue;
1996         for (int ny=0; ny<u.Ny(); ++ny)
1997           prof.set(ny, u.cmplx(mx,ny,mz,i));
1998         sum += cz*L2Norm2(prof, normalize);
1999         cz = 2.0;
2000       }
2001     }
2002   if (!normalize)
2003     sum *= u.Lx()*u.Lz();
2004   return sum;
2005 }
```

Here is the call graph for this function:

1.13.3.24 Real L2Norm2 (const FlowField & *u*, bool *normalize*)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), L2Norm2(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::Mz(), FlowField::Ny(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1934                                        {
1935   assert(u.ystate() == Spectral);
1936   assert(u.xzstate()==Spectral);
1937   Real sum = 0.0;
1938   ComplexChebyCoeff prof(u.Ny(), u.a(), u.b(), Spectral);
1939   for (int i=0; i<u.vectorDim(); ++i)
1940     for (int mx=0; mx<u.Mx(); ++mx) {
1941       Real cz = 1.0; // cz = 2 for kz>0 to take account of kz<0 ghost modes
1942       for (int mz=0; mz<u.Mz(); ++mz) {
1943         for (int ny=0; ny<u.Ny(); ++ny)
1944           prof.set(ny, u.cmplx(mx,ny,mz,i));
1945         sum += cz*L2Norm2(prof, normalize);
1946         cz = 2.0;
1947       }
1948     }
1949   if (!normalize)
1950     sum *= u.Lx()*u.Lz();
1951   return sum;
1952 }
```

Here is the call graph for this function:

1.13.3.25 Real LinfDist (const FlowField & *f*, const FlowField & *g*)

References FlowField::congruent(), Greater(), LinfNorm(), ChebyCoeff::makePhysical(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), FlowField::profile(), Re(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1877                                    {
1878   assert(f.congruent(g));
1879   fieldstate fxstate = f.xzstate();
1880   fieldstate gxstate = g.xzstate();
1881   fieldstate fystate = f.ystate();
```

```
1882   fieldstate gystate = g.ystate();
1883
1884   ((FlowField&) f).makePhysical_xz();
1885   ((FlowField&) g).makePhysical_xz();
1886
1887   Real rtn = 0.0;
1888
1889   ChebyCoeff prof;
1890   ChebyTransform trans(f.Ny());
1891   for (int i=0; i<f.vectorDim(); ++i)
1892     for (int nx=0; nx<f.Nx(); ++nx)
1893       for (int nz=0; nz<f.Nz(); ++nz) {
1894         prof = Re(f.profile(nx,nz,i));
1895         prof -= Re(g.profile(nx,nz,i));
1896         prof.makePhysical();
1897         rtn = Greater(rtn, LinfNorm(prof));
1898       }
1899
1900   ((FlowField&) f).makeState(fxstate, fystate);
1901   ((FlowField&) g).makeState(gxstate, gystate);
1902
1903   return rtn;
1904 }
```

Here is the call graph for this function:

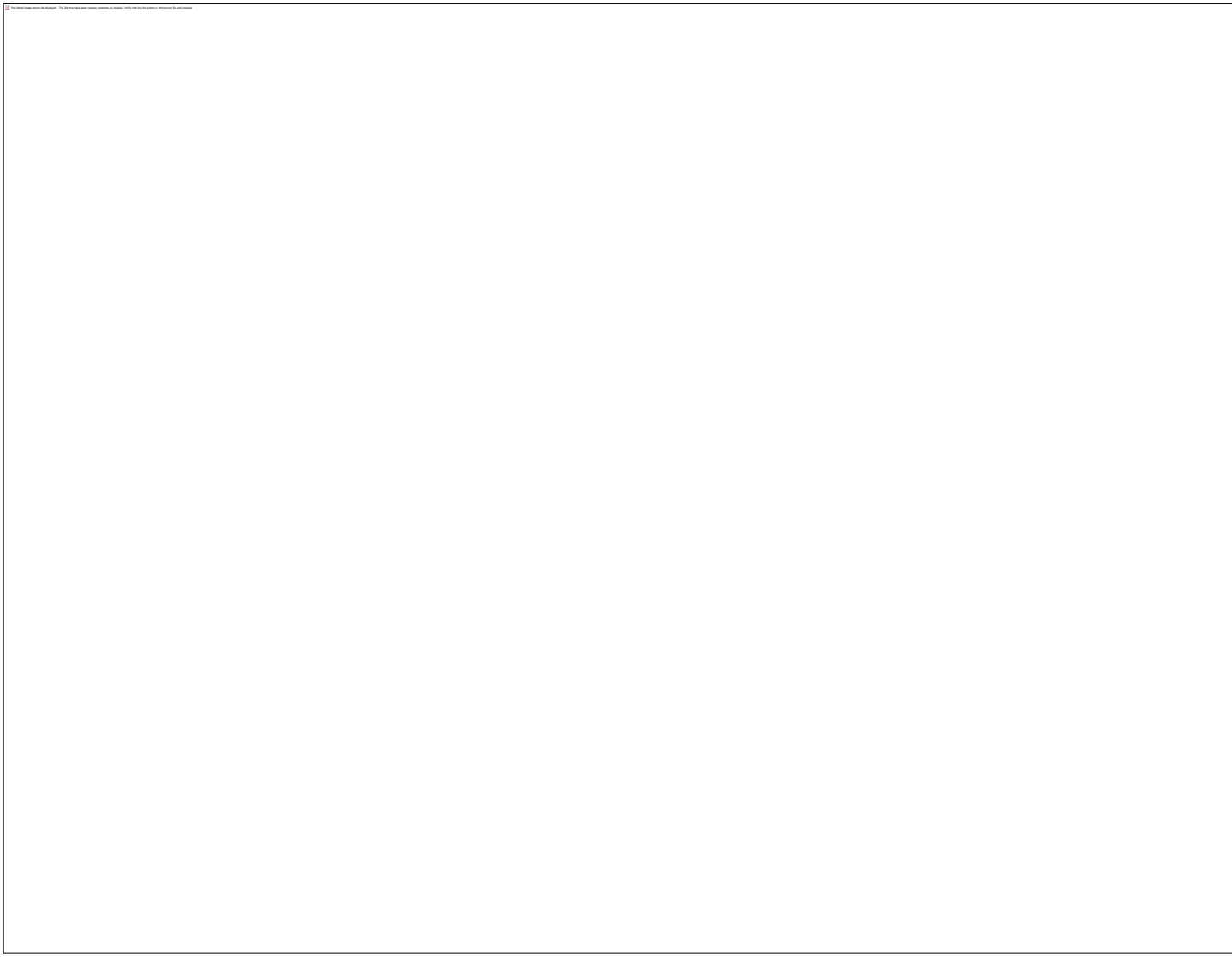
1.13.3.26 Real LinfNorm (const FlowField & *u*)

References Greater(), LinfNorm(), ChebyCoeff::makePhysical(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), FlowField::profile(), Re(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1855                                {
1856   fieldstate xstate = u.xzstate();
1857   fieldstate ystate = u.ystate();
1858   ((FlowField&) u).makePhysical_xz();
1859
1860   Real rtn = 0.0;
1861
1862   ChebyCoeff prof;
1863   ChebyTransform trans(u.Ny());
1864   for (int i=0; i<u.vectorDim(); ++i)
1865     for (int nx=0; nx<u.Nx(); ++nx)
1866       for (int nz=0; nz<u.Nz(); ++nz) {
1867         prof = Re(u.profile(nx,nz,i));
1868         prof.makePhysical(trans);
1869         rtn = Greater(rtn, LinfNorm(prof));
```

```
1870      }
1871
1872   ((FlowField&) u).makeState(xstate, ystate);
1873
1874   return rtn;
1875 }
```

Here is the call graph for this function:



1.13.3.27 void randomProfile (ComplexChebyCoeff & *u*, ComplexChebyCoeff & *v*, ComplexChebyCoeff & *w*, int *kx*, int *kz*, Real *Lx*, Real *Lz*, Real *mag*, Real *decay*)

References ComplexChebyCoeff::a(), abs(), ComplexChebyCoeff::b(), diff(), div(), divNorm(), EPSILON, ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), I(), ComplexChebyCoeff::im, L2Norm(), ComplexChebyCoeff::length(), ComplexChebyCoeff::numModes(), pi, randomUprofile(), randomVprofile(), ComplexChebyCoeff::setState(), ComplexChebyCoeff::setToZero(), ChebyCoeff::setToZero(), and Spectral.

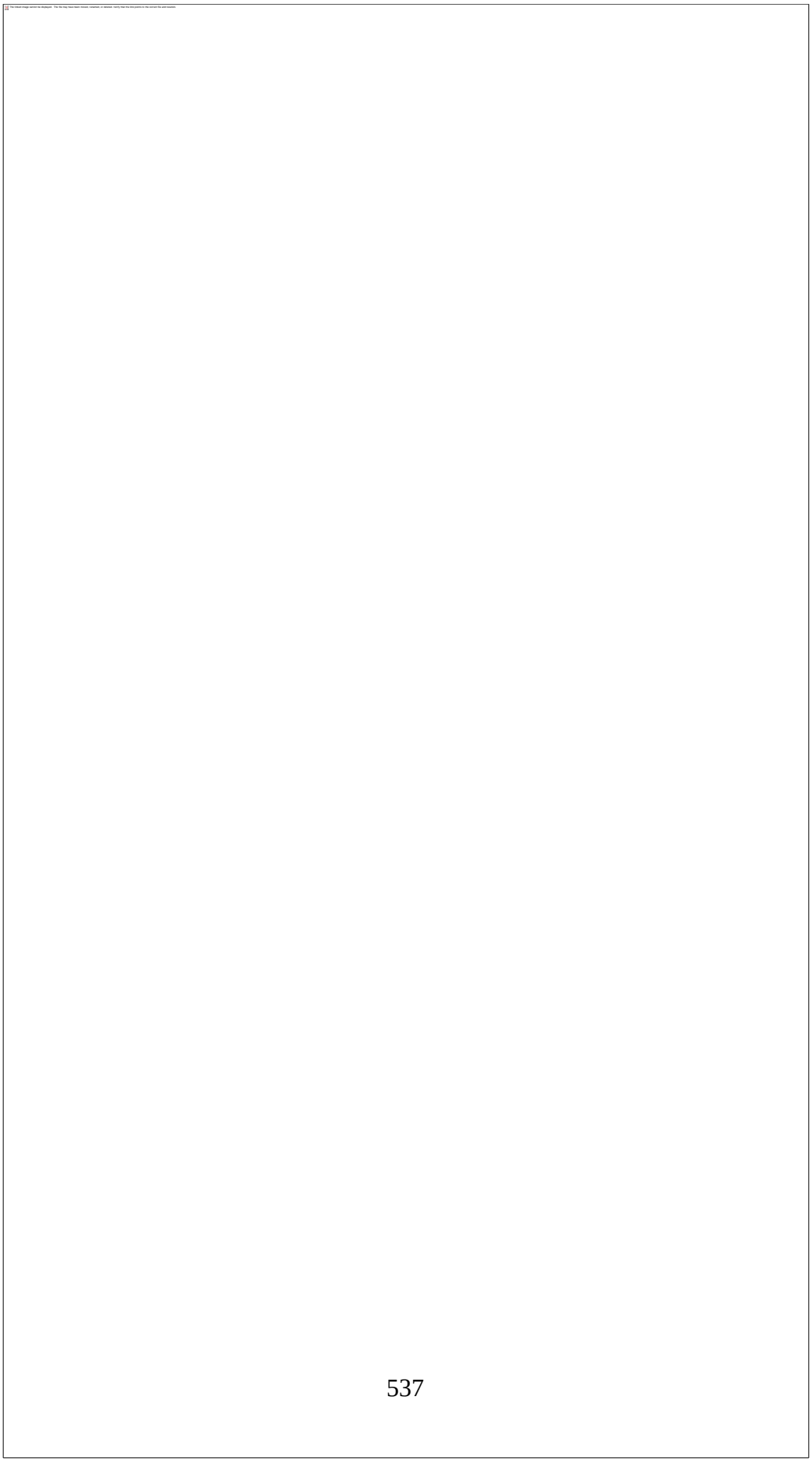
Referenced by FlowField::addPerturbation().

```
2223
{
2224
2225  int N = u.length();
2226  ChebyTransform trans(N);
2227  u.setState(Spectral);
2228  v.setState(Spectral);
2229  w.setState(Spectral);
2230  //u.setToZero();
2231  //v.setToZero();
2232  //w.setToZero();
2233  if (kx == 0 && kz == 0) {
2234    // Assign an odd perturbation to u, so as not to change mean(U).
2235    // Just set even modes to zero.
2236
2237    randomUprofile(w, mag, decay);
2238    w.im.setToZero();
2239
2240    randomUprofile(u, mag, decay);
2241    u.im.setToZero();
2242
2243    v.setToZero();
2244
2245  }
2246  else {
2247    // Other kx,kz cases are based on a random v(y).
2248
2249    randomVprofile(v, mag, decay);
2250    ComplexChebyCoeff vy = diff(v);
2251
2252    if (kx == 0) {
2253      randomUprofile(u, mag, decay);
2254      u.im.setToZero();
2255      w = vy;
2256      w *= -Lz/((2*pi*kz)*I);
2257    }
2258    else if (kz == 0) {
2259      randomUprofile(w, mag, decay);
2260      w.im.setToZero();
```

```
2261      u = vy;
2262      u *= -Lx/((2*pi*kx)*I);
2263    }
2264    else {
2265
2266      ComplexChebyCoeff v0(v);
2267      ComplexChebyCoeff v1(v);
2268      randomVprofile(v0, mag, decay);
2269      randomVprofile(v1, mag, decay);
2270
2271      ComplexChebyCoeff v0y = diff(v0);
2272      ComplexChebyCoeff v1y = diff(v1);
2273
2274      // Finally, the general case, where kx, kz != 0 and u,v,w are nonzero
2275      // Set a random u(y)
2276      ComplexChebyCoeff u0(v.numModes(), v.a(), v.b(), Spectral);
2277      ComplexChebyCoeff w0(v.numModes(), v.a(), v.b(), Spectral);
2278      ComplexChebyCoeff u1(v.numModes(), v.a(), v.b(), Spectral);
2279      ComplexChebyCoeff w1(v.numModes(), v.a(), v.b(), Spectral);
2280
2281      randomUprofile(u0, mag, decay);
2282
2283      // Calculate w0 from div u0 == u0x + v0y + w0z == 0.
2284      ComplexChebyCoeff u0x(u0);
2285      u0x *= (2*pi*kx/Lx)*I;
2286      w0 = v0y;
2287      w0 += u0x;
2288      w0 *= -Lz/((2*pi*kz)*I);       // Set w = -Lz/(2*pi*I*kz) * (ux + vy);
2289
2290
2291      //randomUprofile(w1, mag, decay);
2292
2293      // Calculate u0 from div u0 == u0x + v0y + w0z == 0.
2294      ComplexChebyCoeff w1z(w1);
2295      w1z *= (2*pi*kz/Lz)*I;
2296      u1 = v1y;
2297      u1 += w1z;
2298      u1 *= -Lx/((2*pi*kx)*I);       // Set w = -Lz/(2*pi*I*kz) * (ux + vy);
2299
2300      u = u0;
2301      v = v0;
2302      w = w0;
2303      u += u1;
2304      v += v1;
2305      w += w1;
2306    }
```

```
2307   }
2308
2309   // Check divergence
2310   ComplexChebyCoeff ux(u);
2311   ux *= (2*pi*kx/Lx)*I;
2312   ComplexChebyCoeff wz(w);
2313   wz *= (2*pi*kz/Lz)*I;
2314   ComplexChebyCoeff vy = diff(v);
2315
2316   ComplexChebyCoeff div(ux);
2317   div += vy;
2318   div += wz;
2319
2320   Real divNorm = L2Norm(div);
2321   Real ubcNorm = abs(u.eval_a()) + abs(u.eval_b());
2322   Real vbcNorm = abs(v.eval_a()) + abs(v.eval_b());
2323   Real wbcNorm = abs(w.eval_a()) + abs(w.eval_b());
2324   assert(divNorm < EPSILON);
2325   assert(ubcNorm < EPSILON);
2326   assert(vbcNorm < EPSILON);
2327   assert(wbcNorm < EPSILON);
2328   // supress unused-variable compiler warnings...
2329   wbcNorm += divNorm + ubcNorm + vbcNorm;
2330
2331 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

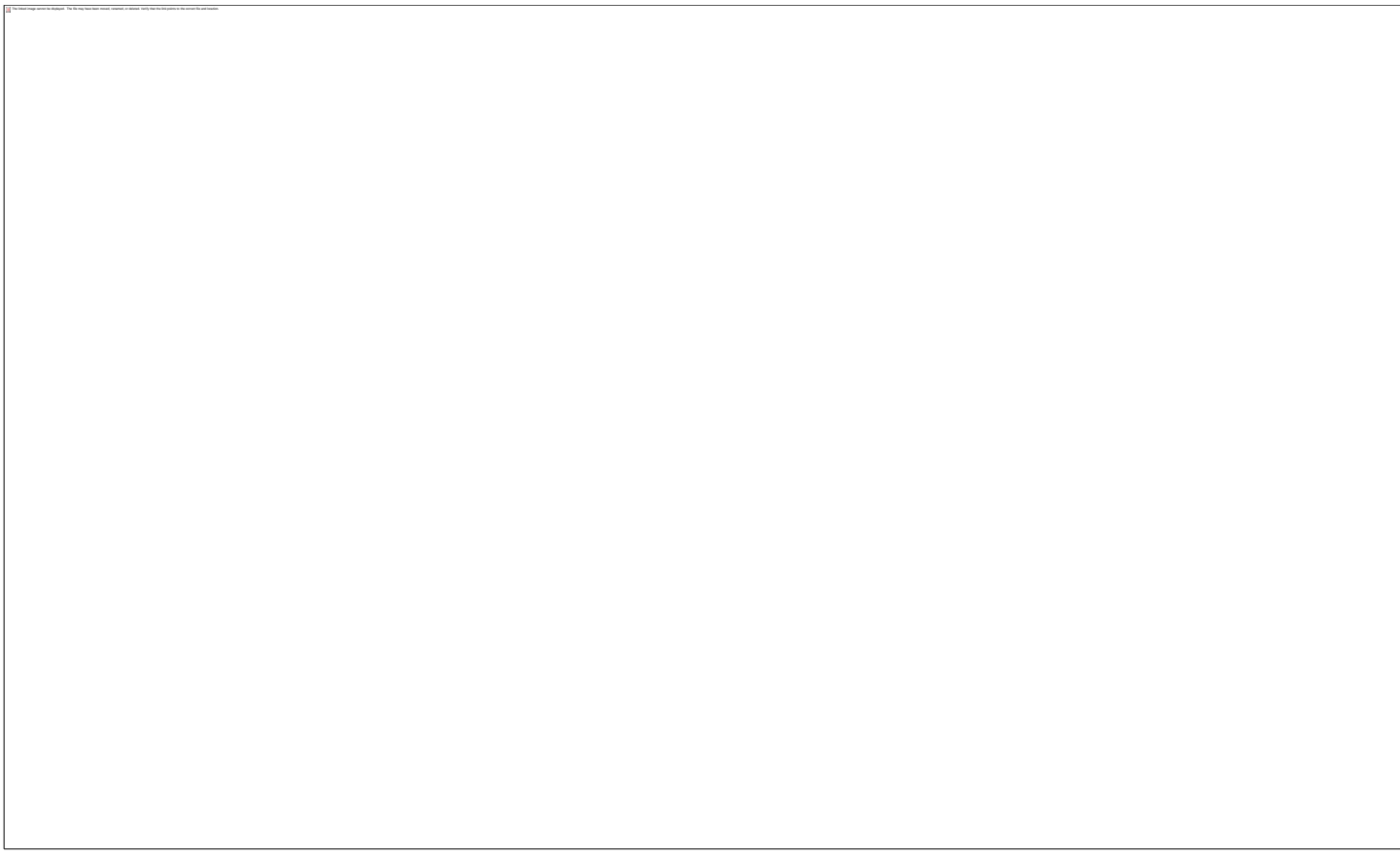
1.13.3.28 void randomUprofile (ComplexChebyCoeff & *u*, Real *mag*, Real *decay*)

References abs(), EPSILON, ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), ComplexChebyCoeff::length(), randomComplex(), ComplexChebyCoeff::set(), ComplexChebyCoeff::setState(), Spectral, and ComplexChebyCoeff::sub().

Referenced by randomProfile().

```
2058                                                {
2059   // Set a random u(y)
2060   int N = u.length();
2061   u.setState(Spectral);
2062   int n; // MSVC++ FOR-SCOPE BUG
2063   for (n=0; n<N; ++n) {
2064     u.set(n, mag*randomComplex());
2065     mag *= decay;
2066   }
2067   ChebyTransform trans(N);
2068   // Adjust u(y) so that u(+-1) == 0
2069   //cout << "before random u(a) == " << u.eval_a() << endl;
2070   //cout << "before random u(b) == " << u.eval_b() << endl;
2071   Complex u0 = (u.eval_b() + u.eval_a())/2.0;
2072   Complex u1 = (u.eval_b() - u.eval_a())/2.0;
2073   u.sub(0, u0);
2074   u.sub(1, u1);
2075   //cout << "after random u(a) == " << u.eval_a() << endl;
2076   //cout << "after random u(b) == " << u.eval_b() << endl;
2077
2078   assert(abs(u.eval_b()) < EPSILON);
2079   assert(abs(u.eval_a()) < EPSILON);
2080 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.13.3.29 void randomVprofile (ComplexChebyCoeff & *v*, Real *mag*, Real *decay*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), Greater(), ComplexChebyCoeff::length(), ComplexChebyCoeff::numModes(), randomComplex(), ComplexChebyCoeff::set(), ComplexChebyCoeff::setState(), Spectral, and ComplexChebyCoeff::sub().

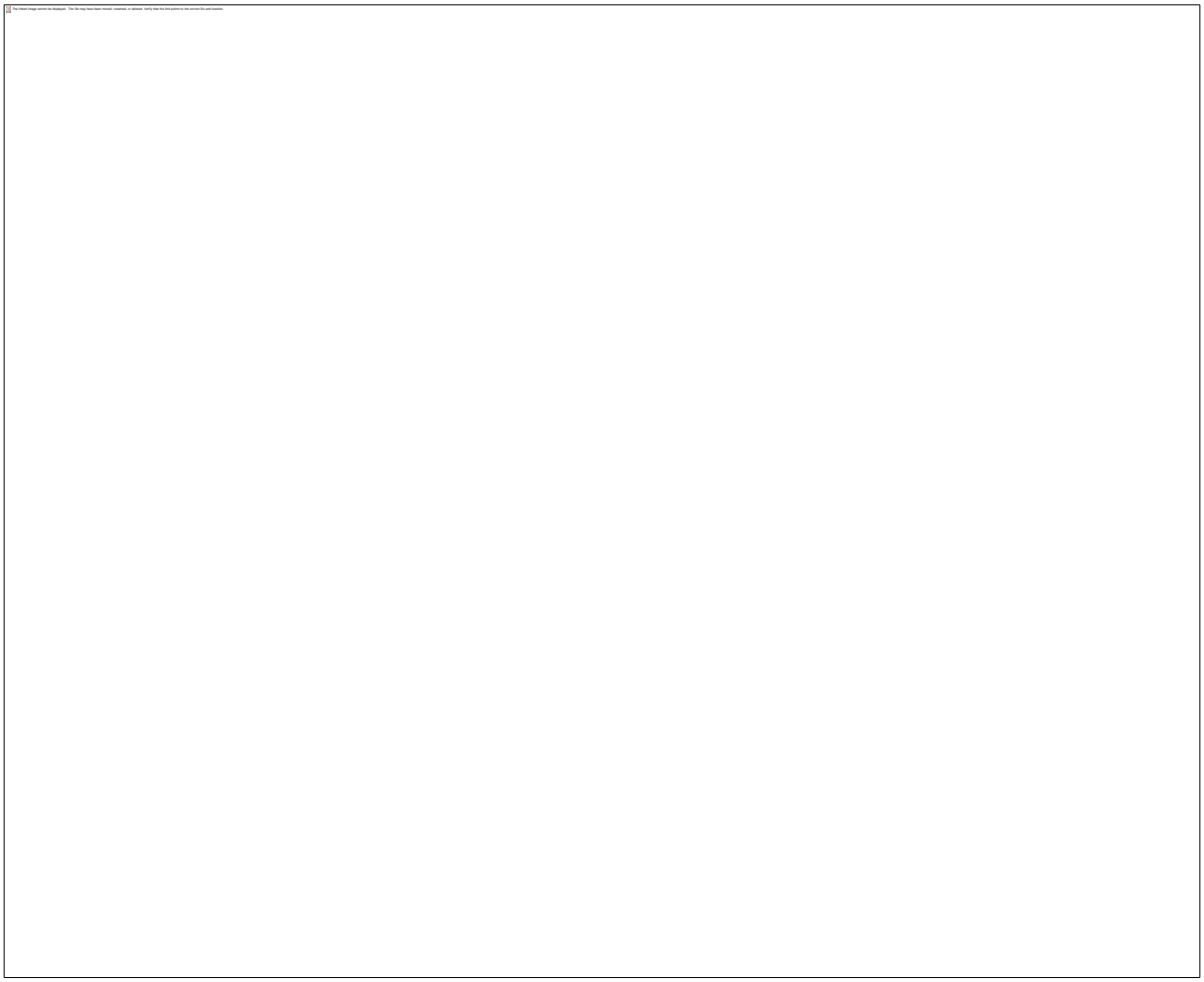
Referenced by randomProfile().

```
2082                                        {
2083   int N = v.length();
2084   v.setState(Spectral);
2085   // Assign a random v(y).
2086   v.set(0, 0.0);
2087   v.set(1, 0.0);
2088   v.set(2, 0.0);
```

```
2089  v.set(3, 0.0);
2090  int n; // MSVC++ FOR-SCOPE BUG
2091  for (n=0; n<4; ++n)
2092    v.set(n, 0.0);
2093  for (n=4; n<N-2; ++n) {
2094    v.set(n, mag*randomComplex());
2095    mag *= decay;
2096  }
2097
2098  for (n=Greater(N-2, 0); n<N; ++n)
2099    v.set(n, 0.0);
2100
2101  // Adjust v so that v(+-1) == v'(+/-1) == 0, by subtracting off
2102  // s0 T0(x) + s1 T1(x) + s2 T2(x) + s3 T3(x), with s's chosen to
2103  // have same BCs as v.
2104  ComplexChebyCoeff vy = diff(v);
2105
2106  Complex a = v.eval_a();
2107  Complex b = v.eval_b();
2108  Complex c = vy.eval_a();
2109  Complex d = vy.eval_b();
2110
2111  // The numercial coeffs are inverse of the matrix (values found with Maple)
2112  // T0(-1)  T1(-1)  T2(-1)  T3(-1)     s0      a
2113  // T0(1)   T1(1)   T2(1)   T3(1)      s1      b
2114  // T0'(-1) T1'(-1) T2'(-1) T3'(-1)    s2  ==  c
2115  // T0'(1)  T1'(1)  T2'(1)  T3'(1)     s3      d
2116
2117  Complex s0 = 0.5*(a + b) + 0.125*(c - d);
2118  Complex s1 = 0.5625*(b - a) - 0.0625*(c + d);
2119  Complex s2 = 0.125*(d - c);
2120  Complex s3 = 0.0625*(a - b + c + d);
2121
2122  ComplexChebyCoeff adj(v.numModes(), v.a(), v.b(), Spectral);
2123  adj.set(0, s0);
2124  adj.set(1, s1);
2125  adj.set(2, s2);
2126  adj.set(3, s3);
2127  ComplexChebyCoeff adjy = diff(adj);
2128
2129
2130  // Subtract off the coeffs
2131  v.sub(0, s0);
2132  v.sub(1, s1);
2133  v.sub(2, s2);
2134  v.sub(3, s3);
```

```
2135
2136   diff(v,vy);
2137
2138   //cout << "random v(a)  == " << v.eval_a() << endl;
2139   //cout << "random v(b)  == " << v.eval_b()  << endl;
2140   //cout << "random v'(a) == " << vy.eval_a() << endl;
2141   //cout << "random v'(b) == " << vy.eval_b() << endl;
2142
2143 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.13.3.30 void swap (FlowField & *f*, FlowField & *g*)

References FlowField::cdata_, FlowField::congruent(), FlowField::rdata_, FlowField::xz_iplan_, FlowField::xz_plan_, and FlowField::y_plan_.

```
1676                                {
1677   assert(f.congruent(g));
1678   Real* tmp = f.rdata_;
1679   f.rdata_ = g.rdata_;
1680   g.rdata_ = tmp;
1681
1682   f.cdata_ = (Complex*)f.rdata_;
1683   g.cdata_ = (Complex*)g.rdata_;
1684
1685   // We must swap the FFTW3 plans, as well, since they're tied to specific
1686   // memory locations. This burned me when I upgraded FFTW2->FFTW3.
1687   fftw_plan tmp_plan;
1688   tmp_plan = f.xz_plan_;
1689   f.xz_plan_ = g.xz_plan_;
1690   g.xz_plan_ = tmp_plan;
1691
1692   tmp_plan = f.xz_iplan_;
1693   f.xz_iplan_ = g.xz_iplan_;
1694   g.xz_iplan_ = tmp_plan;
1695
1696   tmp_plan = f.y_plan_;
1697   f.y_plan_ = g.y_plan_;
1698   g.y_plan_ = tmp_plan;
1699 }
```

Here is the call graph for this function:

1.13.4 Variable Documentation

1.13.4.1 const Real EPSILON = 1e-4

1.13.4.2

1.14 /_cuda/channelflow-0.9.22/channelflow/flowfield.h File Reference

#include <fftw3.h>
#include "channelflow/mathdefs.h"
#include "channelflow/vector.h"
#include "channelflow/chebyshev.h"
#include "channelflow/basisfunc.h"
Include dependency graph for flowfield.h:

This graph shows which files directly or indirectly include this file:

1.14.1 Classes

- class FlowField

1.14.2 Functions

- Real L1Norm (const FlowField &f, bool normalize=true)
- Real L1Dist (const FlowField &f, const FlowField &g, bool normalize=true)
- Real LinfNorm (const FlowField &f)
- Real LinfDist (const FlowField &f, const FlowField &g)
- Real L2Norm (const FlowField &f, bool normalize=true)
- Real L2Norm2 (const FlowField &f, bool normalize=true)
- Real L2Dist (const FlowField &f, const FlowField &g, bool normalize=true)
- Real L2Dist2 (const FlowField &f, const FlowField &g, bool normalize=true)

- Real bcNorm (const FlowField &f, bool normalize=true)
- Real bcNorm2 (const FlowField &f, bool normalize=true)
- Real bcDist (const FlowField &f, const FlowField &g, bool normalize=true)
- Real bcDist2 (const FlowField &f, const FlowField &g, bool normalize=true)
- Real divNorm (const FlowField &f, bool normalize=true)
- Real divNorm2 (const FlowField &f, bool normalize=true)
- Real divDist (const FlowField &f, const FlowField &g, bool normalize=true)
- Real divDist2 (const FlowField &f, const FlowField &g, bool normalize=true)
- Real L2Norm (const FlowField &f, int kxmax, int kzmax, bool normalize=true)
- Real L2Norm2 (const FlowField &f, int kxmax, int kzmax, bool normalize=true)
- Real L2Dist (const FlowField &f, const FlowField &g, int kxmax, int kzmax, bool normalize=true)
- Real L2Dist2 (const FlowField &f, const FlowField &g, int kxmax, int kzmax, bool normalize=true)
- Real L2InnerProduct (const FlowField &f, const FlowField &g, bool normalize=true)
- Complex L2InnerProduct (const FlowField &f, const BasisFunc &phi, bool normalize=false)
- Complex L2InnerProduct (const FlowField &f, const ComplexChebyCoeff &phi, bool normalize=false)
- void swap (FlowField &f, FlowField &v)
- void assignOrrSommField (FlowField &u, FlowField &P, Real t, Real Reynolds, Complex omega, const ComplexChebyCoeff &ueig, const ComplexChebyCoeff &veig, const ComplexChebyCoeff &peig)
- int zero_last_mode (int k, int kmax, int n)
- void randomUprofile (ComplexChebyCoeff &u, Real mag, Real spectralDecay)
- void randomVprofile (ComplexChebyCoeff &v, Real mag, Real spectralDecay)
- void randomProfile (ComplexChebyCoeff &u, ComplexChebyCoeff &v, ComplexChebyCoeff &w, int kx, int kz, Real Lx, Real Lz, Real mag, Real spectralDecay)
- void chebyUprofile (ComplexChebyCoeff &u, int n, Real decay)
- void chebyVprofile (ComplexChebyCoeff &v, int n, Real decay)
- void chebyProfile (ComplexChebyCoeff &u, ComplexChebyCoeff &v, ComplexChebyCoeff &w, int un, int vn, int kx, int kz, Real Lx, Real Lz, Real decay)

1.14.3 Function Documentation

1.14.3.1 void assignOrrSommField (FlowField & *u*, FlowField & *P*, Real *t*, Real *Reynolds*, Complex *omega*, const ComplexChebyCoeff & *ueig*, const ComplexChebyCoeff & *veig*, const ComplexChebyCoeff & *peig*)

References FlowField::chebyfft_y(), FlowField::cmplx(), exp(), I(), FlowField::irealfft_xz(), FlowField::Mx(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), Physical, FlowField::realfft_xz(), FlowField::setState(), FlowField::setToZero(), Spectral, square(), and FlowField::y().

```
2442                                    {
2443
2444   int Ny=u.Ny();
2445
2446   // Reconstruct velocity field (Poisseuille plus OS perturbation) from
2447   // y-profile of (kx,kz) == (1,0) Spectral mode of pertubation (ueig & veig).
2448   u.setState(Spectral, Physical);
2449   u.setToZero();
2450   Complex c = exp((-t*omega)*I);
2451   int n=u.Mx()-1;
2452   int ny; // MSVC++ FOR-SCOPE BUG
2453   for (ny=0; ny<Ny; ++ny) {
2454     Complex uc = c*ueig[ny];
2455     Complex vc = c*veig[ny];
2456     u.cmplx(0, ny, 0, 0) = Complex(1.0-square(u.y(ny)));
2457     u.cmplx(1, ny, 0, 0) = uc;
2458     u.cmplx(1, ny, 0, 1) = vc;
2459     u.cmplx(n, ny, 0, 0) = conj(uc);
2460     u.cmplx(n, ny, 0, 1) = conj(vc);
2461   }
2462
2463   // Assign pressure perturbation p to P field.
2464   P.setState(Spectral, Physical);
2465   P.setToZero();
2466   for (ny=0; ny<Ny; ++ny) {
2467     Complex pc = c*peig[ny];
2468     P.cmplx(1, ny, 0, 0) = pc;
2469     P.cmplx(n, ny, 0, 0) = conj(pc);
2470   }
2471
2472   // Add velocity contrib to get modified pressure P = p + 1/2 |u|^2
2473   u.irealfft_xz();
2474   P.irealfft_xz();
2475
2476   for (ny=0; ny<Ny; ++ny)
2477     for (int nx=0; nx<u.Nx(); ++nx)
2478       for (int nz=0; nz<u.Nz(); ++nz)
2479         P(nx,ny,nz,0) += 0.5*(square(u(nx,ny,nz,0)) +
2480                               square(u(nx,ny,nz,1)) +
2481                               square(u(nx,ny,nz,2)));
2482
2483   u.realfft_xz();
2484   P.realfft_xz();
2485   u.chebyfft_y();
2486   P.chebyfft_y();
2487 }
```

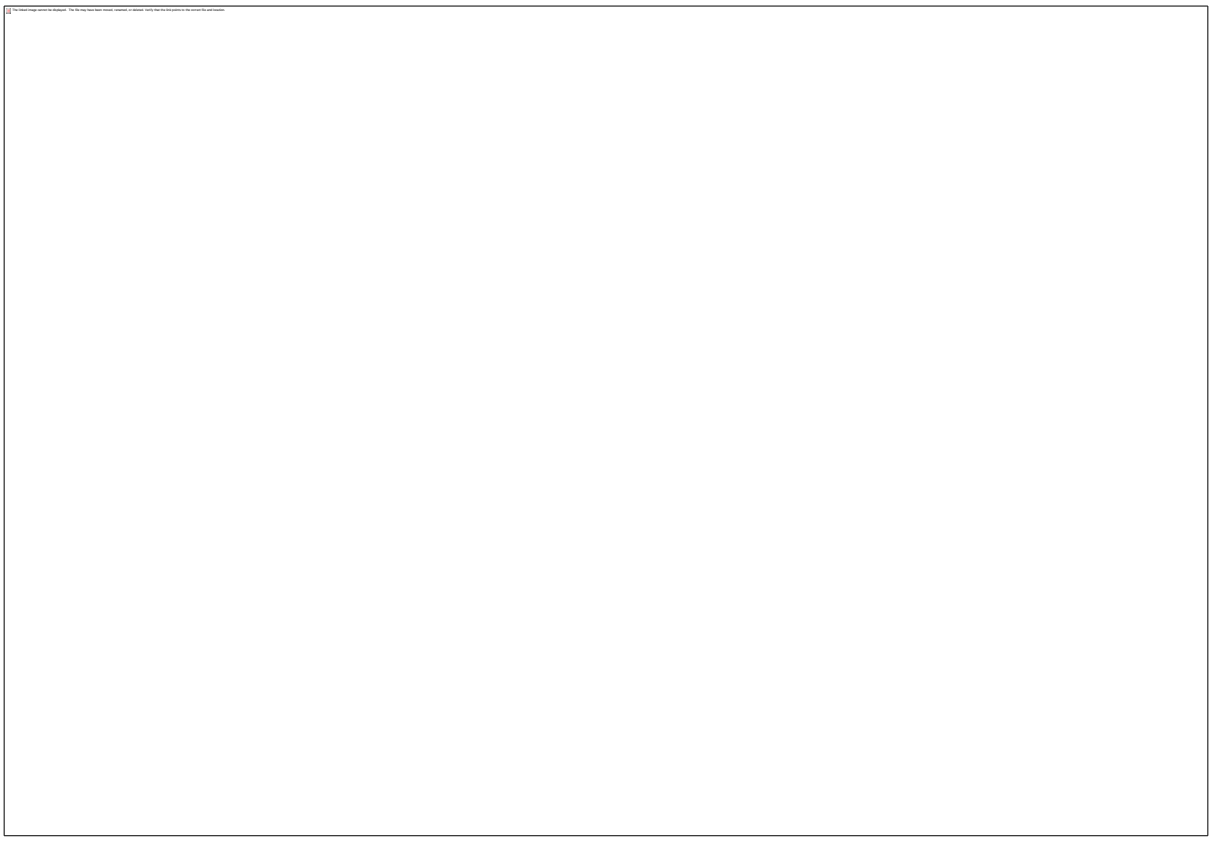
Here is the call graph for this function:

1.14.3.2 Real bcDist (const FlowField & *f*, const FlowField & *g*, bool *normalize* = true)

References bcDist2().

```
1755                                                    {
1756   return sqrt(bcDist2(f,g, normalize));
1757 }
```

Here is the call graph for this function:



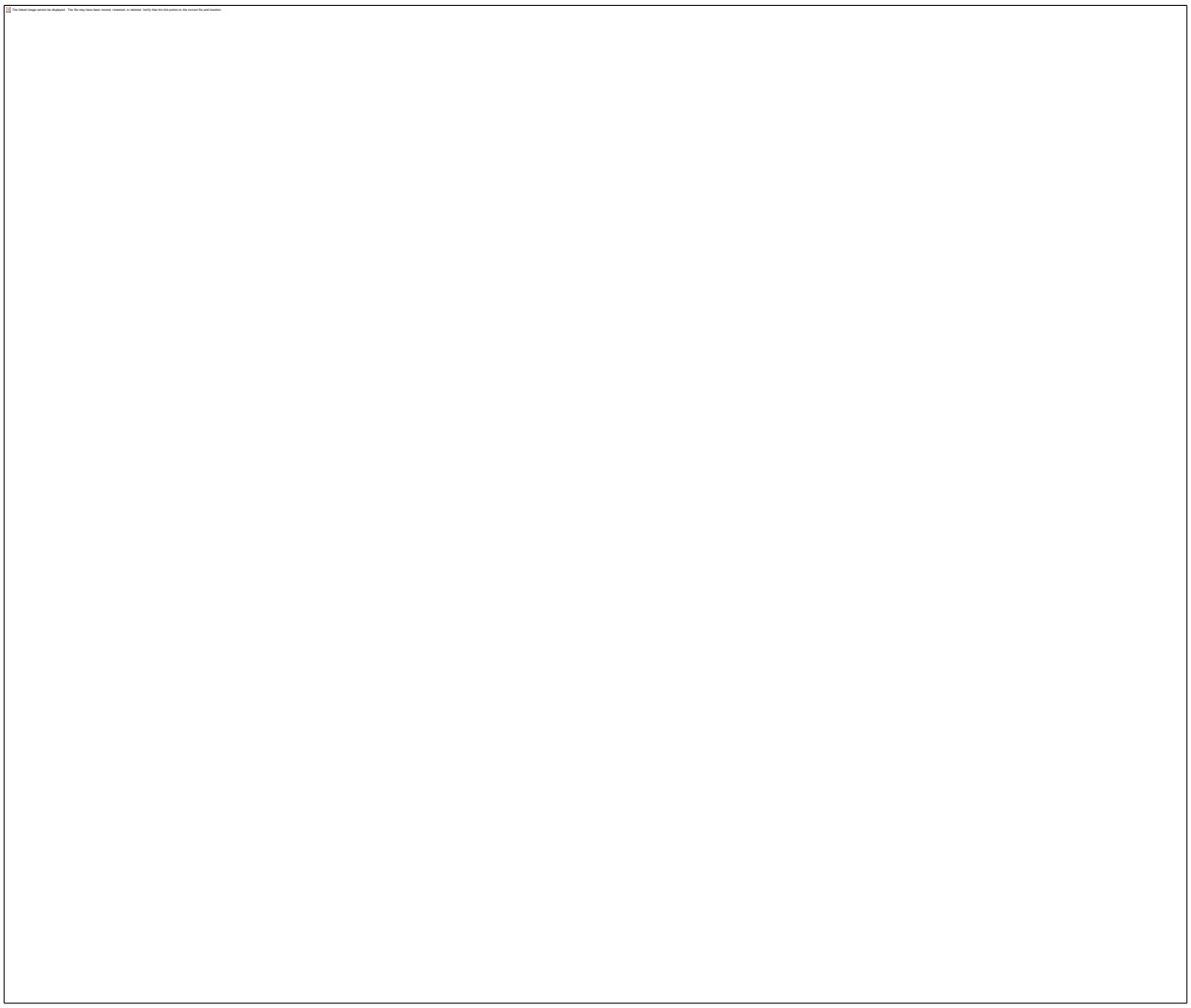
1.14.3.3 Real bcDist2 (const FlowField & *f*, const FlowField & *g*, bool *normalize* = true)

References FlowField::a(), abs2(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), diff(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), Spectral, FlowField::xzstate(), and FlowField::ystate().

```
1724                                                    {
1725   assert(f.congruent(g));
1726   assert(f.xzstate() == Spectral && g.xzstate() == Spectral);
1727   //assert(f.ystate() == Spectral && g.ystate() == Spectral);
1728
1729   Real bc2 = 0.0;
1730   int Nd = f.Nd();
1731   int Mx = f.Mx();
1732   int My = f.My();
```

```
1733   int Mz = f.Mz();
1734   ComplexChebyCoeff diff(My, f.a(), f.b(), f.ystate());
1735
1736   for (int i=0; i<Nd; ++i)
1737     for (int mx=0; mx<Mx; ++mx) {
1738       for (int mz=0; mz<Mz; ++mz) {
1739         for (int my=0; my<My; ++my)
1740           diff.set(my, f.cmplx(mx,my,mz,i)-g.cmplx(mx,my,mz,i));
1741         bc2 += abs2(diff.eval_a());
1742         bc2 += abs2(diff.eval_b());
1743       }
1744     }
1745   if (!normalize)
1746     bc2 *= f.Lx()*f.Lz();
1747   return bc2;
1748 }
```

Here is the call graph for this function:



1.14.3.4 Real bcNorm (const FlowField & *f*, bool *normalize* = true)

References bcNorm2().

```
1751                                    {
1752   return sqrt(bcNorm2(f, normalize));
1753 }
```

Here is the call graph for this function:

1.14.3.5 Real bcNorm2 (const FlowField & *f*, bool *normalize* = true)

References FlowField::a(), abs2(), FlowField::b(), FlowField::cmplx(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::My(), FlowField::Mz(), FlowField::Nd(), FlowField::Ny(), Spectral, FlowField::xzstate(), and FlowField::ystate().

```
1703                                        {
1704   assert(f.xzstate() == Spectral);
1705   Real bc2 = 0.0;
1706   int Mx = f.Mx();
1707   int My = f.My();
1708   int Mz = f.Mz();
1709   ComplexChebyCoeff prof(f.Ny(), f.a(), f.b(), f.ystate());
1710   for (int i=0; i<f.Nd(); ++i)
1711     for (int mx=0; mx<Mx; ++mx) {
1712       for (int mz=0; mz<Mz; ++mz) {
1713         for (int my=0; my<My; ++my)
1714           prof.set(my, f.cmplx(mx,my,mz,i));
1715         bc2 += abs2(prof.eval_a());
1716         bc2 += abs2(prof.eval_b());
1717       }
1718     }
1719   if (!normalize)
1720     bc2 *= f.Lx()*f.Lz();
1721   return bc2;
1722 }
```

Here is the call graph for this function:

1.14.3.6 void chebyProfile (ComplexChebyCoeff & *u*, ComplexChebyCoeff & *v*, ComplexChebyCoeff & *w*, int *un*, int *vn*, int *kx*, int *kz*, Real *Lx*, Real *Lz*, Real *decay*)

References ComplexChebyCoeff::a(), abs(), ComplexChebyCoeff::b(), chebyUprofile(), chebyVprofile(), diff(), div(), divNorm(), EPSILON, ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), I(), ComplexChebyCoeff::im, L2Norm(), ComplexChebyCoeff::length(), ComplexChebyCoeff::numModes(), pi, ComplexChebyCoeff::setState(), ComplexChebyCoeff::setToZero(), ChebyCoeff::setToZero(), and Spectral.

```
2333
{
2334
2335  int N = u.length();
2336  ChebyTransform trans(N);
2337  u.setState(Spectral);
2338  v.setState(Spectral);
2339  w.setState(Spectral);
2340  //u.setToZero();
2341  //v.setToZero();
2342  //w.setToZero();
2343  if (kx == 0 && kz == 0) {
2344    chebyUprofile(u, un, decay);
2345    chebyUprofile(w, vn, decay); // yes, vn
2346    u.im.setToZero();
2347    w.im.setToZero();
2348    v.setToZero();
2349  }
2350  else {
2351    // Other kx,kz cases are based on a random v(y).
2352
2353    chebyVprofile(v, vn, decay);
2354    ComplexChebyCoeff vy = diff(v);
2355
2356    if (kx == 0) {
2357      chebyUprofile(u, un, decay);
2358      u.im.setToZero();
2359      w = vy;
2360      w *= -Lz/((2*pi*kz)*I);
2361    }
2362    else if (kz == 0) {
2363      chebyUprofile(w, un, decay); // yes, un
2364      w.im.setToZero();
2365      u = vy;
2366      u *= -Lx/((2*pi*kx)*I);
2367    }
```

```
2368    else {
2369
2370      ComplexChebyCoeff v0(v);
2371      ComplexChebyCoeff v1(v);
2372      chebyVprofile(v0, vn, decay);
2373      chebyVprofile(v1, vn, decay);
2374
2375      ComplexChebyCoeff v0y = diff(v0);
2376      ComplexChebyCoeff v1y = diff(v1);
2377
2378      // Finally, the general case, where kx, kz != 0 and u,v,w are nonzero
2379      // Set a random u(y)
2380      ComplexChebyCoeff u0(v.numModes(), v.a(), v.b(), Spectral);
2381      ComplexChebyCoeff w0(v.numModes(), v.a(), v.b(), Spectral);
2382      ComplexChebyCoeff u1(v.numModes(), v.a(), v.b(), Spectral);
2383      ComplexChebyCoeff w1(v.numModes(), v.a(), v.b(), Spectral);
2384
2385      chebyUprofile(u0, un, decay);
2386
2387      // Calculate w0 from div u0 == u0x + v0y + w0z == 0.
2388      ComplexChebyCoeff u0x(u0);
2389      u0x *= (2*pi*kx/Lx)*I;
2390      w0 = v0y;
2391      w0 += u0x;
2392      w0 *= -Lz/((2*pi*kz)*I);       // Set w = -Lz/(2*pi*I*kz) * (ux + vy);
2393
2394
2395      //randomUprofile(w1, mag, decay);
2396
2397      // Calculate u0 from div u0 == u0x + v0y + w0z == 0.
2398      ComplexChebyCoeff w1z(w1);
2399      w1z *= (2*pi*kz/Lz)*I;
2400      u1 = v1y;
2401      u1 += w1z;
2402      u1 *= -Lx/((2*pi*kx)*I);       // Set w = -Lz/(2*pi*I*kz) * (ux + vy);
2403
2404      u = u0;
2405      v = v0;
2406      w = w0;
2407      u += u1;
2408      v += v1;
2409      w += w1;
2410    }
2411  }
2412
2413  // Check divergence
```

```
2414   ComplexChebyCoeff ux(u);
2415   ux *= (2*pi*kx/Lx)*I;
2416   ComplexChebyCoeff wz(w);
2417   wz *= (2*pi*kz/Lz)*I;
2418   ComplexChebyCoeff vy = diff(v);
2419
2420   ComplexChebyCoeff div(ux);
2421   div += vy;
2422   div += wz;
2423
2424   Real divNorm = L2Norm(div);
2425   Real ubcNorm = abs(u.eval_a()) + abs(u.eval_b());
2426   Real vbcNorm = abs(v.eval_a()) + abs(v.eval_b());
2427   Real wbcNorm = abs(w.eval_a()) + abs(w.eval_b());
2428   assert(divNorm < EPSILON);
2429   assert(ubcNorm < EPSILON);
2430   assert(vbcNorm < EPSILON);
2431   assert(wbcNorm < EPSILON);
2432   // supress unused-variable compiler warnings...
2433   wbcNorm += divNorm + ubcNorm + vbcNorm;
2434
2435 }
```

Here is the call graph for this function:

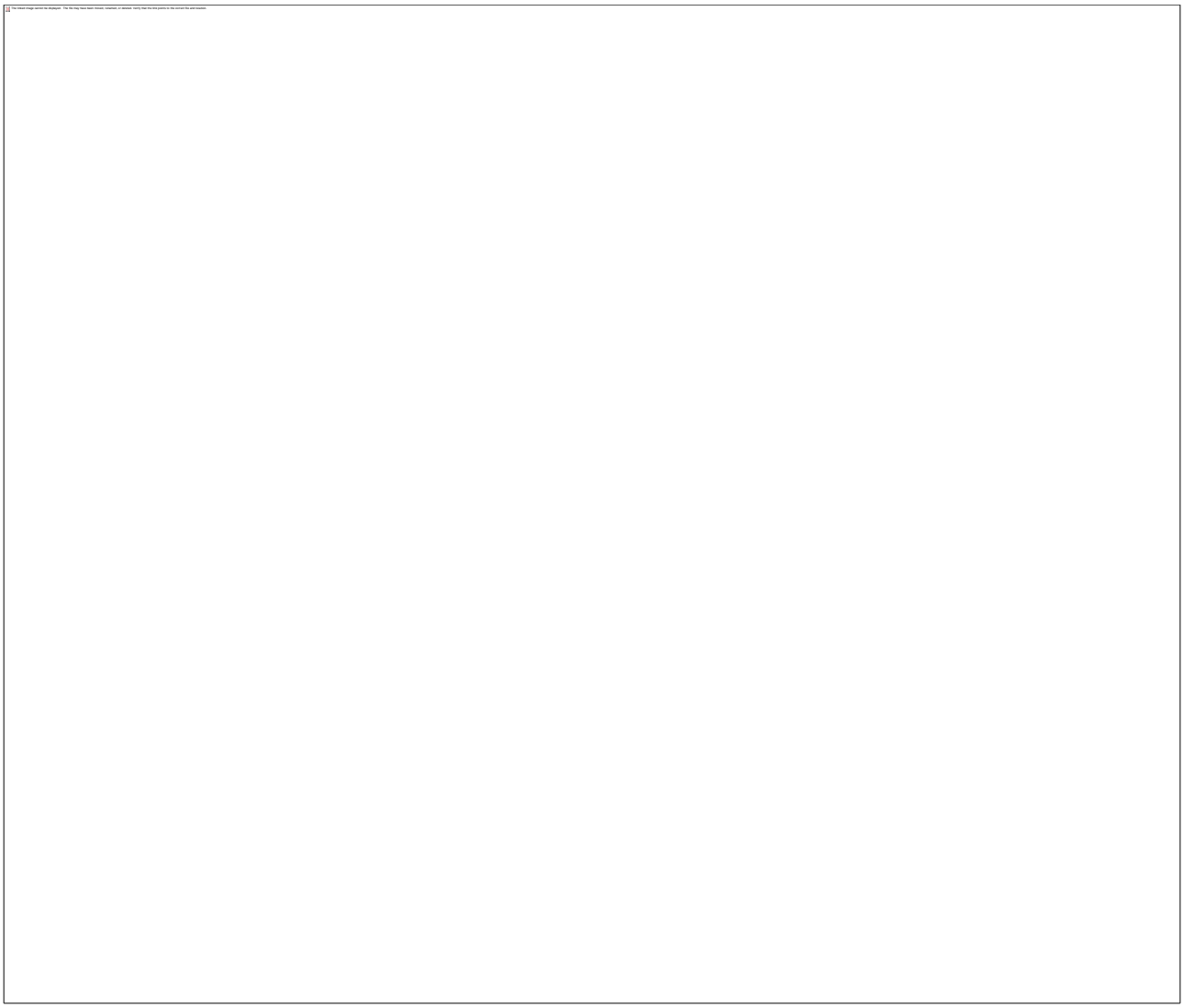
1.14.3.7 void chebyUprofile (ComplexChebyCoeff & *u*, int *n*, Real *decay*)

References abs(), EPSILON, ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), I(), ComplexChebyCoeff::length(), pi, pow(), randomReal(), ComplexChebyCoeff::set(), ComplexChebyCoeff::setState(), ComplexChebyCoeff::setToZero(), Spectral, and ComplexChebyCoeff::sub().

Referenced by chebyProfile().

```
2146                                        {
2147   // Set a random u(y)
2148   int N = u.length();
2149   u.setToZero();
2150   u.setState(Spectral);
2151   Real theta = pi*randomReal();
2152   u.set(n, (cos(theta) + I*sin(theta))*pow(decay,n));
2153
2154   ChebyTransform trans(N);
2155
2156   // Adjust u(y) so that u(+-1) == 0
2157   Complex u0 = (u.eval_b() + u.eval_a())/2.0;  // 2.0 is correct for genl a,b
2158   Complex u1 = (u.eval_b() - u.eval_a())/2.0;  // 2.0 is correct for genl a,b
2159   u.sub(0, u0);
2160   u.sub(1, u1);
2161   //cout << "random u(a) == " << u.eval_a() << endl;
2162   //cout << "random u(b) == " << u.eval_b() << endl;
2163   assert(abs(u.eval_b()) < EPSILON);
2164   assert(abs(u.eval_a()) < EPSILON);
2165 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.14.3.8 void chebyVprofile (ComplexChebyCoeff & *v*, int *n*, Real *decay*)

References diff(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), I(), pi, pow(), randomReal(), ComplexChebyCoeff::set(), ComplexChebyCoeff::setState(), ComplexChebyCoeff::setToZero(), Spectral, and ComplexChebyCoeff::sub().

Referenced by chebyProfile().

```
2167                                        {
2168   v.setToZero();
2169   v.setState(Spectral);
```

```
2170  Real theta = pi*randomReal();
2171  v.set(n, (cos(theta) + I*sin(theta))*pow(decay,n));
2172
2173  // Adjust v so that v(+-1) == v'(+/-1) == 0, by subtracting off
2174  // s0 T0(x) + s1 T1(x) + s2 T2(x) + s3 T3(x), with s's chosen to
2175  // have same BCs as v.
2176  ComplexChebyCoeff vy = diff(v);
2177
2178  Complex a = v.eval_a();
2179  Complex b = v.eval_b();
2180  Complex c = vy.eval_a();
2181  Complex d = vy.eval_b();
2182
2183  // The numercial coeffs are inverse of the matrix (values found with Maple)
2184  // T0(-1)  T1(-1)  T2(-1)  T3(-1)     s0      a
2185  // T0(1)   T1(1)   T2(1)   T3(1)      s1      b
2186  // T0'(-1) T1'(-1) T2'(-1) T3'(-1)    s2  ==  c
2187  // T0'(1)  T1'(1)  T2'(1)  T3'(1)     s3      d
2188
2189  // The above matrix is
2190  // 1  -1   1  -1
2191  // 1   1   1   1
2192  // 0   1  -4   9
2193  // 0   1   4   9
2194
2195  Complex s0 = 0.5*(a + b) + 0.125*(c - d);
2196  Complex s1 = 0.5625*(b - a) - 0.0625*(c + d);
2197  Complex s2 = 0.125*(d - c);
2198  Complex s3 = 0.0625*(a - b + c + d);
2199
2200  //ComplexChebyCoeff adj(v.numModes(), v.a(), v.b(), Spectral);
2201  //adj.set(0, s0);
2202  //adj.set(1, s1);
2203  //adj.set(2, s2);
2204  //adj.set(3, s3);
2205  //ComplexChebyCoeff adjy = diff(adj);
2206
2207  // Subtract off the coeffs
2208  v.sub(0, s0);
2209  v.sub(1, s1);
2210  v.sub(2, s2);
2211  v.sub(3, s3);
2212
2213  //diff(v,vy);
2214
2215  //cout << "random v(a)  == " << v.eval_a() << endl;
```

```
2216   //cout << "random v(b)  == " << v.eval_b()  << endl;
2217   //cout << "random v'(a) == " << vy.eval_a() << endl;
2218   //cout << "random v'(b) == " << vy.eval_b() << endl;
2219
2220 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.14.3.9 Real divDist (const FlowField & *f*, const FlowField & *g*, bool *normalize* = true)

References divDist2().

```
1790                                                    {
1791   return sqrt(divDist2(f,g,normalize));
1792 }
```

Here is the call graph for this function:

1.14.3.10 Real divDist2 (const FlowField & *f*, const FlowField & *g*, bool *normalize* = true)

References FlowField::congruent(), divDist2(), FlowField::Mx(), FlowField::Mz(), FlowField::Nd(), FlowField::profile(), Spectral, FlowField::xzstate(), and FlowField::ystate().

```
1776                                                    {
1777   assert(f.xzstate() == Spectral && f.ystate() == Spectral);
1778   assert(f.congruent(g));
1779   assert(f.Nd() == 3);
1780
1781   Real div2=0.0;
1782   int Mx = f.Mx();
1783   int Mz = f.Mz();
1784   for (int mx=0; mx<Mx; ++mx)
1785     for (int mz=0; mz<Mz; ++mz)
1786       div2 += divDist2(f.profile(mx,mz), g.profile(mx,mz), normalize);
1787   return div2;
1788 }
```

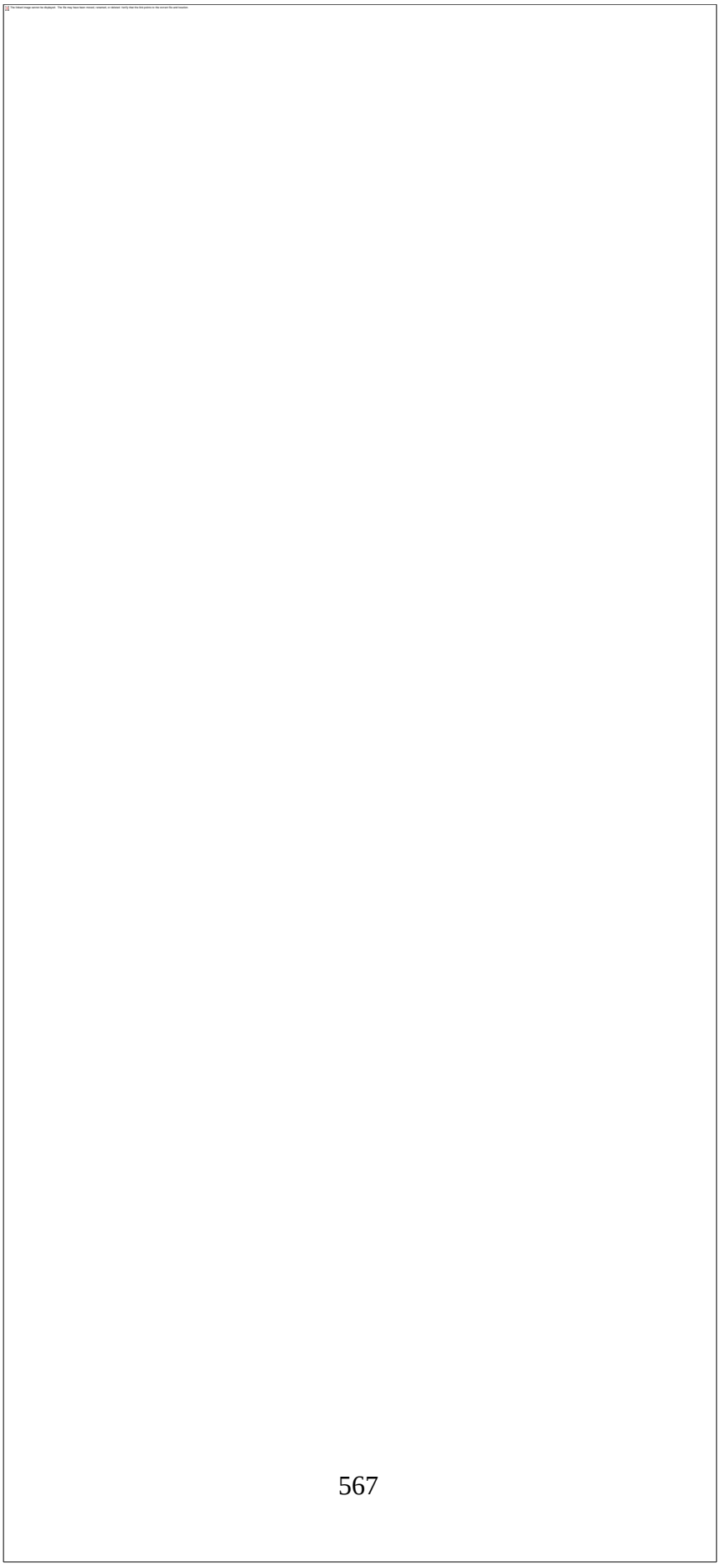
Here is the call graph for this function:

1.14.3.11 Real divNorm (const FlowField & *f*, bool *normalize* = true)

References divNorm2().

```
1759                                        {
1760   return sqrt(divNorm2(f,normalize));
1761 }
```

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.

1.14.3.12 Real divNorm2 (const FlowField & *f*, bool *normalize* = true)

References divNorm2(), FlowField::Mx(), FlowField::Mz(), FlowField::Nd(), FlowField::profile(), Spectral, FlowField::xzstate(), and FlowField::ystate().

```
1763                                        {
1764   assert(f.xzstate() == Spectral && f.ystate() == Spectral);
1765   assert(f.Nd() == 3);
1766
1767   Real div2=0.0;
1768   int Mx = f.Mx();
1769   int Mz = f.Mz();
1770   BasisFunc prof;
1771   for (int mx=0; mx<Mx; ++mx)
1772     for (int mz=0; mz<Mz; ++mz)
1773       div2 += divNorm2(f.profile(mx,mz), normalize);
1774   return div2;
1775 }
```

Here is the call graph for this function:

1.14.3.13 Real L1Dist (const FlowField & *f*, const FlowField & *g*, bool *normalize* = true)

References FlowField::congruent(), L1Norm(), FlowField::Lx(), FlowField::Lz(), ChebyCoeff::makePhysical(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), FlowField::profile(), Re(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1821                                                {
1822   assert(f.congruent(g));
1823   fieldstate fxstate = f.xzstate();
1824   fieldstate gxstate = g.xzstate();
1825   fieldstate fystate = f.ystate();
1826   fieldstate gystate = g.ystate();
1827
1828   ((FlowField&) f).makePhysical_xz();
1829   ((FlowField&) g).makePhysical_xz();
1830
1831   Real sum = 0.0;
1832
1833   ChebyCoeff prof;
1834   ChebyTransform trans(f.Ny());
1835   for (int i=0; i<f.vectorDim(); ++i)
1836     for (int nx=0; nx<f.Nx(); ++nx)
1837       for (int nz=0; nz<f.Nz(); ++nz) {
1838         prof = Re(f.profile(nx,nz,i));
1839         prof -= Re(g.profile(nx,nz,i));
1840         prof.makePhysical(trans);
1841         sum += L1Norm(prof, normalize);
1842       }
1843
1844   sum /= f.Nx()*f.Nz();
1845
1846   if (!normalize)
1847     sum *= f.Lx()*f.Lz();
1848
1849   ((FlowField&) f).makeState(fxstate, fystate);
1850   ((FlowField&) g).makeState(gxstate, gystate);
1851
1852   return sum;
1853 }
```

Here is the call graph for this function:

1.14.3.14 Real L1Norm (const FlowField & *f*, bool *normalize* = true)

References L1Norm(), FlowField::Lx(), FlowField::Lz(), ChebyCoeff::makePhysical(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), FlowField::profile(), Re(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1794                                    {
1795   fieldstate xstate = u.xzstate();
1796   fieldstate ystate = u.ystate();
1797   ((FlowField&) u).makePhysical_xz();
1798
1799   Real sum = 0.0;
1800
1801   ChebyCoeff prof;
1802   ChebyTransform trans(u.Ny());
1803   for (int i=0; i<u.vectorDim(); ++i)
1804     for (int nx=0; nx<u.Nx(); ++nx)
1805       for (int nz=0; nz<u.Nz(); ++nz) {
1806         prof = Re(u.profile(nx,nz,i));
1807         prof.makePhysical(trans);
1808         sum += L1Norm(prof);
1809       }
1810
1811   sum /= u.Nx()*u.Nz();
1812
1813   if (!normalize)
1814     sum *= u.Lx()*u.Lz();
1815
1816   ((FlowField&) u).makeState(xstate, ystate);
1817
1818   return sum;
1819 }
```

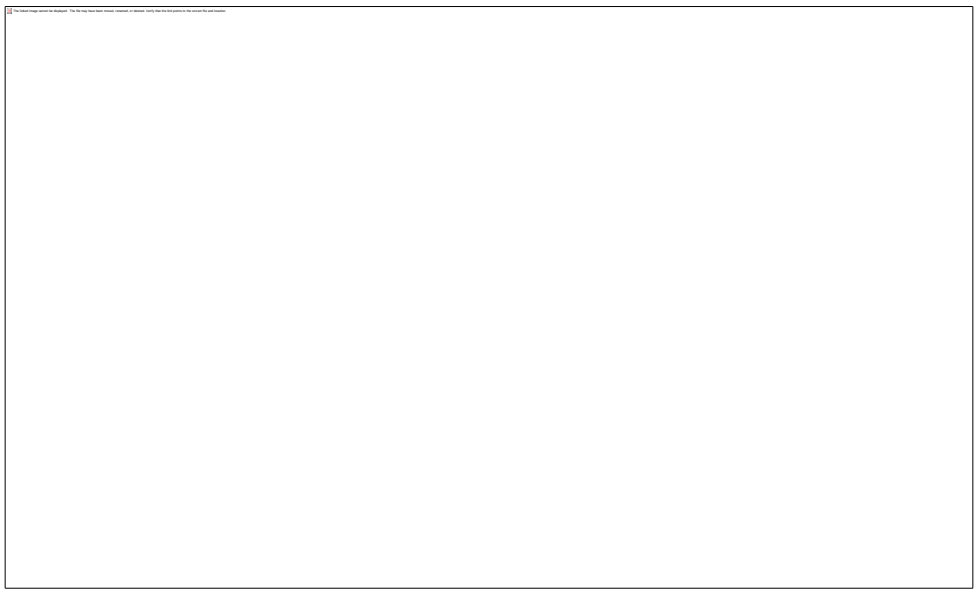
Here is the call graph for this function:

1.14.3.15 Real L2Dist (const FlowField & *f*, const FlowField & *g*, int *kxmax*, int *kzmax*, bool *normalize* = true)

References L2Dist2().

```
2010                                                                {
2011   return sqrt(L2Dist2(u, v, kxmax, kzmax, normalize));
2012 }
```

Here is the call graph for this function:



1.14.3.16 Real L2Dist (const FlowField & *f*, const FlowField & *g*, bool *normalize* = true)

References L2Dist2().

```
1906                                                        {
1907   return sqrt(L2Dist2(u,v,normalize));
1908 }
```

Here is the call graph for this function:

1.14.3.17 Real L2Dist2 (const FlowField & *f*, const FlowField & *g*, int *kxmax*, int *kzmax*, bool *normalize* = true)

References FlowField::a(), abs(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), FlowField::kx(), FlowField::kz(), L2Norm2(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::Mz(), FlowField::Ny(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
2014                                                    {
2015   Real sum = 0.0;
2016   assert(u.congruent(v));
2017   assert(u.xzstate()==Spectral && u.ystate()==Spectral);
2018   ComplexChebyCoeff u_v(u.Ny(), u.a(), u.b(), Spectral);
2019   for (int i=0; i<u.vectorDim(); ++i)
2020     for (int mx=0; mx<u.Mx(); ++mx) {
2021       if (abs(u.kx(mx)) > kxmax)
2022         continue;
2023       int cz = 1;
2024       for (int mz=0; mz<u.Mz(); ++mz) {
2025         if (abs(u.kz(mz)) > kzmax)
2026           continue;
2027         for (int ny=0; ny<u.Ny(); ++ny)
2028           u_v.set(ny, u.cmplx(mx,ny,mz,i)-v.cmplx(mx,ny,mz,i));
2029         sum += cz*L2Norm2(u_v, normalize);
2030         cz = 2;
2031       }
2032     }
2033   if (!normalize)
2034     sum *= u.Lx()*u.Lz();
```

```
2035   return sum;
2036 }
```

Here is the call graph for this function:

1.14.3.18 Real L2Dist2 (const FlowField & *f*, const FlowField & *g*, bool *normalize* = true)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), L2Norm2(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::Mz(), FlowField::Ny(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1910                                                {
1911   Real sum = 0.0;
1912   assert(u.congruent(v));
1913   assert(u.xzstate()==Spectral && u.ystate()==Spectral);
1914   ComplexChebyCoeff u_v(u.Ny(), u.a(), u.b(), Spectral);
1915   for (int i=0; i<u.vectorDim(); ++i)
1916     for (int mx=0; mx<u.Mx(); ++mx) {
1917       int cz = 1;
1918       for (int mz=0; mz<u.Mz(); ++mz) {
1919         for (int ny=0; ny<u.Ny(); ++ny)
1920           u_v.set(ny, u.cmplx(mx,ny,mz,i)-v.cmplx(mx,ny,mz,i));
1921         sum += cz*L2Norm2(u_v, normalize);
1922         cz = 2;
1923       }
1924     }
1925   if (!normalize)
1926     sum *= u.Lx()*u.Lz();
1927   return sum;
1928 }
```

Here is the call graph for this function:

1.14.3.19 Complex L2InnerProduct (const FlowField & *f*, const ComplexChebyCoeff & *phi*, bool *normalize* = false)

1.14.3.20 Complex L2InnerProduct (const FlowField & *f*, const BasisFunc & *phi*, bool *normalize* = false)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), BasisFunc::kx(), BasisFunc::kz(), L2InnerProduct(), FlowField::Lx(), FlowField::Lz(), FlowField::mx(), FlowField::mz(), FlowField::Nd(), FlowField::Ny(), Spectral, FlowField::xzstate(), and FlowField::ystate().

```
2039                                {
2040   assert(u.xzstate()==Spectral && u.ystate()==Spectral);
2041   assert(u.congruent(phi));
2042   int kx = phi.kx();
2043   int kz = phi.kz();
2044   int mx = u.mx(kx);
2045   int mz = u.mz(kz);
2046   int Ny = u.Ny();
2047   BasisFunc profile(Ny, kx, kz, u.Lx(), u.Lz(), u.a(), u.b(), Spectral);
2048
2049
2050   for (int i=0; i<u.Nd(); ++i)
2051     for (int ny=0; ny<Ny; ++ny)
2052       profile[i].set(ny, u.cmplx(mx,ny,mz, i));
2053
2054   return L2InnerProduct(profile, phi, normalize);
2055 }
```

Here is the call graph for this function:

1.14.3.21 Real L2InnerProduct (const FlowField & *f*, const FlowField & *g*, bool *normalize* = true)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), FlowField::congruent(), L2InnerProduct(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::Mz(), FlowField::Ny(), Re(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1954                                                    {
1955   assert(f.ystate() == Spectral);
1956   assert(g.ystate() == Spectral);
1957   assert(f.xzstate() == Spectral);
1958   assert(g.xzstate() == Spectral);
1959   assert(f.congruent(g));
1960
1961   Real sum = 0.0;
1962   ComplexChebyCoeff fprof(f.Ny(), f.a(), f.b(), Spectral);
1963   ComplexChebyCoeff gprof(g.Ny(), g.a(), g.b(), Spectral);
1964   for (int i=0; i<f.vectorDim(); ++i)
1965     for (int mx=0; mx<f.Mx(); ++mx) {
1966       Real cz = 1.0; // cz = 2 for kz>0 to take account of kz<0 ghost modes
1967       for (int mz=0; mz<f.Mz(); ++mz) {
1968         for (int ny=0; ny<f.Ny(); ++ny) {
1969           fprof.set(ny, f.cmplx(mx,ny,mz,i));
1970           gprof.set(ny, g.cmplx(mx,ny,mz,i));
1971         }
1972         sum += cz*Re(L2InnerProduct(fprof, gprof, normalize));
1973         cz = 2.0;
1974       }
1975     }
1976   if (!normalize)
1977     sum *= f.Lx()*f.Lz();
1978   return sum;
1979 }
```

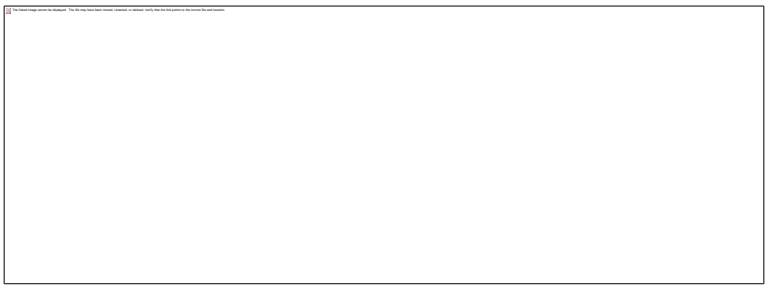
Here is the call graph for this function:

1.14.3.22 Real L2Norm (const FlowField & *f*, int *kxmax*, int *kzmax*, bool *normalize* = true)

References L2Norm2().

```
2007                                                    {
2008   return sqrt(L2Norm2(u, kxmax, kzmax, normalize));
2009 }
```

Here is the call graph for this function:

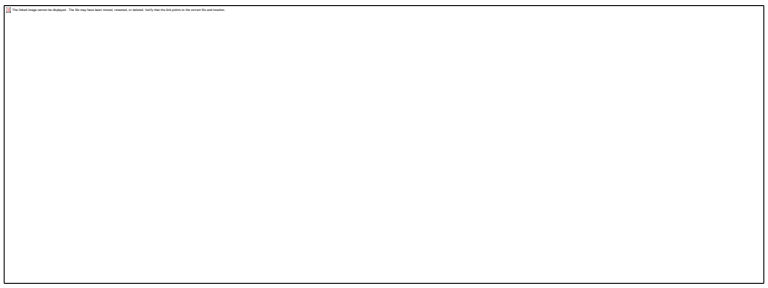


1.14.3.23 Real L2Norm (const FlowField & *f*, bool *normalize* = true)

References L2Norm2().

```
1930                                        {
1931   return sqrt(L2Norm2(u, normalize));
1932 }
```

Here is the call graph for this function:



1.14.3.24 Real L2Norm2 (const FlowField & *f*, int *kxmax*, int *kzmax*, bool *normalize* = true)

References FlowField::a(), abs(), FlowField::b(), FlowField::cmplx(), FlowField::kx(), FlowField::kz(), L2Norm2(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::Mz(), FlowField::Ny(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1981                                                    {
1982   assert(u.ystate()==Spectral);
1983   assert(u.xzstate()==Spectral);
1984   Real sum = 0.0;
```

```
1985
1986   // EFFICIENCY: would be better with smarter looping, but prob not worth trouble.
1987   ComplexChebyCoeff prof(u.Ny(), u.a(), u.b(), Spectral);
1988   for (int i=0; i<u.vectorDim(); ++i)
1989     for (int mx=0; mx<u.Mx(); ++mx) {
1990       if (abs(u.kx(mx)) > kxmax)
1991         continue;
1992       Real cz = 1.0; // cz = 2 for kz>0 to take account of kz<0 ghost modes
1993       for (int mz=0; mz<u.Mz(); ++mz) {
1994         if (abs(u.kz(mz)) > kzmax)
1995           continue;
1996         for (int ny=0; ny<u.Ny(); ++ny)
1997           prof.set(ny, u.cmplx(mx,ny,mz,i));
1998         sum += cz*L2Norm2(prof, normalize);
1999         cz = 2.0;
2000       }
2001     }
2002   if (!normalize)
2003     sum *= u.Lx()*u.Lz();
2004   return sum;
2005 }
```

Here is the call graph for this function:

1.14.3.25 Real L2Norm2 (const FlowField & *f*, bool *normalize* = true)

References FlowField::a(), FlowField::b(), FlowField::cmplx(), L2Norm2(), FlowField::Lx(), FlowField::Lz(), FlowField::Mx(), FlowField::Mz(), FlowField::Ny(), Spectral, FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1934                                        {
1935   assert(u.ystate() == Spectral);
1936   assert(u.xzstate()==Spectral);
1937   Real sum = 0.0;
1938   ComplexChebyCoeff prof(u.Ny(), u.a(), u.b(), Spectral);
1939   for (int i=0; i<u.vectorDim(); ++i)
1940     for (int mx=0; mx<u.Mx(); ++mx) {
1941       Real cz = 1.0; // cz = 2 for kz>0 to take account of kz<0 ghost modes
1942       for (int mz=0; mz<u.Mz(); ++mz) {
1943         for (int ny=0; ny<u.Ny(); ++ny)
1944           prof.set(ny, u.cmplx(mx,ny,mz,i));
1945         sum += cz*L2Norm2(prof, normalize);
1946         cz = 2.0;
1947       }
1948     }
1949   if (!normalize)
1950     sum *= u.Lx()*u.Lz();
1951   return sum;
1952 }
```

Here is the call graph for this function:

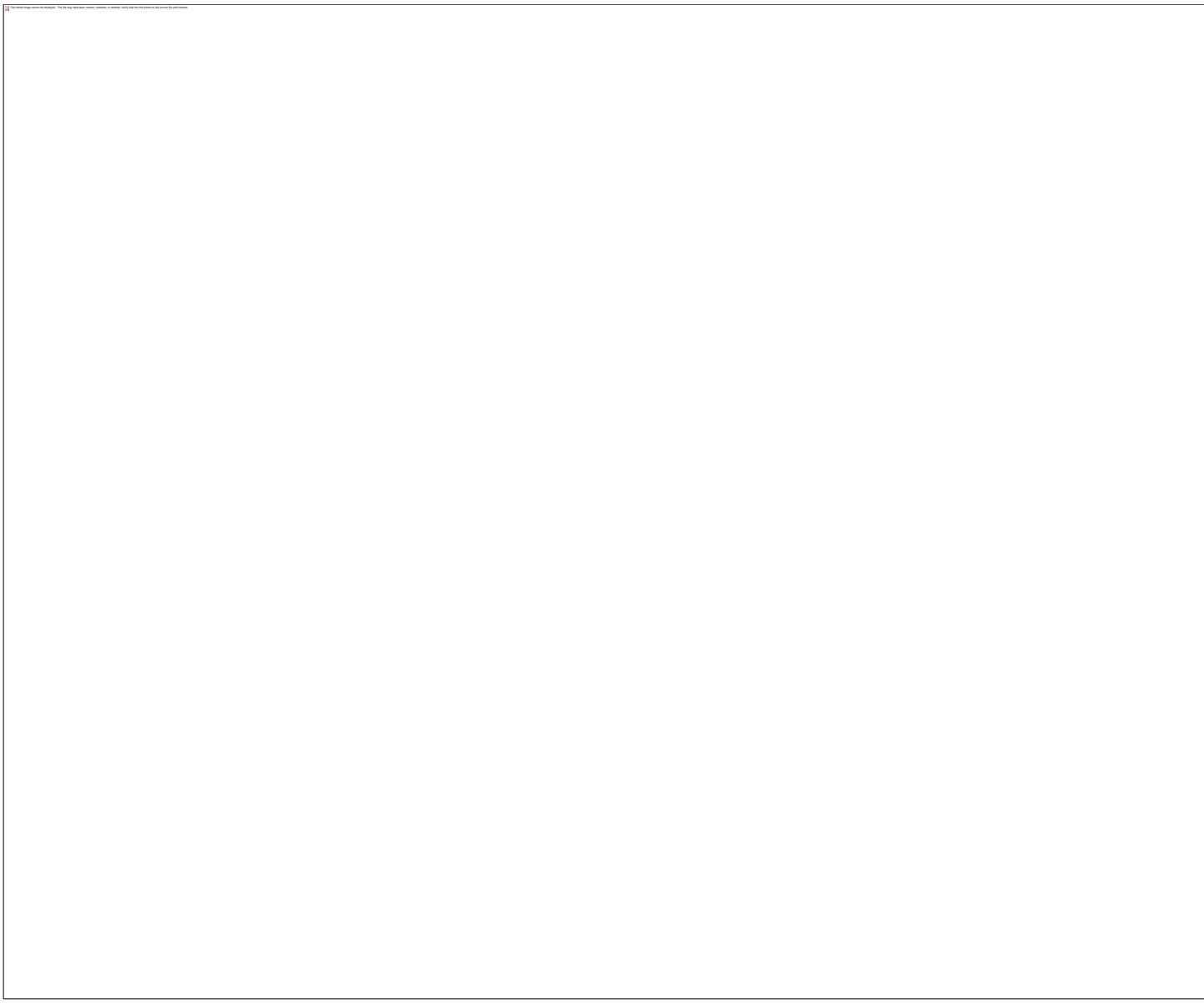
1.14.3.26 Real LinfDist (const FlowField & *f*, const FlowField & *g*)

References FlowField::congruent(), Greater(), LinfNorm(), ChebyCoeff::makePhysical(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), FlowField::profile(), Re(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1877                                        {
1878   assert(f.congruent(g));
1879   fieldstate fxstate = f.xzstate();
1880   fieldstate gxstate = g.xzstate();
1881   fieldstate fystate = f.ystate();
```

```
1882   fieldstate gystate = g.ystate();
1883
1884   ((FlowField&) f).makePhysical_xz();
1885   ((FlowField&) g).makePhysical_xz();
1886
1887   Real rtn = 0.0;
1888
1889   ChebyCoeff prof;
1890   ChebyTransform trans(f.Ny());
1891   for (int i=0; i<f.vectorDim(); ++i)
1892     for (int nx=0; nx<f.Nx(); ++nx)
1893       for (int nz=0; nz<f.Nz(); ++nz) {
1894         prof = Re(f.profile(nx,nz,i));
1895         prof -= Re(g.profile(nx,nz,i));
1896         prof.makePhysical();
1897         rtn = Greater(rtn, LinfNorm(prof));
1898       }
1899
1900   ((FlowField&) f).makeState(fxstate, fystate);
1901   ((FlowField&) g).makeState(gxstate, gystate);
1902
1903   return rtn;
1904 }
```

Here is the call graph for this function:



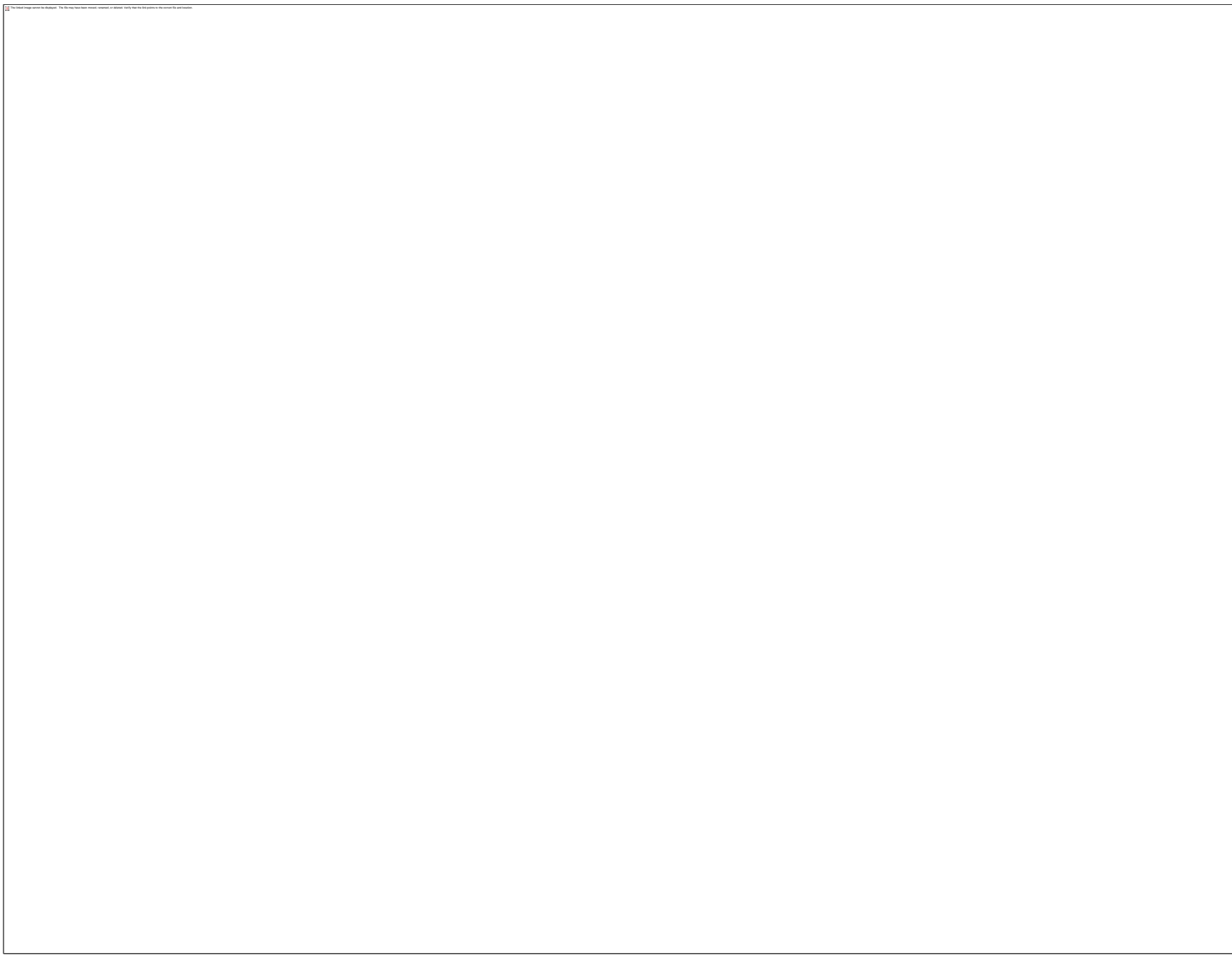
1.14.3.27 Real LinfNorm (const FlowField & *f*)

References Greater(), LinfNorm(), ChebyCoeff::makePhysical(), FlowField::Nx(), FlowField::Ny(), FlowField::Nz(), FlowField::profile(), Re(), FlowField::vectorDim(), FlowField::xzstate(), and FlowField::ystate().

```
1855                                {
1856   fieldstate xstate = u.xzstate();
1857   fieldstate ystate = u.ystate();
1858   ((FlowField&) u).makePhysical_xz();
1859
1860   Real rtn = 0.0;
```

```
1861
1862   ChebyCoeff prof;
1863   ChebyTransform trans(u.Ny());
1864   for (int i=0; i<u.vectorDim(); ++i)
1865     for (int nx=0; nx<u.Nx(); ++nx)
1866       for (int nz=0; nz<u.Nz(); ++nz) {
1867         prof = Re(u.profile(nx,nz,i));
1868         prof.makePhysical(trans);
1869         rtn = Greater(rtn, LinfNorm(prof));
1870       }
1871
1872   ((FlowField&) u).makeState(xstate, ystate);
1873
1874   return rtn;
1875 }
```

Here is the call graph for this function:



1.14.3.28 void randomProfile (ComplexChebyCoeff & *u*, ComplexChebyCoeff & *v*, ComplexChebyCoeff & *w*, int *kx*, int *kz*, Real *Lx*, Real *Lz*, Real *mag*, Real *spectralDecay*)

References ComplexChebyCoeff::a(), abs(), ComplexChebyCoeff::b(), diff(), div(), divNorm(), EPSILON, ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), I(), ComplexChebyCoeff::im, L2Norm(), ComplexChebyCoeff::length(), ComplexChebyCoeff::numModes(), pi, randomUprofile(), randomVprofile(), ComplexChebyCoeff::setState(), ComplexChebyCoeff::setToZero(), ChebyCoeff::setToZero(), and Spectral.

Referenced by FlowField::addPerturbation().

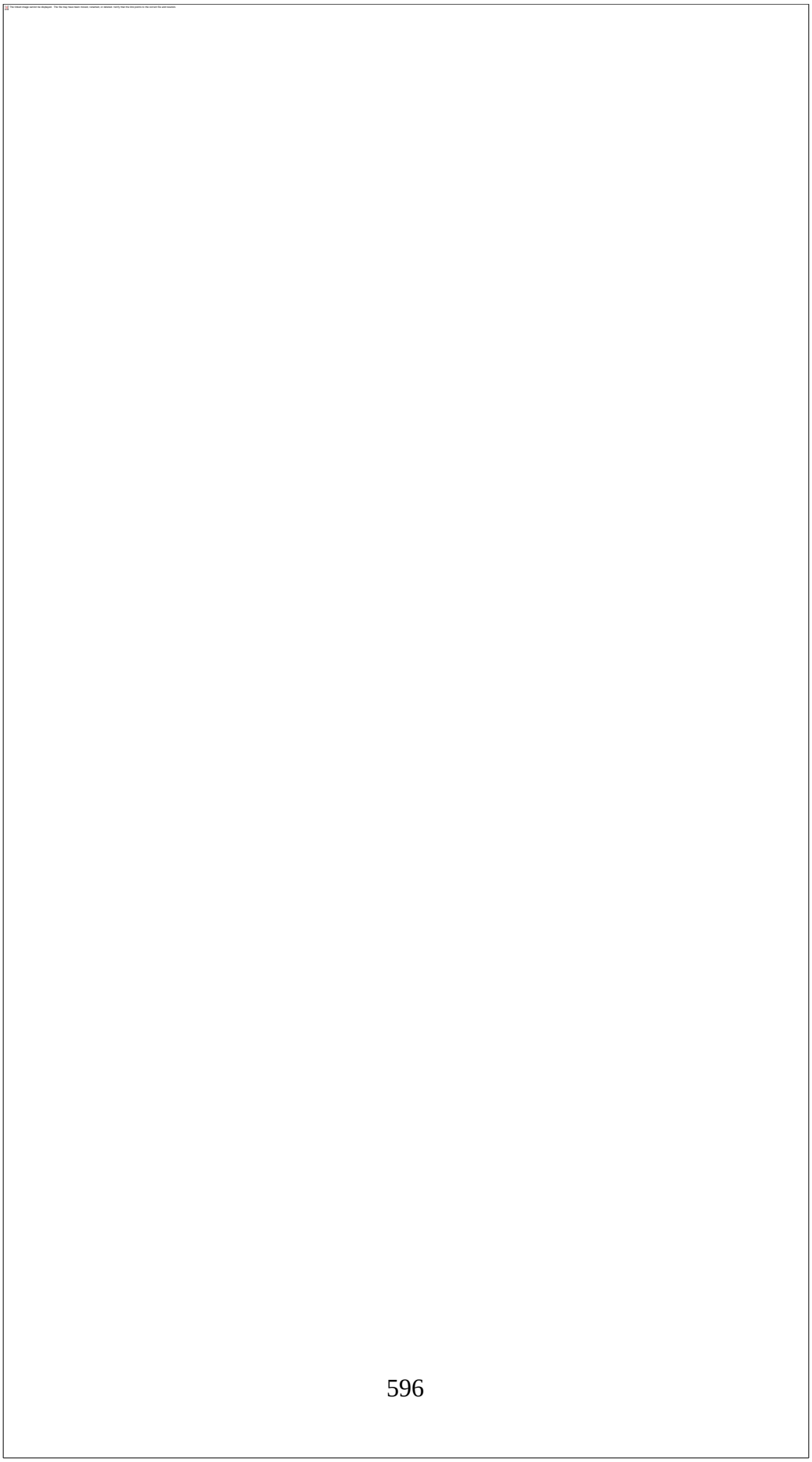
```
2223
{
```

```
2224
2225  int N = u.length();
2226  ChebyTransform trans(N);
2227  u.setState(Spectral);
2228  v.setState(Spectral);
2229  w.setState(Spectral);
2230  //u.setToZero();
2231  //v.setToZero();
2232  //w.setToZero();
2233  if (kx == 0 && kz == 0) {
2234    // Assign an odd perturbation to u, so as not to change mean(U).
2235    // Just set even modes to zero.
2236
2237    randomUprofile(w, mag, decay);
2238    w.im.setToZero();
2239
2240    randomUprofile(u, mag, decay);
2241    u.im.setToZero();
2242
2243    v.setToZero();
2244
2245  }
2246  else {
2247    // Other kx,kz cases are based on a random v(y).
2248
2249    randomVprofile(v, mag, decay);
2250    ComplexChebyCoeff vy = diff(v);
2251
2252    if (kx == 0) {
2253      randomUprofile(u, mag, decay);
2254      u.im.setToZero();
2255      w = vy;
2256      w *= -Lz/((2*pi*kz)*I);
2257    }
2258    else if (kz == 0) {
2259      randomUprofile(w, mag, decay);
2260      w.im.setToZero();
2261      u = vy;
2262      u *= -Lx/((2*pi*kx)*I);
2263    }
2264    else {
2265
2266      ComplexChebyCoeff v0(v);
2267      ComplexChebyCoeff v1(v);
2268      randomVprofile(v0, mag, decay);
2269      randomVprofile(v1, mag, decay);
```

```
2270
2271      ComplexChebyCoeff v0y = diff(v0);
2272      ComplexChebyCoeff v1y = diff(v1);
2273
2274      // Finally, the general case, where kx, kz != 0 and u,v,w are nonzero
2275      // Set a random u(y)
2276      ComplexChebyCoeff u0(v.numModes(), v.a(), v.b(), Spectral);
2277      ComplexChebyCoeff w0(v.numModes(), v.a(), v.b(), Spectral);
2278      ComplexChebyCoeff u1(v.numModes(), v.a(), v.b(), Spectral);
2279      ComplexChebyCoeff w1(v.numModes(), v.a(), v.b(), Spectral);
2280
2281      randomUprofile(u0, mag, decay);
2282
2283      // Calculate w0 from div u0 == u0x + v0y + w0z == 0.
2284      ComplexChebyCoeff u0x(u0);
2285      u0x *= (2*pi*kx/Lx)*I;
2286      w0 = v0y;
2287      w0 += u0x;
2288      w0 *= -Lz/((2*pi*kz)*I);       // Set w = -Lz/(2*pi*I*kz) * (ux + vy);
2289
2290
2291      //randomUprofile(w1, mag, decay);
2292
2293      // Calculate u0 from div u0 == u0x + v0y + w0z == 0.
2294      ComplexChebyCoeff w1z(w1);
2295      w1z *= (2*pi*kz/Lz)*I;
2296      u1 = v1y;
2297      u1 += w1z;
2298      u1 *= -Lx/((2*pi*kx)*I);       // Set w = -Lz/(2*pi*I*kz) * (ux + vy);
2299
2300      u = u0;
2301      v = v0;
2302      w = w0;
2303      u += u1;
2304      v += v1;
2305      w += w1;
2306    }
2307  }
2308
2309  // Check divergence
2310  ComplexChebyCoeff ux(u);
2311  ux *= (2*pi*kx/Lx)*I;
2312  ComplexChebyCoeff wz(w);
2313  wz *= (2*pi*kz/Lz)*I;
2314  ComplexChebyCoeff vy = diff(v);
2315
```

```
2316   ComplexChebyCoeff div(ux);
2317   div += vy;
2318   div += wz;
2319
2320   Real divNorm = L2Norm(div);
2321   Real ubcNorm = abs(u.eval_a()) + abs(u.eval_b());
2322   Real vbcNorm = abs(v.eval_a()) + abs(v.eval_b());
2323   Real wbcNorm = abs(w.eval_a()) + abs(w.eval_b());
2324   assert(divNorm < EPSILON);
2325   assert(ubcNorm < EPSILON);
2326   assert(vbcNorm < EPSILON);
2327   assert(wbcNorm < EPSILON);
2328   // supress unused-variable compiler warnings...
2329   wbcNorm += divNorm + ubcNorm + vbcNorm;
2330
2331 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

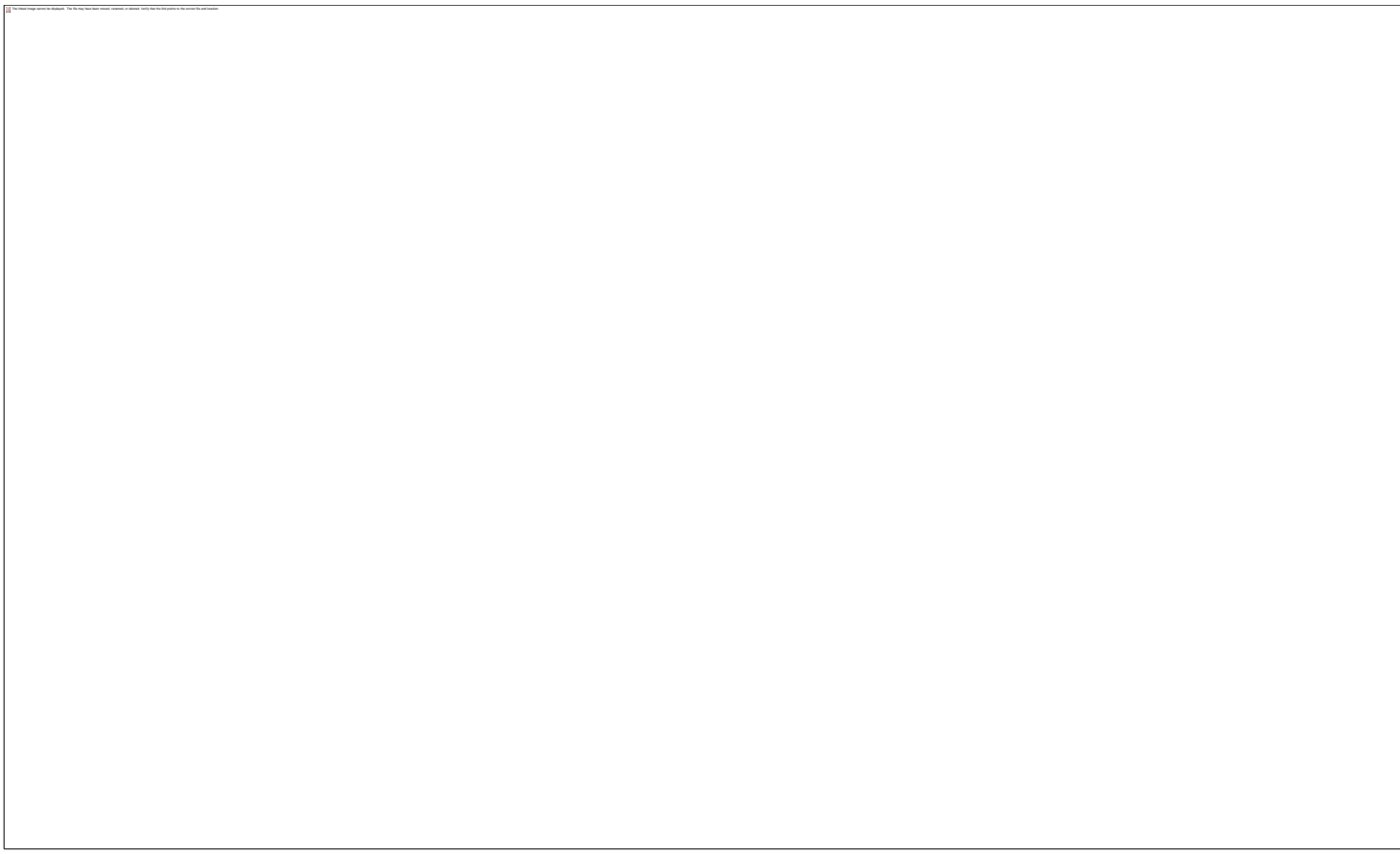
1.14.3.29 void randomUprofile (ComplexChebyCoeff & *u*, Real *mag*, Real *spectralDecay*)

References abs(), EPSILON, ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), ComplexChebyCoeff::length(), randomComplex(), ComplexChebyCoeff::set(), ComplexChebyCoeff::setState(), Spectral, and ComplexChebyCoeff::sub().

Referenced by randomProfile().

```
2058                                                {
2059   // Set a random u(y)
2060   int N = u.length();
2061   u.setState(Spectral);
2062   int n; // MSVC++ FOR-SCOPE BUG
2063   for (n=0; n<N; ++n) {
2064     u.set(n, mag*randomComplex());
2065     mag *= decay;
2066   }
2067   ChebyTransform trans(N);
2068   // Adjust u(y) so that u(+-1) == 0
2069   //cout << "before random u(a) == " << u.eval_a() << endl;
2070   //cout << "before random u(b) == " << u.eval_b() << endl;
2071   Complex u0 = (u.eval_b() + u.eval_a())/2.0;
2072   Complex u1 = (u.eval_b() - u.eval_a())/2.0;
2073   u.sub(0, u0);
2074   u.sub(1, u1);
2075   //cout << "after random u(a) == " << u.eval_a() << endl;
2076   //cout << "after random u(b) == " << u.eval_b() << endl;
2077
2078   assert(abs(u.eval_b()) < EPSILON);
2079   assert(abs(u.eval_a()) < EPSILON);
2080 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.14.3.30 void randomVprofile (ComplexChebyCoeff & *v*, Real *mag*, Real *spectralDecay*)

References ComplexChebyCoeff::a(), ComplexChebyCoeff::b(), diff(), ComplexChebyCoeff::eval_a(), ComplexChebyCoeff::eval_b(), Greater(), ComplexChebyCoeff::length(), ComplexChebyCoeff::numModes(), randomComplex(), ComplexChebyCoeff::set(), ComplexChebyCoeff::setState(), Spectral, and ComplexChebyCoeff::sub().

Referenced by randomProfile().

```
2082                                                {
2083   int N = v.length();
2084   v.setState(Spectral);
2085   // Assign a random v(y).
2086   v.set(0, 0.0);
```

```
2087  v.set(1, 0.0);
2088  v.set(2, 0.0);
2089  v.set(3, 0.0);
2090  int n; // MSVC++ FOR-SCOPE BUG
2091  for (n=0; n<4; ++n)
2092    v.set(n, 0.0);
2093  for (n=4; n<N-2; ++n) {
2094    v.set(n, mag*randomComplex());
2095    mag *= decay;
2096  }
2097
2098  for (n=Greater(N-2, 0); n<N; ++n)
2099    v.set(n, 0.0);
2100
2101  // Adjust v so that v(+-1) == v'(+/-1) == 0, by subtracting off
2102  // s0 T0(x) + s1 T1(x) + s2 T2(x) + s3 T3(x), with s's chosen to
2103  // have same BCs as v.
2104  ComplexChebyCoeff vy = diff(v);
2105
2106  Complex a = v.eval_a();
2107  Complex b = v.eval_b();
2108  Complex c = vy.eval_a();
2109  Complex d = vy.eval_b();
2110
2111  // The numercial coeffs are inverse of the matrix (values found with Maple)
2112  // T0(-1)  T1(-1)  T2(-1)  T3(-1)     s0      a
2113  // T0(1)   T1(1)   T2(1)   T3(1)      s1      b
2114  // T0'(-1) T1'(-1) T2'(-1) T3'(-1)    s2  ==  c
2115  // T0'(1)  T1'(1)  T2'(1)  T3'(1)     s3      d
2116
2117  Complex s0 = 0.5*(a + b) + 0.125*(c - d);
2118  Complex s1 = 0.5625*(b - a) - 0.0625*(c + d);
2119  Complex s2 = 0.125*(d - c);
2120  Complex s3 = 0.0625*(a - b + c + d);
2121
2122  ComplexChebyCoeff adj(v.numModes(), v.a(), v.b(), Spectral);
2123  adj.set(0, s0);
2124  adj.set(1, s1);
2125  adj.set(2, s2);
2126  adj.set(3, s3);
2127  ComplexChebyCoeff adjy = diff(adj);
2128
2129
2130  // Subtract off the coeffs
2131  v.sub(0, s0);
2132  v.sub(1, s1);
```

```
2133   v.sub(2, s2);
2134   v.sub(3, s3);
2135
2136   diff(v,vy);
2137
2138   //cout << "random v(a)  == " << v.eval_a() << endl;
2139   //cout << "random v(b)  == " << v.eval_b()  << endl;
2140   //cout << "random v'(a) == " << vy.eval_a() << endl;
2141   //cout << "random v'(b) == " << vy.eval_b() << endl;
2142
2143 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.14.3.31 void swap (FlowField & *f*, FlowField & *v*)

References FlowField::cdata_, FlowField::congruent(), FlowField::rdata_, FlowField::xz_iplan_, FlowField::xz_plan_, and FlowField::y_plan_.

```
1676                                    {
1677   assert(f.congruent(g));
1678   Real* tmp = f.rdata_;
1679   f.rdata_ = g.rdata_;
1680   g.rdata_ = tmp;
1681
1682   f.cdata_ = (Complex*)f.rdata_;
1683   g.cdata_ = (Complex*)g.rdata_;
1684
1685   // We must swap the FFTW3 plans, as well, since they're tied to specific
1686   // memory locations. This burned me when I upgraded FFTW2->FFTW3.
1687   fftw_plan tmp_plan;
1688   tmp_plan = f.xz_plan_;
1689   f.xz_plan_ = g.xz_plan_;
1690   g.xz_plan_ = tmp_plan;
1691
1692   tmp_plan = f.xz_iplan_;
1693   f.xz_iplan_ = g.xz_iplan_;
1694   g.xz_iplan_ = tmp_plan;
1695
1696   tmp_plan = f.y_plan_;
1697   f.y_plan_ = g.y_plan_;
1698   g.y_plan_ = tmp_plan;
1699 }
```

Here is the call graph for this function:

1.14.3.32 int zero_last_mode (int *k*, int *kmax*, int *n*) [inline]

Referenced by curl(), diff(), div(), FlowField::Dx(), FlowField::Dz(), grad(), linearizedNL(), rotationalNL(), skewsymmetricNL_task3::Run(), FlowField::saveDivSpectrum(), skewsymmetricNL(), skewsymmetricNL_GPU(), xdiff(), and zdiff().

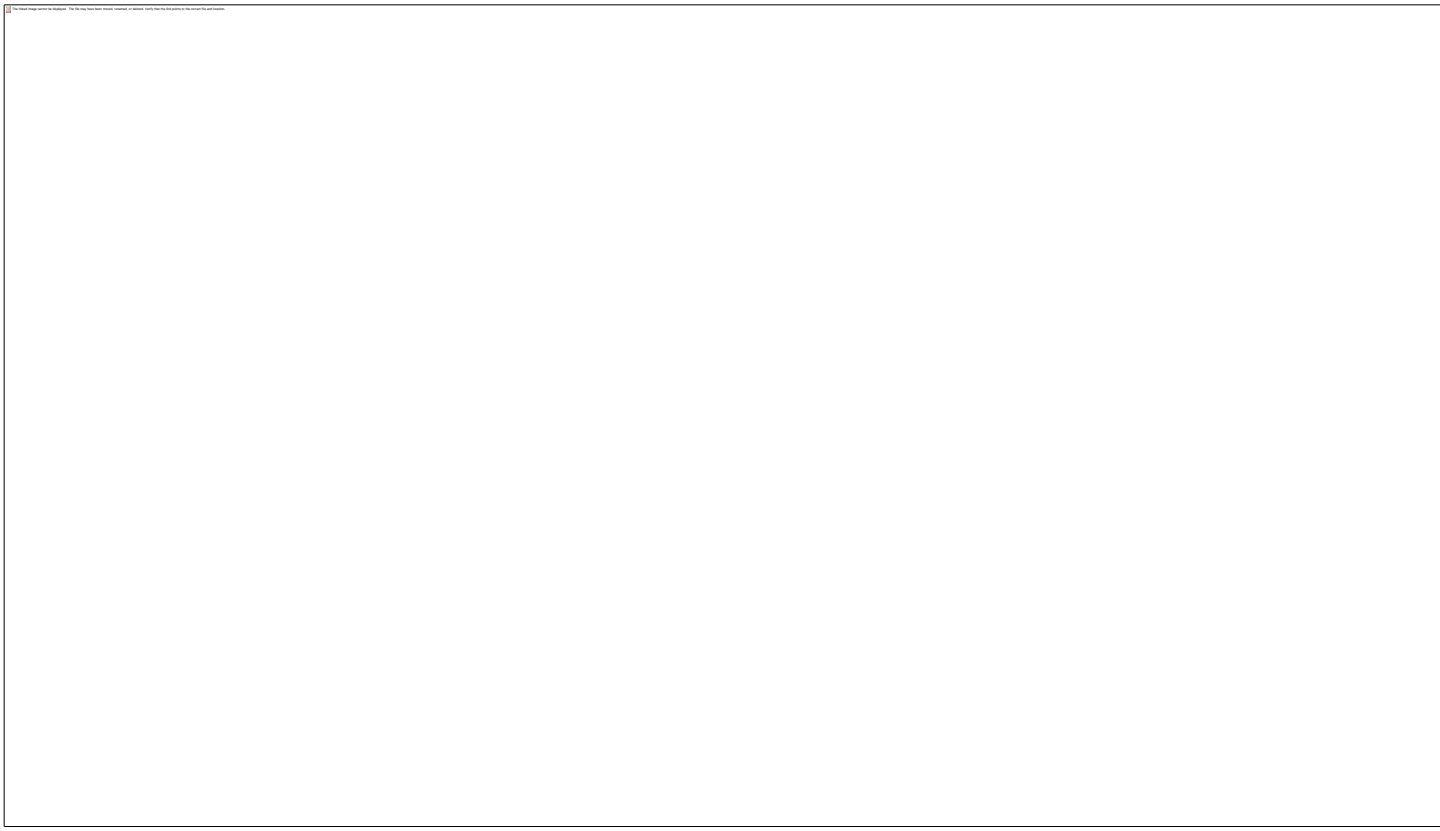
```
503                                     {
504   return ((k == kmax) && (n%2 == 1)) ? 0 : 1;
505 }
```

Here is the caller graph for this function:

1.14.3.33

1.15 /_cuda/channelflow-0.9.22/channelflow/helmholtz.cpp File Reference

#include "channelflow/helmholtz.h"
#include "channelflow/chebyshev.h"
Include dependency graph for helmholtz.cpp:



1.15.1 Variables

- const Real EPSILON = 1.0

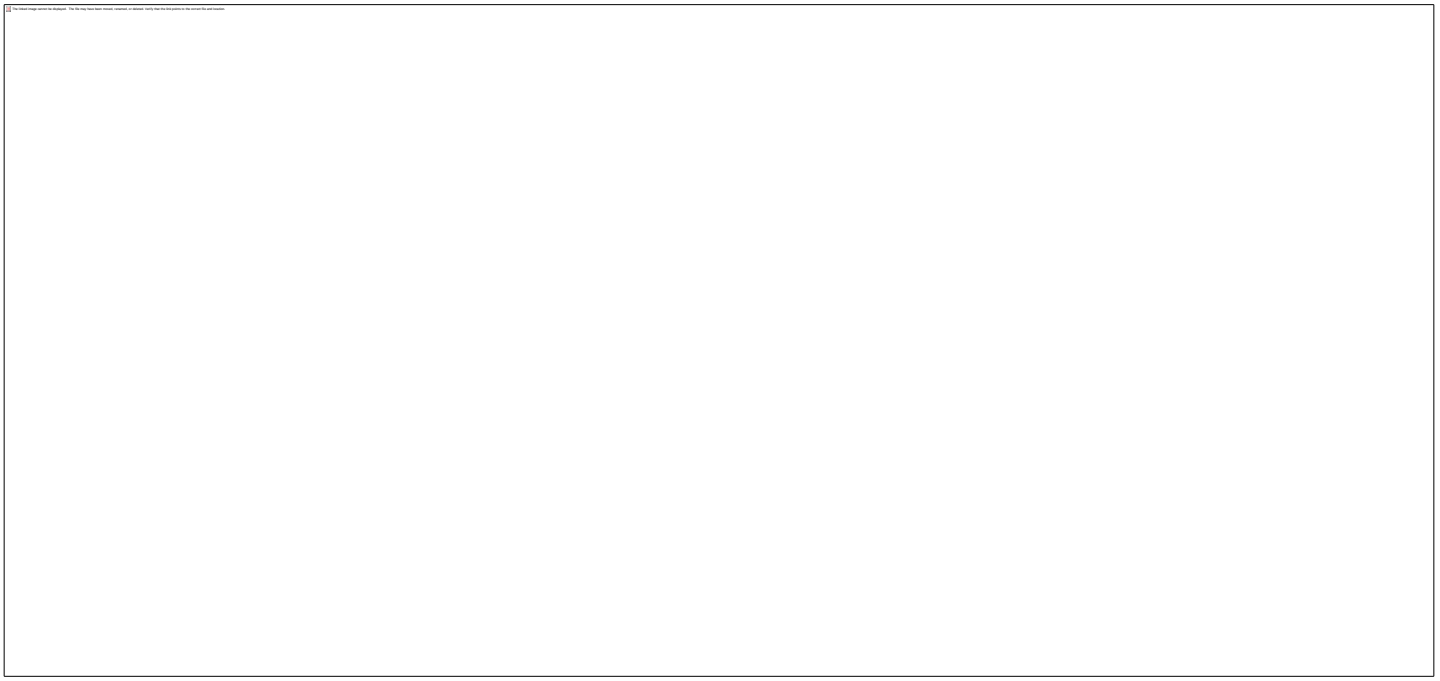
1.15.2 Variable Documentation

1.15.2.1 const Real EPSILON = 1.0

1.15.2.2

1.16 /_cuda/channelflow-0.9.22/channelflow/helmholtz.h File Reference

#include "channelflow/mathdefs.h"
#include "channelflow/bandedtridiag.h"
#include "channelflow/chebyshev.h"
Include dependency graph for helmholtz.h:



This graph shows which files directly or indirectly include this file:

1.16.1 Classes

- class HelmholtzSolver

1.17 /_cuda/channelflow-0.9.22/channelflow/mathdefs.cpp File Reference

#include <stdlib.h>
#include <stdio.h>
#include <fstream>
#include <iomanip>
#include <arpa/inet.h>
#include "channelflow/mathdefs.h"
#include "config.h"
#include <string.h>
Include dependency graph for mathdefs.cpp:

1.17.1 Defines

- #define drand48() rand()/(double(RAND_MAX));

1.17.2 Functions

- string i2s (int n)
- void cferror (const string &message)
- void cfpause ()
- Real pythag (Real a, Real b)
- bool isPowerOfTwo (int N)
- int roundReal (Real x)
- int pow (int x, int n)
- void save (Complex c, const string &filebase)
- void load (Complex &c, const string &filebase)
- void save (Real c, const string &filebase)
- void load (Real &c, const string &filebase)
- void write (ostream &os, bool b)
- void read (istream &is, bool &b)
- void write (ostream &os, fieldstate s)
- void read (istream &is, fieldstate &s)
- void write (ostream &os, Complex z)
- void read (istream &is, Complex &z)
- void write (ostream &os, int n)
- void read (istream &is, int &n)
- void write (ostream &os, Real x)

- void read (istream &is, Real &x)
- Real randomReal ()
- Complex randomComplex ()
- ostream & operator<< (ostream &os, Complex z)
- ostream & operator<< (ostream &os, fieldstate s)
- istream & operator>> (istream &is, fieldstate &s)
- void channelflowVersion (int &major, int &minor, int &update)

1.17.3 Define Documentation

1.17.3.1 #define drand48() rand()/(double(RAND_MAX));

Referenced by randomReal().

1.17.4 Function Documentation

1.17.4.1 void cferror (const string & *message*)

Referenced by bisectSearch(), diff(), div(), FlowField::Dx(), FlowField::Dz(), grad(), FieldSeries::interpolate(), load(), makeroll(), navierstokesNL(), polyInterp(), pow(), save(), secantSearch(), xdiff(), and zdiff().

```
61                              {
62   cerr << message << endl;
63   exit(-1);
64 }
```

Here is the caller graph for this function:

1.17.4.2 void cfpause ()

```
66              {
67   cout << "cfpause..." << flush;
68   char s[10];
```

```
69   cin.getline(s,10);
70 }
```

1.17.4.3 void channelflowVersion (int & *major*, int & *minor*, int & *update*)

Referenced by FlowField::binarySave().

```
391                                          {
392   const char delimiters[] = ".";
393   char* cp = strdup(PACKAGE_VERSION);  // Make writable copy of version str
394   assert(cp != NULL);
395   major  = atoi(strtok(cp, delimiters));
396   minor  = atoi(strtok(NULL, delimiters));
397   update = atoi(strtok(NULL, delimiters));
398   free(cp);
399 }
```

Here is the caller graph for this function:

1.17.4.4 string i2s (int *n*)

References abs().

```
49                {
50   const int Nbuf = 32;
51   char buff[Nbuf];
52   if (n >= 0)
53     sprintf(buff, "%d", n);
54   else {
55     n = abs(n);
56     sprintf(buff, "_%d", n);
57   }
58   return string(buff);
59 }
```

Here is the call graph for this function:

1.17.4.5 bool isPowerOfTwo (int *N*)

```
82                      {
83   int onbits = 0;
84   for (unsigned int i = 0; i<int(8*sizeof(int)); ++i) {
85     onbits += N % 2;
86     N = N >> 1;
87     if (onbits > 1)
88       return false;
89   }
90   return true;
91 }
```

1.17.4.6 void load (Real & *c*, const string & *filebase*)

References cferror().

```
168                                     {
169   string filename = filebase + string(".real");
170   ifstream is(filename.c_str());
171   if (!is.good())
172     cferror("load(Real, filebase) :  can't open file " + filename);
173   Real r;
174   is >> r;
175   c = r;
176 }
```

Here is the call graph for this function:

1.17.4.7 void load (Complex & *c*, const string & *filebase*)

References cferror().

```
147                                        {
148   string filename = filebase + string(".cmplx");
149   ifstream is(filename.c_str());
150   if (!is.good())
151     cferror("load(Complex, filebase) :  can't open file " + filename);
152   Real a, b;
153   is >> a >> b;
154   c = Complex(a,b);
155 }
```

Here is the call graph for this function:

1.17.4.8 ostream& operator<< (ostream & *os*, fieldstate *s*)

References Spectral.

```
366                                         {
367   os << ((s==Spectral) ? 'S' : 'P');
368   return os;
369 }
```

1.17.4.9 ostream& operator<< (ostream & *os*, Complex *z*)

References Im(), and Re().

```
361                                         {
362   os << '(' << Re(z) << ", " << Im(z) << ')';
363   return os;
364 }
```

Here is the call graph for this function:

1.17.4.10 istream& operator>> (istream & *is*, fieldstate & *s*)

References Physical, and Spectral.

```
371                                         {
372   char c = ' ';
373   while (c == ' ')
374     is >> c;
375   switch(c) {
376   case 'P':
377     s = Physical;
378     break;
379   case 'S':
```

```
380     s = Spectral;
381     break;
382   default:
383     cerr << "read fieldstate error: unknown fieldstate " << c << endl;
384     s = Spectral;
385     assert(false);
386   }
387   return is;
388 }
```

1.17.4.11 int pow (int *x*, int *n*)

References cferror().

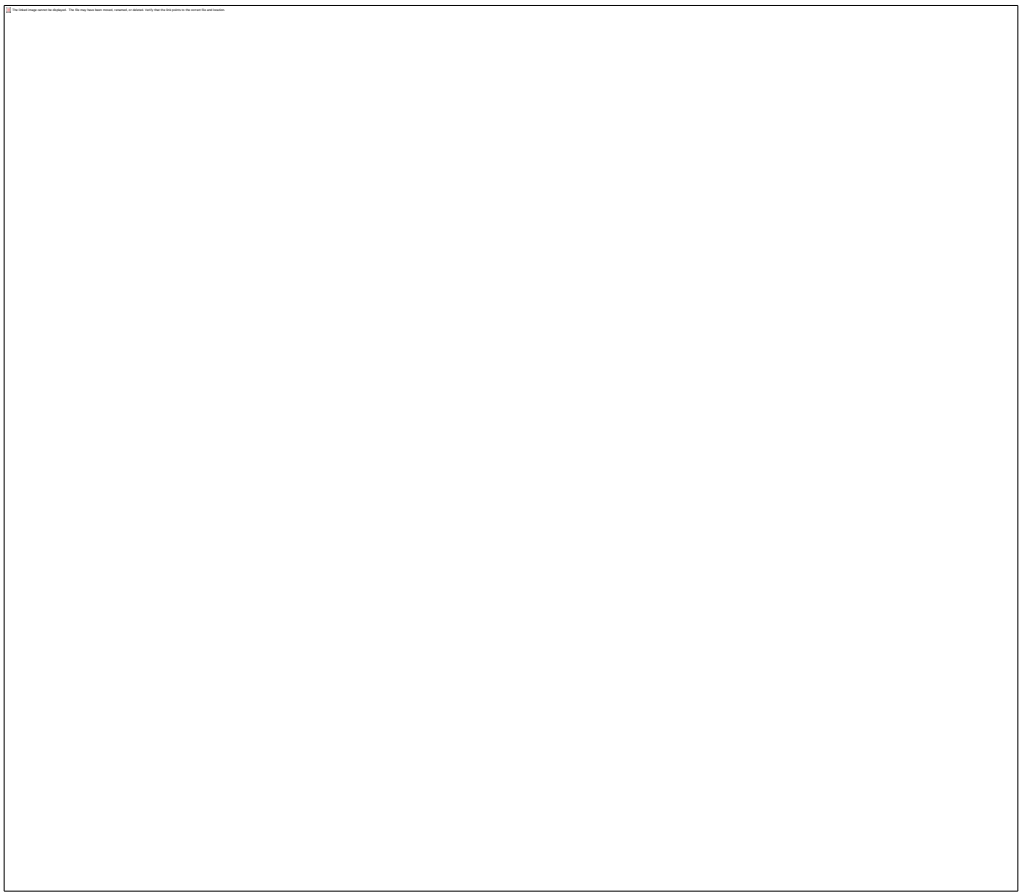
Referenced by FlowField::addPerturbations(), chebyUprofile(), chebyVprofile(), diff(), FlowField::Dx(), FlowField::Dz(), makeroll(), xdiff(), and zdiff().

```
102                     {
103   if (n < 0)
104     cferror("int pow(int, int) : can't do negative exponents, use Real pow(Real,int)");
105   switch(n) {
106   case 0:
107     return 1;
108   case 1:
109     return x;
110   case 2:
111     return x*x;
112   case 3:
113     return x*x*x;
114   case 4: {
115     int xx = x*x;
116     return xx*xx;
117   }
118   default: {
119     // a*x^n is invariant and n is decreasing in this loop
120     int a = 1;
121     while (n>0) {
122       if (n%2 == 0) {
123         x *= x; //     x -> x^2
124         n /= 2; //     n -> n/2
125       }         // a x^n -> a (x^2)^(n/2) = a x^n
126       else {
127         a *= x; //     a -> ax
128         --n;    //     n -> n-1
129       }         // a x^n -> (a x) x^(n-1) = a x^n
130     }
131     return a;
```

```
132   }
133   }
134 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.17.4.12 Real pythag (Real *a*, Real *b*)

References square().

```
72                          {
73   Real absa = fabs(a);
74   Real absb = fabs(b);
75
76   if (absa > absb)
```

```
77     return absa * (sqrt(1.0 + square(absb/absa)));
78   else
79     return (absb == 0.0) ? 0.0 : absb * (sqrt (1.0 + square(absa / absb)));
80 }
```

Here is the call graph for this function:

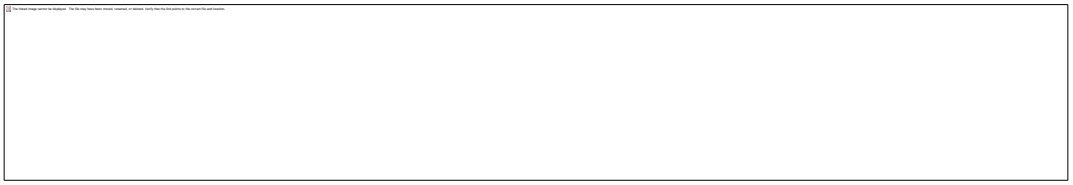
1.17.4.13 Complex randomComplex ()

References log(), and randomReal().

Referenced by randomUprofile(), and randomVprofile().

```
349                     {
350   Real a;
351   Real b;
352   Real r2;
353   do {
354     a = randomReal();
355     b = randomReal();
356     r2 = a * a + b * b;
357   } while (r2 >= 1.0 || r2 == 0.0);
358   return sqrt(-log(r2) / r2) * Complex(a,b);
359 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

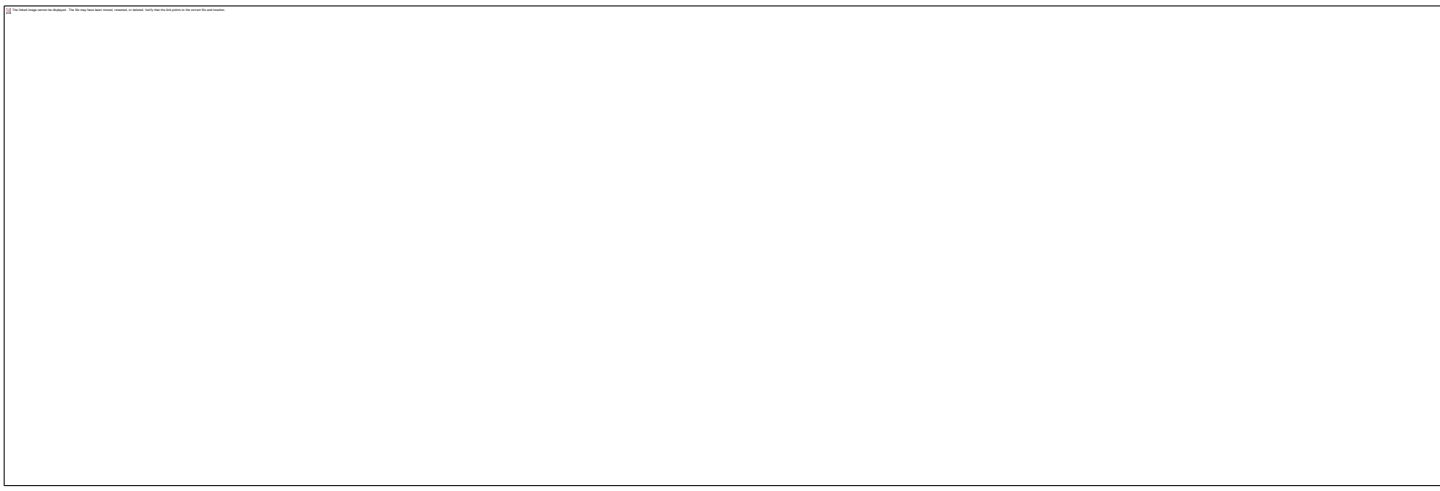
1.17.4.14 Real randomReal ()

References drand48.

Referenced by chebyUprofile(), chebyVprofile(), randomComplex(), Vector::randomize(), and ChebyCoeff::randomize().

```
345                 {
346   return 2*drand48() - 1.0;
347 }
```

Here is the caller graph for this function:



1.17.4.15 void read (istream & *is*, Real & *x*)

```
277                         {
278
279 #if SIZEOF_DOUBLE != 8
280   cerr << "read(istream& is, Real x) error: channelflow binary IO assumes 64-bit double-
precision floating point.\n";
281   cerr << "The double on this platform is " << 8*SIZEOF_DOUBLE << " bits" << endl;
282   exit(1);
283 #endif
284
285 #if SIZEOF_INT != 4
286   cerr << "write(ostream& os, Real x) error: channelflow binary IO assumes 64-bit
double-precision floating point and 32-bit ints.\n";
287   cerr << "The int on this platform is " << 8*SIZEOF_INT << " bits" << endl;
288   exit(1);
289 #endif
290
291 #if __BYTE_ORDER == __LITTLE_ENDIAN
```

```
292   int i[2];
293
294   // Read in two reversed and byteswapped 32bit ints
295   is.read((char*)(i), sizeof(double));
296
297   // Byteswap each 32 bit int and reverse their order
298   int tmp = i[1];
299   i[1] = ntohl(i[0]);
300   i[0] = ntohl(tmp);
301
302   // Copy the two consecutive 32bit ints into the Real
303   memcpy(&x, i, sizeof(double));
304 #else
305   is.read((char*) &x, sizeof(double));
306 #endif
307 }
```

1.17.4.16 void read (istream & *is*, int & *n*)

```
232                          {
233
234 #if SIZEOF_INT != 4
235   cerr << "read(istream& is, int n) error: channelflow binary IO assumes 32-bit int.\n";
236   cerr << "The int on this platform is " << 8*SIZEOF_INT << " bits" << endl;
237   exit(1);
238 #endif
239
240   is.read((char*)(&n), sizeof(int));
241   n = ntohl(n);
242 }
```

1.17.4.17 void read (istream & *is*, Complex & *z*)

References read().

```
209                             {
210   Real a, b;
211   read(is, a);
212   read(is, b);
213   z = Complex(a, b);
214 }
```

Here is the call graph for this function:

1.17.4.18 void read (istream & *is*, fieldstate & *s*)

References Physical, and Spectral.

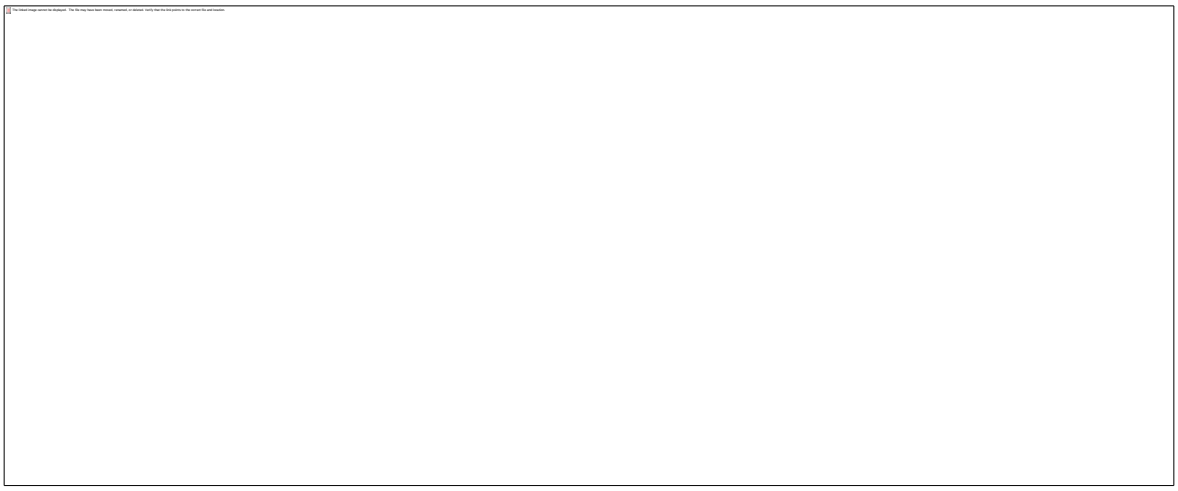
```
196                                {
197   char c;
198   is.read(&c, 1);
199   s = (c=='S') ? Spectral : Physical;
200 }
```

1.17.4.19 void read (istream & *is*, bool & *b*)

Referenced by ChebyCoeff::binaryLoad(), BasisFunc::binaryLoad(), array< T >::binaryLoad(), FlowField::FlowField(), and read().

```
185                                {
186   char c;
187   is.read(&c, 1);
188   b = (c=='0') ? false : true;
189 }
```

Here is the caller graph for this function:



1.17.4.20 int roundReal (Real *x*)

References abs().

```
93                     {
94   int ix = int(x);
95   Real dm = abs(x-(ix-1)); // m==minus:  distance x to (xcast minus 1)
96   Real dc = abs(x-ix);     // c==center: distance x to (xcast)
97   Real dp = abs(x-(ix+1)); // p==plus:   distance x to (xcast plus 1)
98
99   return ((dm<dp) ? ((dm<dc) ? ix-1 : ix) : ((dc<dp) ? ix : ix+1));
100 }
```

Here is the call graph for this function:

1.17.4.21 void save (Real *c*, const string & *filebase*)

References cferror().

```
157                                     {
158   string extension(".real");
159   string filename = filebase + extension;
160   ofstream os(filename.c_str());
161   if (!os.good())
162     cferror("save(Real, filebase) :  can't open file " + filename);
163   os << setprecision(17);
164   os << c << '\n';      // format can be read by matlab.
165   os.close();
166 }
```

Here is the call graph for this function:

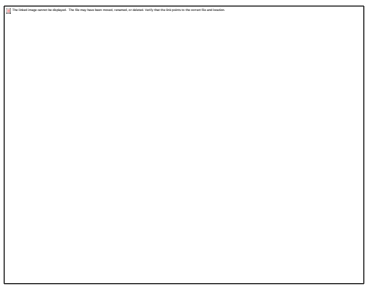
1.17.4.22 void save (Complex *c*, const string & *filebase*)

References cferror(), Im(), and Re().

```
136                                       {
137   string extension(".cmplx");
138   string filename = filebase + extension;
139   ofstream os(filename.c_str());
140   if (!os.good())
141     cferror("save(Complex, filebase) :  can't open file " + filename);
142   os << setprecision(17);
```

```
143   os << Re(c) << ' ' << Im(c) << '\n';  // format can be read by matlab.
144   os.close();
145 }
```

Here is the call graph for this function:



1.17.4.23 void write (ostream & *os*, Real *x*)

```
245                         {
246
247 #if SIZEOF_DOUBLE != 8
248   cerr << "write(ostream& os, Real x) error: channelflow binary IO assumes 64-bit
double-precision floating point.\n";
249   cerr << "The double on this platform is " << 8*SIZEOF_DOUBLE << " bits" << endl;
250   exit(1);
251 #endif
252
253 #if SIZEOF_INT != 4
254   cerr << "write(ostream& os, Real x) error: channelflow binary IO assumes 64-bit
double-precision floating point and 32-bit ints.\n";
255   cerr << "The int on this platform is " << 8*SIZEOF_INT << " bits" << endl;
256   exit(1);
257 #endif
258
259 #if __BYTE_ORDER == __LITTLE_ENDIAN
260   int i[2];
261
262   // Copy 64 bit x into two consecutive 32 bit ints
263   memcpy(i, &x, sizeof(uint64_t));
264
265   // Byteswap each 32 bit int and reverse their order.
266   int tmp = i[1];
267   i[1] = htonl(i[0]);
268   i[0] = htonl(tmp);
269
270   // Write out the two reversed and byteswapped 32bit ints
```

```
271   os.write((char*) i, sizeof(double));
272 #else
273   os.write((char*) &x, sizeof(double));
274 #endif
275 }
```

1.17.4.24 void write (ostream & *os*, int *n*)

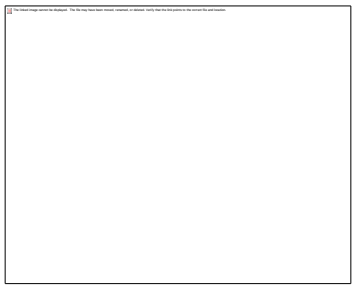
```
220                          {
221
222 #if SIZEOF_INT != 4
223   cerr << "write(ostream& os, int n) :  channelflow binary IO assumes 32-bit int\n";
224   cerr << "The int on this platform is " << 8*SIZEOF_INT << " bits" << endl;
225   exit(1);
226 #endif
227
228   n = htonl(n);
229   os.write((char*) &n, sizeof(int));
230 }
```

1.17.4.25 void write (ostream & *os*, Complex *z*)

References Im(), Re(), and write().

```
202                          {
203   Real x = Re(z);
204   write(os, x);
205   x = Im(z);
206   write(os, x);
207 }
```

Here is the call graph for this function:



1.17.4.26 void write (ostream & *os*, fieldstate *s*)

References Spectral.

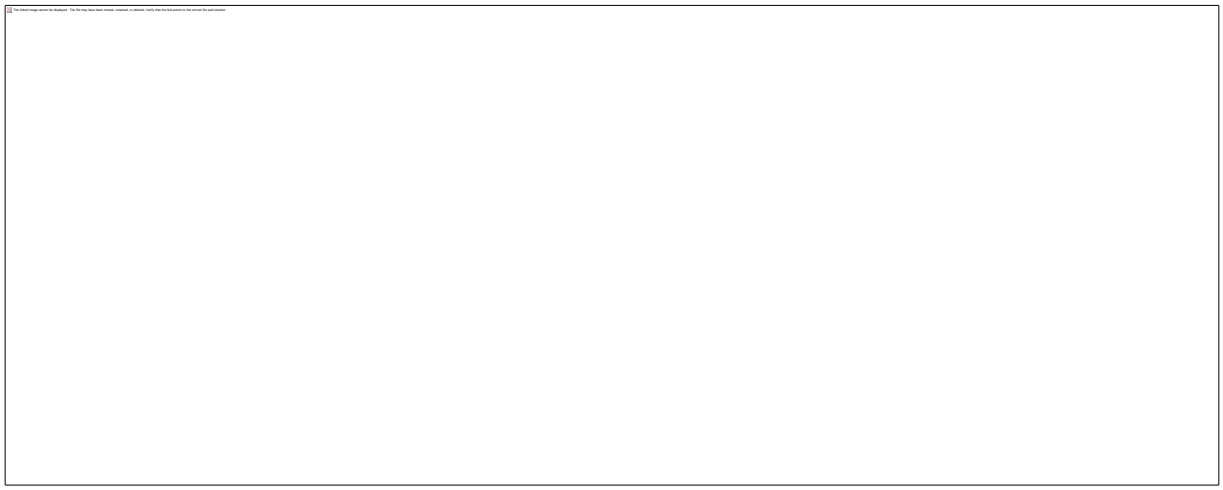
```
191                                 {
192   char c = (s == Spectral) ? 'S' : 'P';
193   os.write(&c, 1);
194 }
```

1.17.4.27 void write (ostream & *os*, bool *b*)

Referenced by ChebyCoeff::binaryDump(), BasisFunc::binaryDump(), array< T >::binaryDump(), FlowField::binarySave(), and write().

```
180                                 {
181   char c = (b) ? '1' : '0';
182   os.write(&c, 1);  // sizeof(char) == 1
183 }
```

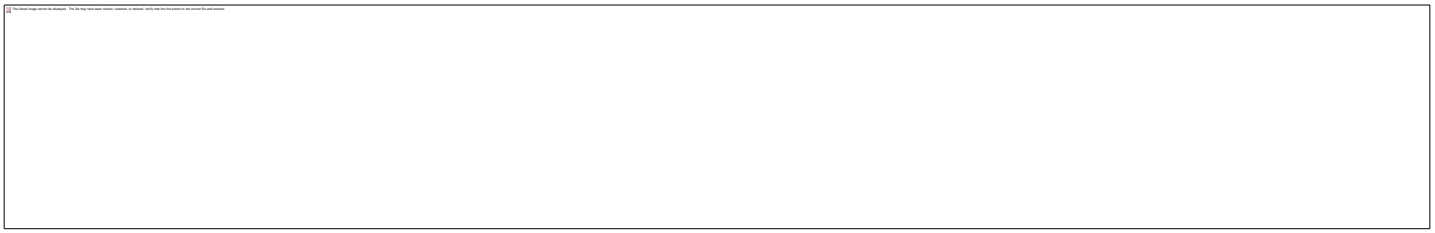
Here is the caller graph for this function:



1.17.4.28

1.18 /_cuda/channelflow-0.9.22/channelflow/mathdefs.h File Reference

#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <iostream>
#include <complex>
#include <string>
Include dependency graph for mathdefs.h:



This graph shows which files directly or indirectly include this file:

1.18.1 Defines

- #define M_PI 3.14159265358979323846

1.18.2 Typedefs

- typedef double Real
- typedef std::complex< double > Complex

1.18.3 Enumerations

- enum fieldstate { Physical, Spectral }
- enum parity { Even, Odd }

1.18.4 Functions

- const Complex I (0.0, 1.0)
- std::string i2s (int n)
- void cferror (const std::string &message)
- void cfpause ()
- int kronecker (int m, int n)
- int square (int x)
- int cube (int x)

- Real square (Real x)
- Real cube (Real x)
- Real nr_sign (Real a, Real b)
- void swap (int &a, int &b)
- void swap (Real &a, Real &b)
- void swap (Complex &a, Complex &b)
- int Greater (int a, int b)
- int lesser (int a, int b)
- Real Greater (Real a, Real b)
- Real lesser (Real a, Real b)
- int roundReal (Real x)
- int pow (int n, int p)
- Real pythag (Real a, Real b)
- bool isPowerOfTwo (int n)
- void save (Real c, const std::string &filebase)
- void load (Real &c, const std::string &filebase)
- void save (Complex c, const std::string &filebase)
- void load (Complex &c, const std::string &filebase)
- void write (std::ostream &os, int n)
- void write (std::ostream &os, bool b)
- void write (std::ostream &os, Real x)
- void write (std::ostream &os, Complex z)
- void write (std::ostream &os, fieldstate s)
- void read (std::istream &is, int &n)
- void read (std::istream &is, bool &b)
- void read (std::istream &is, Real &x)
- void read (std::istream &is, Complex &z)
- void read (std::istream &os, fieldstate &s)
- void readLittleEndian (std::istream &is, int &n)
- void readLittleEndian (std::istream &is, Real &x)
- Real Re (const Complex &z)
- Real Im (const Complex &z)
- Real abs2 (const Complex &z)
- Real randomReal ()
- Complex randomComplex ()
- std::ostream & operator<< (std::ostream &os, Complex z)
- std::ostream & operator<< (std::ostream &os, fieldstate f)
- std::istream & operator>> (std::istream &is, fieldstate &f)
- void channelflowVersion (int &major, int &minor, int &update)

1.18.5 Variables

- const int REAL_DIGITS = 17
- const int REAL_IOWIDTH = 24
- const Real pi = M_PI

1.18.6 Define Documentation

1.18.6.1 #define M_PI 3.14159265358979323846

Referenced by gaussLegendreQuadrature().

1.18.7 Typedef Documentation

1.18.7.1 typedef std::complex<double> Complex

1.18.7.2 typedef double Real

1.18.8 Enumeration Type Documentation

1.18.8.1 enum fieldstate

Enumerator:

Physical
Spectral

```
46 { Physical, Spectral };
```

1.18.8.2 enum parity

Enumerator:

Even
Odd

```
47 { Even, Odd };
```

1.18.9 Function Documentation

1.18.9.1 Real abs2 (const Complex & *z*) [inline]

References norm().

Referenced by bcDist2(), BasisFunc::bcNorm(), bcNorm2(), divcheck(), FlowField::saveDivSpectrum(), and TauSolver::verify().

```
192                                 {
193   return norm(z); // NOTE std lib norm is misnamed! Returns a^2+b^2 for a+bi.
194 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.18.9.2 void cferror (const std::string & *message*)

Referenced by bisectSearch(), diff(), div(), FlowField::Dx(), FlowField::Dz(), grad(), FieldSeries::interpolate(), load(), makeroll(), navierstokesNL(), polyInterp(), pow(), save(), secantSearch(), xdiff(), and zdiff().

```
61                                  {
62   cerr << message << endl;
63   exit(-1);
64 }
```

Here is the caller graph for this function:

1.18.9.3 void cfpause ()

```
66              {
67   cout << "cfpause..." << flush;
68   char s[10];
```

```
69   cin.getline(s,10);
70 }
```

1.18.9.4 void channelflowVersion (int & *major*, int & *minor*, int & *update*)

Referenced by FlowField::binarySave().

```
391                                          {
392   const char delimiters[] = ".";
393   char* cp = strdup(PACKAGE_VERSION);  // Make writable copy of version str
394   assert(cp != NULL);
395   major  = atoi(strtok(cp, delimiters));
396   minor  = atoi(strtok(NULL, delimiters));
397   update = atoi(strtok(NULL, delimiters));
398   free(cp);
399 }
```

Here is the caller graph for this function:

1.18.9.5 Real cube (Real *x*) [inline]

```
144                       {
145   return x * x * x;
146 }
```

1.18.9.6 int cube (int *x*) [inline]

```
154                       {
155   return x * x * x;
156 }
```

1.18.9.7 Real Greater (Real *a*, Real *b*) [inline]

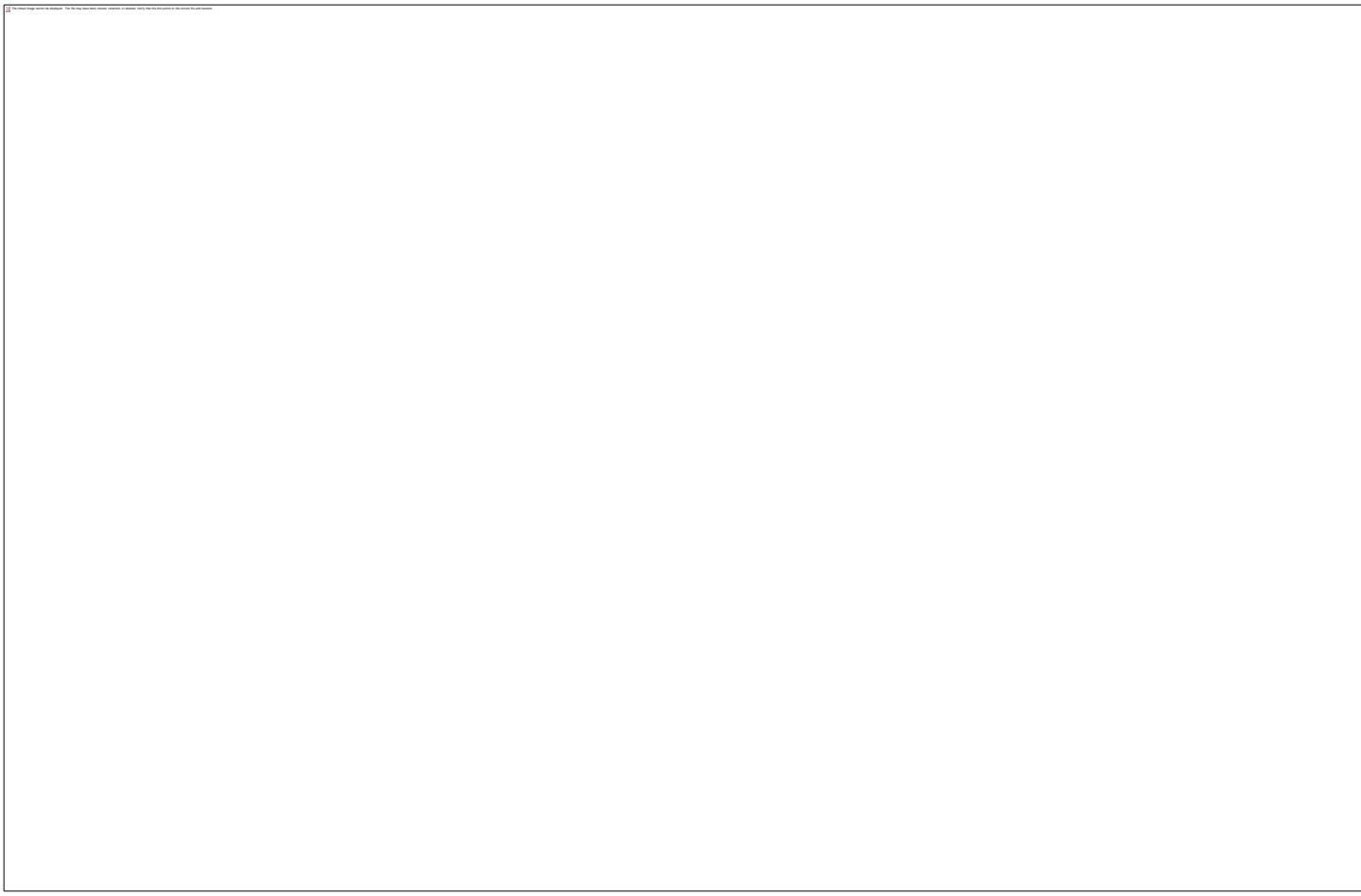
```
180                                 {
181   return (a > b) ? a : b;
182 }
```

1.18.9.8 int Greater (int *a*, int *b*) [inline]

Referenced by FlowField::addPerturbations(), FlowField::CFLfactor(), LinfDist(), LinfNorm(), randomVprofile(), TauSolver::TauSolver(), and HelmholtzSolver::verify().

```
174                             {
175   return (a > b) ? a : b;
176 }
```

Here is the caller graph for this function:



1.18.9.9 const Complex I (0. *0*, 1. *0*)

Referenced by assignOrrSommField(), chebyProfile(), chebyUprofile(), chebyVprofile(), curl(), divcheck(), divDist2(), divergence(), divNorm2(), gradient(), ComplexChebyCoeff::operator[](), BasisFunc::randomize(), randomProfile(), TauSolver::solve(), TauSolver::verify(), xdiff(), and zdiff().

Here is the caller graph for this function:

1.18.9.10 std::string i2s (int *n*)

References abs().

```
49                {
50   const int Nbuf = 32;
51   char buff[Nbuf];
52   if (n >= 0)
53     sprintf(buff, "%d", n);
54   else {
55     n = abs(n);
56     sprintf(buff, "_%d", n);
57   }
58   return string(buff);
59 }
```

Here is the call graph for this function:

1.18.9.11 Real Im (const Complex & *z*) [inline]

```
189                          {
190   return z.imag();
191 }
```

1.18.9.12 bool isPowerOfTwo (int *n*)

```
82                       {
83   int onbits = 0;
84   for (unsigned int i = 0; i<int(8*sizeof(int)); ++i) {
85     onbits += N % 2;
86     N = N >> 1;
87     if (onbits > 1)
88       return false;
89   }
90   return true;
91 }
```

1.18.9.13 int kronecker (int *m*, int *n*) [inline]

```
136                             {
137   return (m==n) ? 1 : 0;
138 }
```

1.18.9.14 Real lesser (Real *a*, Real *b*) [inline]

```
183                             {
184   return (a < b) ? a : b;
185 }
```

1.18.9.15 int lesser (int *a*, int *b*) [inline]

Referenced by FlowField::addPerturbations(), ChebyCoeff::ChebyCoeff(), ChebyCoeff::fill(), FlowField::profile(), and Vector::resize().

```
177                             {
178   return (a < b) ? a : b;
179 }
```

Here is the caller graph for this function:

1.18.9.16 void load (Complex & *c*, const std::string & *filebase*)

References cferror().

```
147                                     {
148   string filename = filebase + string(".cmplx");
149   ifstream is(filename.c_str());
150   if (!is.good())
151     cferror("load(Complex, filebase) :  can't open file " + filename);
152   Real a, b;
153   is >> a >> b;
154   c = Complex(a,b);
155 }
```

Here is the call graph for this function:

1.18.9.17 void load (Real & *c*, const std::string & *filebase*)

References cferror().

```
168                                     {
169   string filename = filebase + string(".real");
170   ifstream is(filename.c_str());
171   if (!is.good())
172     cferror("load(Real, filebase) :  can't open file " + filename);
173   Real r;
174   is >> r;
175   c = r;
176 }
```

Here is the call graph for this function:

1.18.9.18 Real nr_sign (Real *a*, Real *b*) [inline]

```
147                                     {
148   return (b >= 0.0) ? fabs (a) : -fabs (a);
149 }
```

1.18.9.19 std::ostream& operator<< (std::ostream & *os*, fieldstate *f*)

References Spectral.

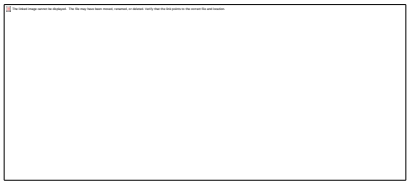
```
366                                         {
367   os << ((s==Spectral) ? 'S' : 'P');
368   return os;
369 }
```

1.18.9.20 std::ostream& operator<< (std::ostream & *os*, Complex *z*)

References Im(), and Re().

```
361                                         {
362   os << '(' << Re(z) << ", " << Im(z) << ')';
363   return os;
364 }
```

Here is the call graph for this function:



1.18.9.21 std::istream& operator>> (std::istream & *is*, fieldstate & *f*)

References Physical, and Spectral.

```
371                                         {
372   char c = ' ';
373   while (c == ' ')
374     is >> c;
375   switch(c) {
376   case 'P':
377     s = Physical;
378     break;
379   case 'S':
380     s = Spectral;
381     break;
382   default:
383     cerr << "read fieldstate error: unknown fieldstate " << c << endl;
384     s = Spectral;
```

```
385     assert(false);
386   }
387   return is;
388 }
```

1.18.9.22 int pow (int *n*, int *p*)

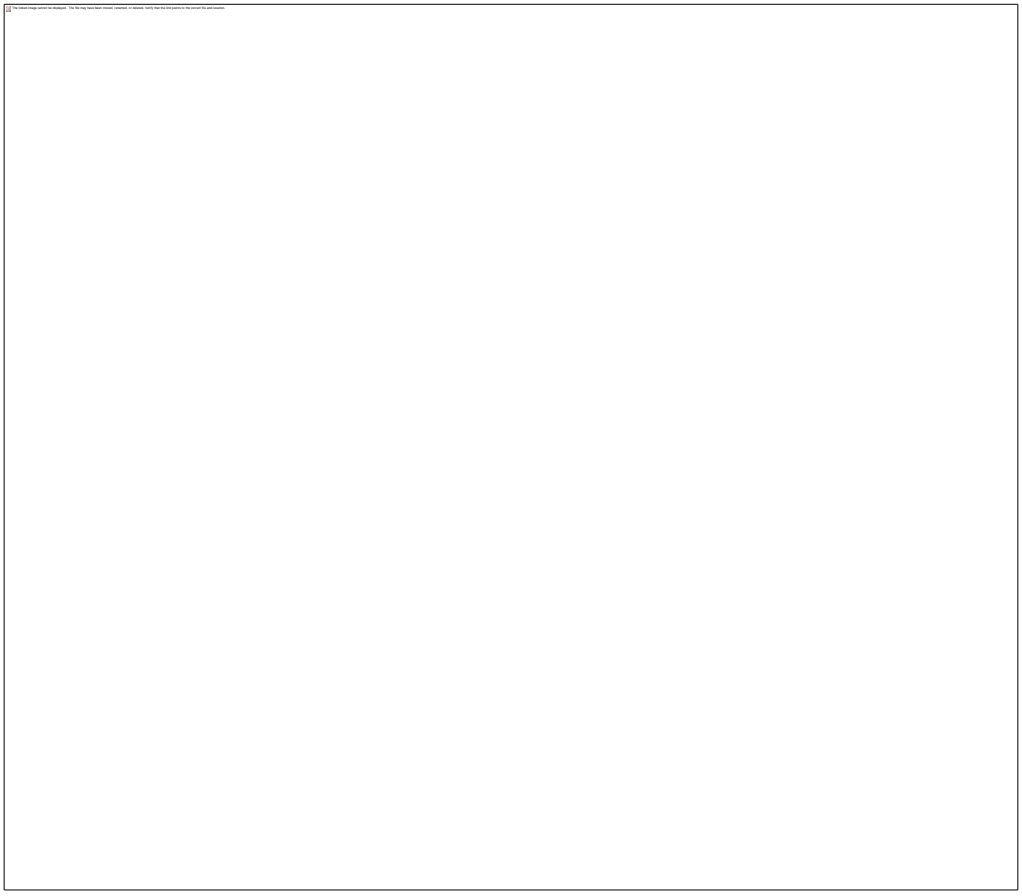
References cferror().

Referenced by FlowField::addPerturbations(), chebyUprofile(), chebyVprofile(), diff(), FlowField::Dx(), FlowField::Dz(), makeroll(), xdiff(), and zdiff().

```
102                     {
103   if (n < 0)
104     cferror("int pow(int, int) : can't do negative exponents, use Real pow(Real,int)");
105   switch(n) {
106   case 0:
107     return 1;
108   case 1:
109     return x;
110   case 2:
111     return x*x;
112   case 3:
113     return x*x*x;
114   case 4: {
115     int xx = x*x;
116     return xx*xx;
117   }
118   default: {
119     // a*x^n is invariant and n is decreasing in this loop
120     int a = 1;
121     while (n>0) {
122       if (n%2 == 0) {
123         x *= x; //     x -> x^2
124         n /= 2; //     n -> n/2
125       }         // a x^n -> a (x^2)^(n/2) = a x^n
126       else {
127         a *= x; //     a -> ax
128         --n;    //     n -> n-1
129       }         // a x^n -> (a x) x^(n-1) = a x^n
130     }
131     return a;
132   }
133   }
134 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



1.18.9.23 Real pythag (Real *a*, Real *b*)

References square().

```
72                          {
73   Real absa = fabs(a);
74   Real absb = fabs(b);
75
76   if (absa > absb)
77     return absa * (sqrt(1.0 + square(absb/absa)));
78   else
79     return (absb == 0.0) ? 0.0 : absb * (sqrt (1.0 + square(absa / absb)));
80 }
```

Here is the call graph for this function:

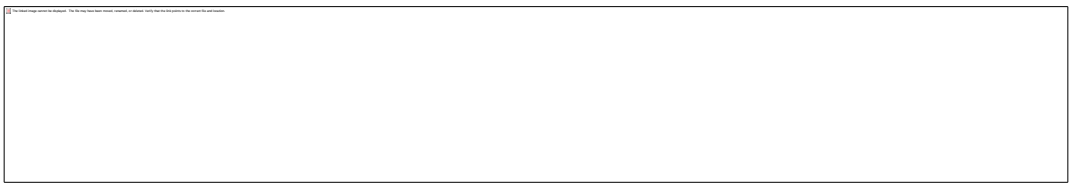
1.18.9.24 Complex randomComplex ()

References log(), and randomReal().

Referenced by randomUprofile(), and randomVprofile().

```
349                     {
350   Real a;
351   Real b;
352   Real r2;
353   do {
354     a = randomReal();
355     b = randomReal();
356     r2 = a * a + b * b;
357   } while (r2 >= 1.0 || r2 == 0.0);
358   return sqrt(-log(r2) / r2) * Complex(a,b);
359 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

1.18.9.25 Real randomReal ()

References drand48.

Referenced by chebyUprofile(), chebyVprofile(), randomComplex(), Vector::randomize(), and ChebyCoeff::randomize().

```
345                 {
```

```
346   return 2*drand48() - 1.0;
347 }
```

Here is the caller graph for this function:

1.18.9.26 Real Re (const Complex & *z*) [inline]

```
186                             {
187   return z.real();
188 }
```

1.18.9.27 void read (std::istream & *os*, fieldstate & *s*)

References Physical, and Spectral.

```
196                               {
197   char c;
198   is.read(&c, 1);
199   s = (c=='S') ? Spectral : Physical;
200 }
```

1.18.9.28 void read (std::istream & *is*, Complex & *z*)

References read().

```
209                             {
210   Real a, b;
211   read(is, a);
212   read(is, b);
213   z = Complex(a, b);
214 }
```

Here is the call graph for this function:

1.18.9.29 void read (std::istream & *is*, Real & *x*)

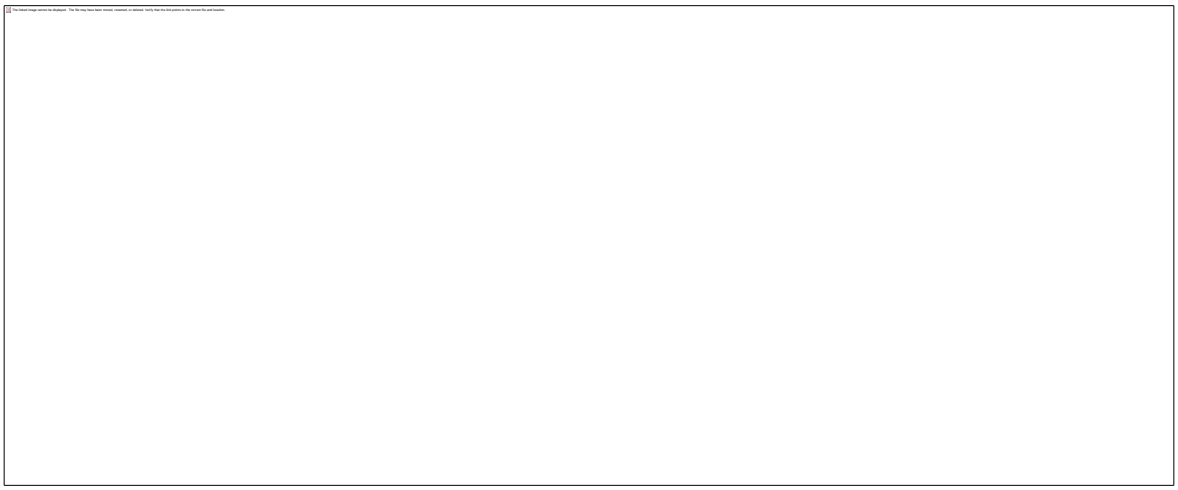
```
277                         {
278
279 #if SIZEOF_DOUBLE != 8
280   cerr << "read(istream& is, Real x) error: channelflow binary IO assumes 64-bit double-
precision floating point.\n";
281   cerr << "The double on this platform is " << 8*SIZEOF_DOUBLE << " bits" << endl;
282   exit(1);
283 #endif
284
285 #if SIZEOF_INT != 4
286   cerr << "write(ostream& os, Real x) error: channelflow binary IO assumes 64-bit
double-precision floating point and 32-bit ints.\n";
287   cerr << "The int on this platform is " << 8*SIZEOF_INT << " bits" << endl;
288   exit(1);
289 #endif
290
291 #if __BYTE_ORDER == __LITTLE_ENDIAN
292   int i[2];
293
294   // Read in two reversed and byteswapped 32bit ints
295   is.read((char*)(i), sizeof(double));
296
297   // Byteswap each 32 bit int and reverse their order
298   int tmp = i[1];
299   i[1] = ntohl(i[0]);
300   i[0] = ntohl(tmp);
301
302   // Copy the two consecutive 32bit ints into the Real
303   memcpy(&x, i, sizeof(double));
304 #else
305   is.read((char*) &x, sizeof(double));
306 #endif
307 }
```

1.18.9.30 void read (std::istream & *is*, bool & *b*)

Referenced by ChebyCoeff::binaryLoad(), BasisFunc::binaryLoad(), array< T >::binaryLoad(), FlowField::FlowField(), and read().

```
185                         {
186   char c;
187   is.read(&c, 1);
188   b = (c=='0') ? false : true;
189 }
```

Here is the caller graph for this function:



1.18.9.31 void read (std::istream & *is*, int & *n*)

```
232                             {
233
234 #if SIZEOF_INT != 4
235   cerr << "read(istream& is, int n) error: channelflow binary IO assumes 32-bit int.\n";
236   cerr << "The int on this platform is " << 8*SIZEOF_INT << " bits" << endl;
237   exit(1);
238 #endif
239
240   is.read((char*)(&n), sizeof(int));
241   n = ntohl(n);
242 }
```

1.18.9.32 void readLittleEndian (std::istream & *is*, Real & *x*)

1.18.9.33 void readLittleEndian (std::istream & *is*, int & *n*)

1.18.9.34 int roundReal (Real *x*)

References abs().

```
93                      {
94   int ix = int(x);
95   Real dm = abs(x-(ix-1)); // m==minus:  distance x to (xcast minus 1)
96   Real dc = abs(x-ix);     // c==center: distance x to (xcast)
97   Real dp = abs(x-(ix+1)); // p==plus:   distance x to (xcast plus 1)
98
```

```
99   return ((dm<dp) ? ((dm<dc) ? ix-1 : ix) : ((dc<dp) ? ix : ix+1));
100 }
```

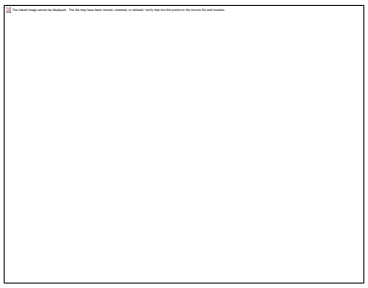
Here is the call graph for this function:

1.18.9.35 void save (Complex *c*, const std::string & *filebase*)

References cferror(), Im(), and Re().

```
136                                     {
137   string extension(".cmplx");
138   string filename = filebase + extension;
139   ofstream os(filename.c_str());
140   if (!os.good())
141     cferror("save(Complex, filebase) :  can't open file " + filename);
142   os << setprecision(17);
143   os << Re(c) << ' ' << Im(c) << '\n';  // format can be read by matlab.
144   os.close();
145 }
```

Here is the call graph for this function:



1.18.9.36 void save (Real *c*, const std::string & *filebase*)

References cferror().

```
157                                     {
158   string extension(".real");
159   string filename = filebase + extension;
160   ofstream os(filename.c_str());
161   if (!os.good())
162     cferror("save(Real, filebase) :  can't open file " + filename);
163   os << setprecision(17);
```

```
164   os << c << '\n';      // format can be read by matlab.
165   os.close();
166 }
```

Here is the call graph for this function:

1.18.9.37 Real square (Real *x*) [inline]

```
141                       {
142   return x * x;
143 }
```

1.18.9.38 int square (int *x*) [inline]

Referenced by CNABstyleDNS::advance(), RungeKuttaDNS::advance(), assignOrrSommField(), changeBaseFlow(), chebyDist2(), chebyNorm2(), energy(), HelmholtzSolver::HelmholtzSolver(), L2Dist(), L2Dist2(), L2Norm(), L2Norm2(), lapl(), laplacian(), TurbStats::msave(), norm(), norm2(), PoissonSolver::PoissonSolver(), pythag(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), MultistepDNS::reset_dt(), up2q(), uq2p(), and TurbStats::uu().

```
151                       {
152   return x * x;
153 }
```

Here is the caller graph for this function:

1.18.9.39 void swap (Complex & *a*, Complex & *b*) [inline]

```
167                                 {
168   Complex tmp = a;
169   a = b;
170   b = tmp;
171 }
```

1.18.9.40 void swap (Real & *a*, Real & *b*) [inline]

```
162                                 {
163   Real tmp = a;
164   a = b;
165   b = tmp;
166 }
```

1.18.9.41 void swap (int & *a*, int & *b*) [inline]

```
157                                 {
158   int tmp = a;
159   a = b;
160   b = tmp;
161 }
```

1.18.9.42 void write (std::ostream & *os*, fieldstate *s*)

References Spectral.

```
191                                   {
192   char c = (s == Spectral) ? 'S' : 'P';
193   os.write(&c, 1);
194 }
```

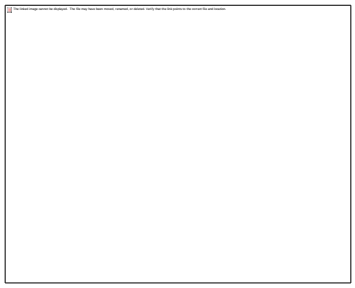
1.18.9.43 void write (std::ostream & *os*, Complex *z*)

References Im(), Re(), and write().

```
202                                 {
203   Real x = Re(z);
204   write(os, x);
205   x = Im(z);
```

```
206   write(os, x);
207 }
```

Here is the call graph for this function:



1.18.9.44 void write (std::ostream & *os*, Real *x*)

```
245                         {
246
247 #if SIZEOF_DOUBLE != 8
248   cerr << "write(ostream& os, Real x) error: channelflow binary IO assumes 64-bit
double-precision floating point.\n";
249   cerr << "The double on this platform is " << 8*SIZEOF_DOUBLE << " bits" << endl;
250   exit(1);
251 #endif
252
253 #if SIZEOF_INT != 4
254   cerr << "write(ostream& os, Real x) error: channelflow binary IO assumes 64-bit
double-precision floating point and 32-bit ints.\n";
255   cerr << "The int on this platform is " << 8*SIZEOF_INT << " bits" << endl;
256   exit(1);
257 #endif
258
259 #if __BYTE_ORDER == __LITTLE_ENDIAN
260   int i[2];
261
262   // Copy 64 bit x into two consecutive 32 bit ints
263   memcpy(i, &x, sizeof(uint64_t));
264
265   // Byteswap each 32 bit int and reverse their order.
266   int tmp = i[1];
267   i[1] = htonl(i[0]);
268   i[0] = htonl(tmp);
269
270   // Write out the two reversed and byteswapped 32bit ints
271   os.write((char*) i, sizeof(double));
```

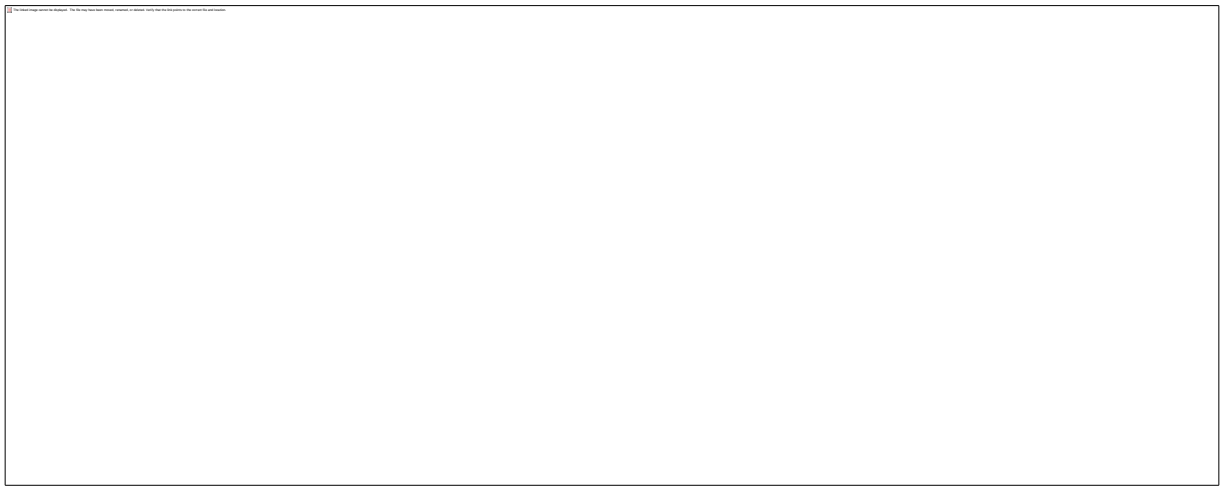
```
272 #else
273   os.write((char*) &x, sizeof(double));
274 #endif
275 }
```

1.18.9.45 void write (std::ostream & *os*, bool *b*)

Referenced by ChebyCoeff::binaryDump(), BasisFunc::binaryDump(), array< T >::binaryDump(), FlowField::binarySave(), and write().

```
180                                {
181   char c = (b) ? '1' : '0';
182   os.write(&c, 1);  // sizeof(char) == 1
183 }
```

Here is the caller graph for this function:



1.18.9.46 void write (std::ostream & *os*, int *n*)

```
220                                {
221
222 #if SIZEOF_INT != 4
223   cerr << "write(ostream& os, int n) :  channelflow binary IO assumes 32-bit int\n";
224   cerr << "The int on this platform is " << 8*SIZEOF_INT << " bits" << endl;
225   exit(1);
226 #endif
227
228   n = htonl(n);
229   os.write((char*) &n, sizeof(int));
230 }
```

1.18.10Variable Documentation

1.18.10.1 const Real pi = M_PI

Referenced by FlowField::addPerturbations(), PPEDNS::advance(), CNABstyleDNS::advance(), RungeKuttaDNS::advance(), MultistepDNS::advance(), chebyDist2(), chebyInnerProduct(), chebyNorm2(), chebypoints(), chebyProfile(), chebyshev(), chebyUprofile(), chebyVprofile(), curl(), diff(), div(), divcheck(), divDist2(), divergence(), divNorm2(), DNSAlgorithm::DNSAlgorithm(), FlowField::Dx(), FlowField::Dz(), grad(), gradient(), ChebyCoeff::interpolate(), lapl(), laplacian(), legendre(), linearizedNL(), makemeen(), makeprtb(), makeroll(), makewave(), makewobl(), MultistepDNS::MultistepDNS(), PoissonSolver::PoissonSolver(), PPEDNS::PPEDNS(), PPEDNS::push(), MultistepDNS::push(), BasisFunc::randomize(), randomProfile(), ChebyCoeff::reflect(), PPEDNS::reset_dt(), CNABstyleDNS::reset_dt(), RungeKuttaDNS::reset_dt(), MultistepDNS::reset_dt(), rotationalNL(), skewsymmetricNL_task3::Run(), FlowField::saveDivSpectrum(), skewsymmetricNL(), skewsymmetricNL_GPU(), TauSolver::solve(), FlowField::translate(), xdiff(), FlowField::y(), FlowField::ygridpts(), and zdiff().

1.18.10.2 const int REAL_DIGITS = 17

Referenced by FlowField::asciiSave(), TurbStats::msave(), BandedTridiag::operator==(), Vector::save(), ComplexChebyCoeff::save(), ChebyCoeff::save(), BasisFunc::save(), BandedTridiag::save(), and FlowField::saveProfile().

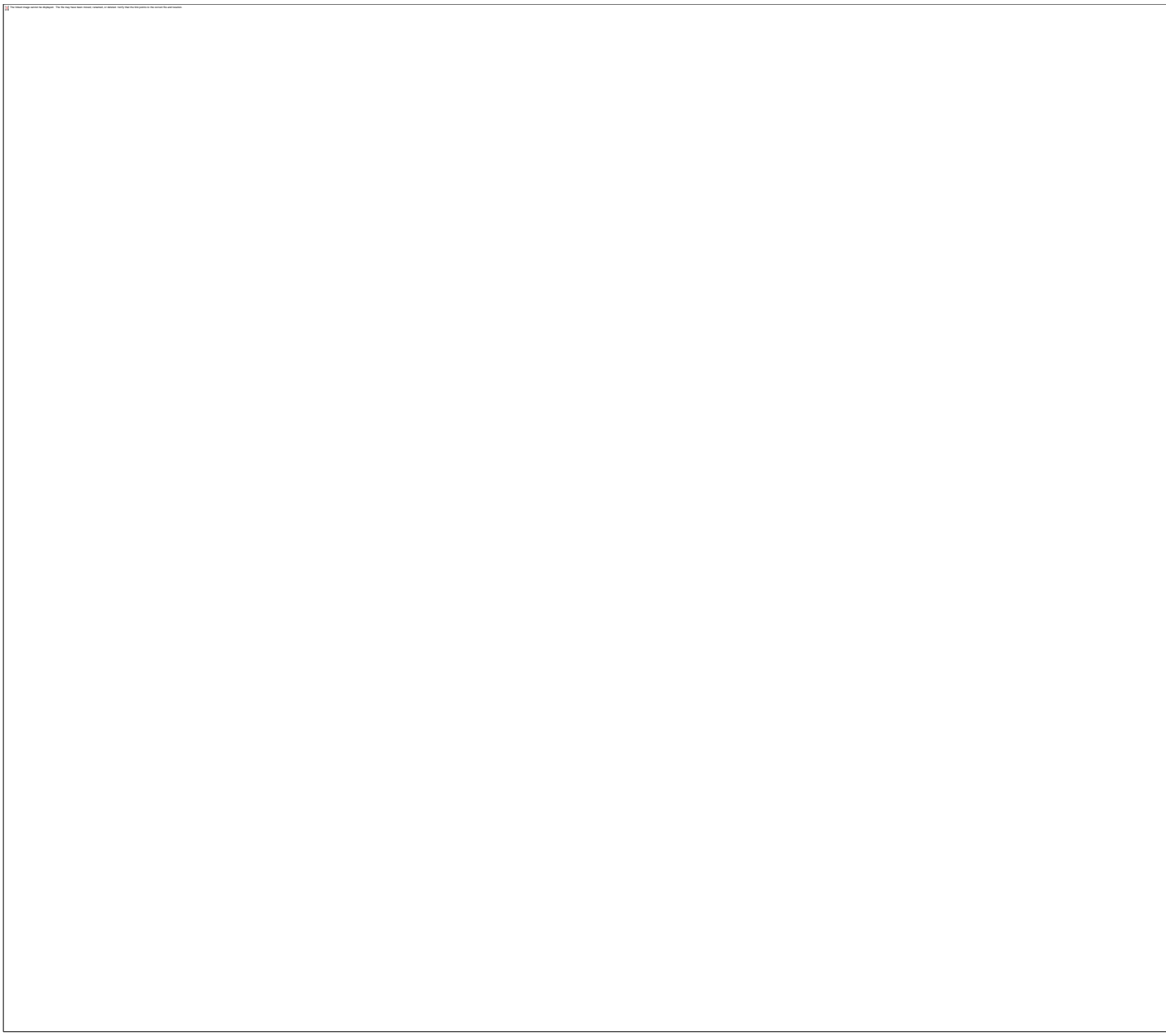
1.18.10.3 const int REAL_IOWIDTH = 24

Referenced by FlowField::asciiSave(), Vector::save(), ComplexChebyCoeff::save(), and ChebyCoeff::save().

1.18.10.4

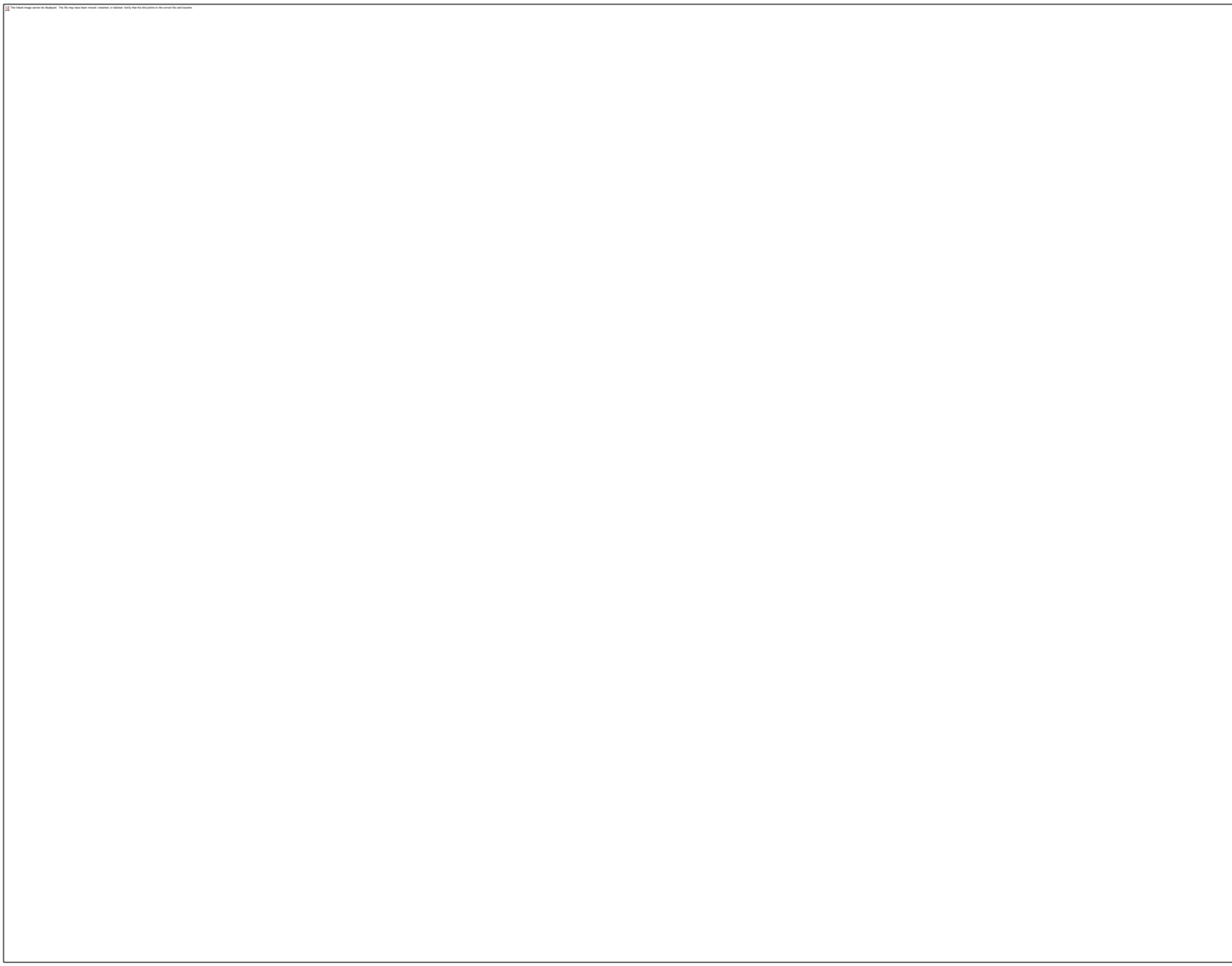
1.19 /_cuda/channelflow-0.9.22/channelflow/poissonsolver.cpp File Reference

#include "channelflow/poissonsolver.h"
Include dependency graph for poissonsolver.cpp:

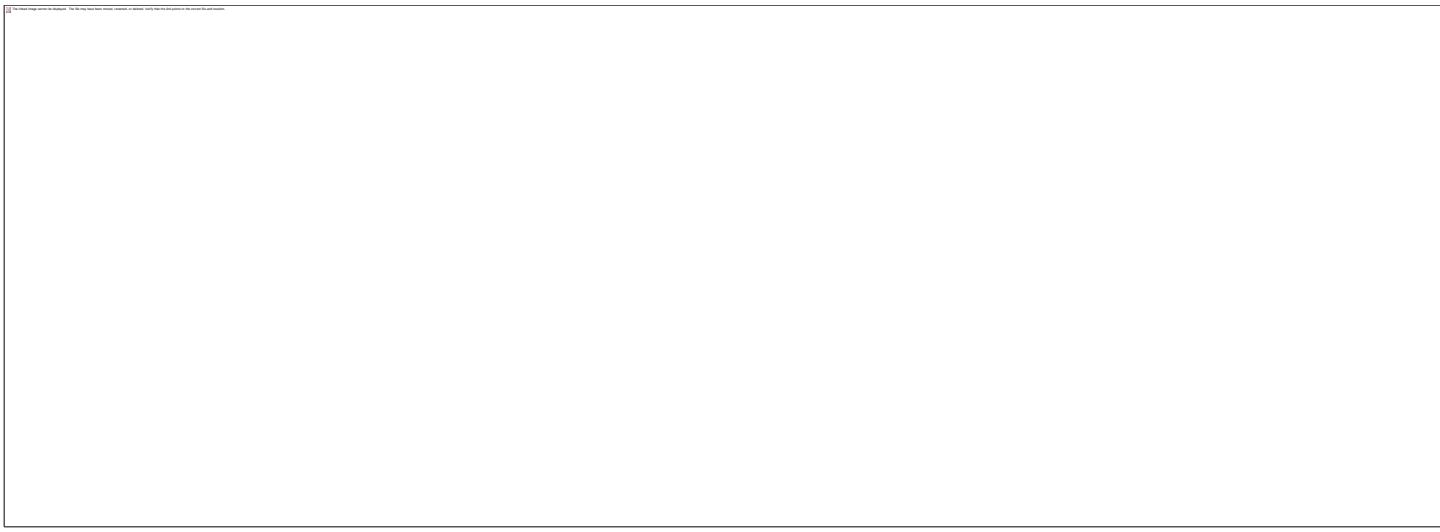


1.20 /_cuda/channelflow-0.9.22/channelflow/poissonsolver.h File Reference

#include "channelflow/mathdefs.h"
#include "channelflow/vector.h"
#include "channelflow/chebyshev.h"
#include "channelflow/flowfield.h"
#include "channelflow/helmholtz.h"
#include "channelflow/diffops.h"
Include dependency graph for poissonsolver.h:



This graph shows which files directly or indirectly include this file:

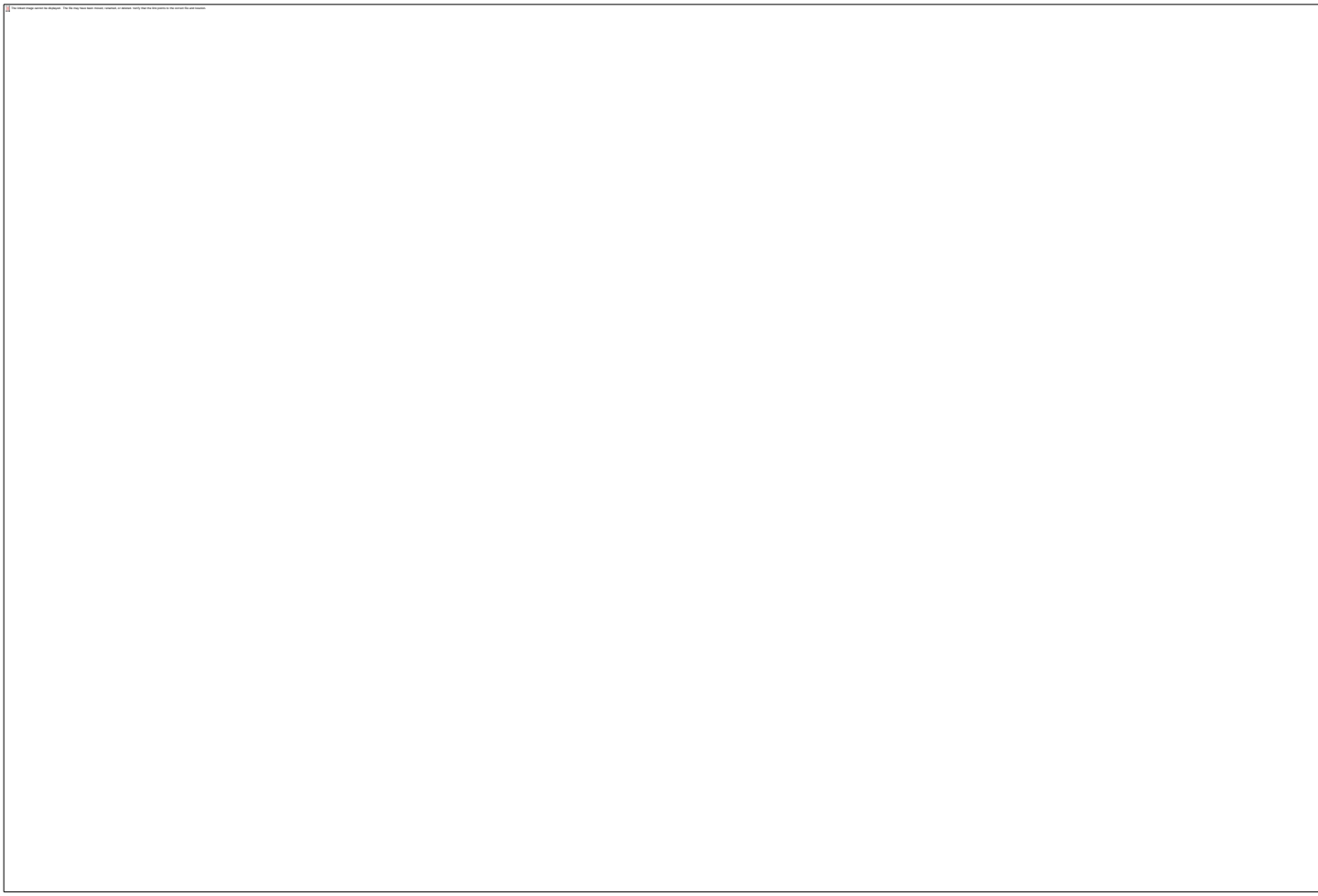


1.20.1 Classes

- class PoissonSolver
- class PressureSolver

1.21 /_cuda/channelflow-0.9.22/channelflow/tausolver.cpp File Reference

#include <iomanip>
#include "channelflow/tausolver.h"
Include dependency graph for tausolver.cpp:



1.21.1 Functions

- int kprime_func (int k, int Nb)
- int n_func (int k, int Nb)
- Real divcheck (string &label, int kx, int kz, Real kxLx, Real kzLz, const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, const ComplexChebyCoeff &w, bool verbose)

1.21.2 Variables

- const Real MINIMUM_DISCRIMINANT = 1e-4
- const Real EPSILON = 1e-7

1.21.3 Function Documentation

1.21.3.1 Real divcheck (string & *label*, int *kx*, int *kz*, Real *kxLx*, Real *kzLz*, const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, const ComplexChebyCoeff & *w*, bool *verbose*)

References abs2(), diff(), I(), ComplexChebyCoeff::length(), and pi.

```
50                                          {
51
52   // Verify divergence
53   int N = u.length();
54   if (v.length() != N || w.length() != N) {
55     cout << "divcheck length problem!" << endl;
56     exit(1);
57   }
58
59   ComplexChebyCoeff tmp = diff(v);
60   Real diverge=0.0;
61   for (int n=N-1; n>=0; --n)
62     diverge += abs2(tmp[n] + (pi*2*(kxLx*u[n] + kzLz*w[n]))*I);
63   diverge = sqrt(diverge);
64
65   if (diverge > 1e-13 || verbose) {
66     cout << label << endl;
67     cout << "kx, kz == " << kx << ", " << kz << endl;
68     cout << "kxLx, kzLz == " << kxLx << ", " << kzLz << endl;
69     cout << "divergence == " << diverge << endl;
70   }
71   return diverge;
72 }
```

Here is the call graph for this function:

1.21.3.2 int kprime_func (int *k*, int *Nb*) [inline]

```
33                                  {
34   return (k%2 == 0) ? Nb-1 : Nb;
35 }
```

1.21.3.3 int n_func (int *k*, int *Nb*)

Referenced by TauSolver::TauSolver().

```
37                          {
38   if (k==0)
39     return Nb-1;
40   else if (k==Nb)
41     return 0;
42   else if (k%2 == 0)
43     return 2*(Nb-1);
44   else
45     return 2*Nb;
46 }
```

Here is the caller graph for this function:

1.21.4 Variable Documentation

1.21.4.1 const Real EPSILON = 1e-7

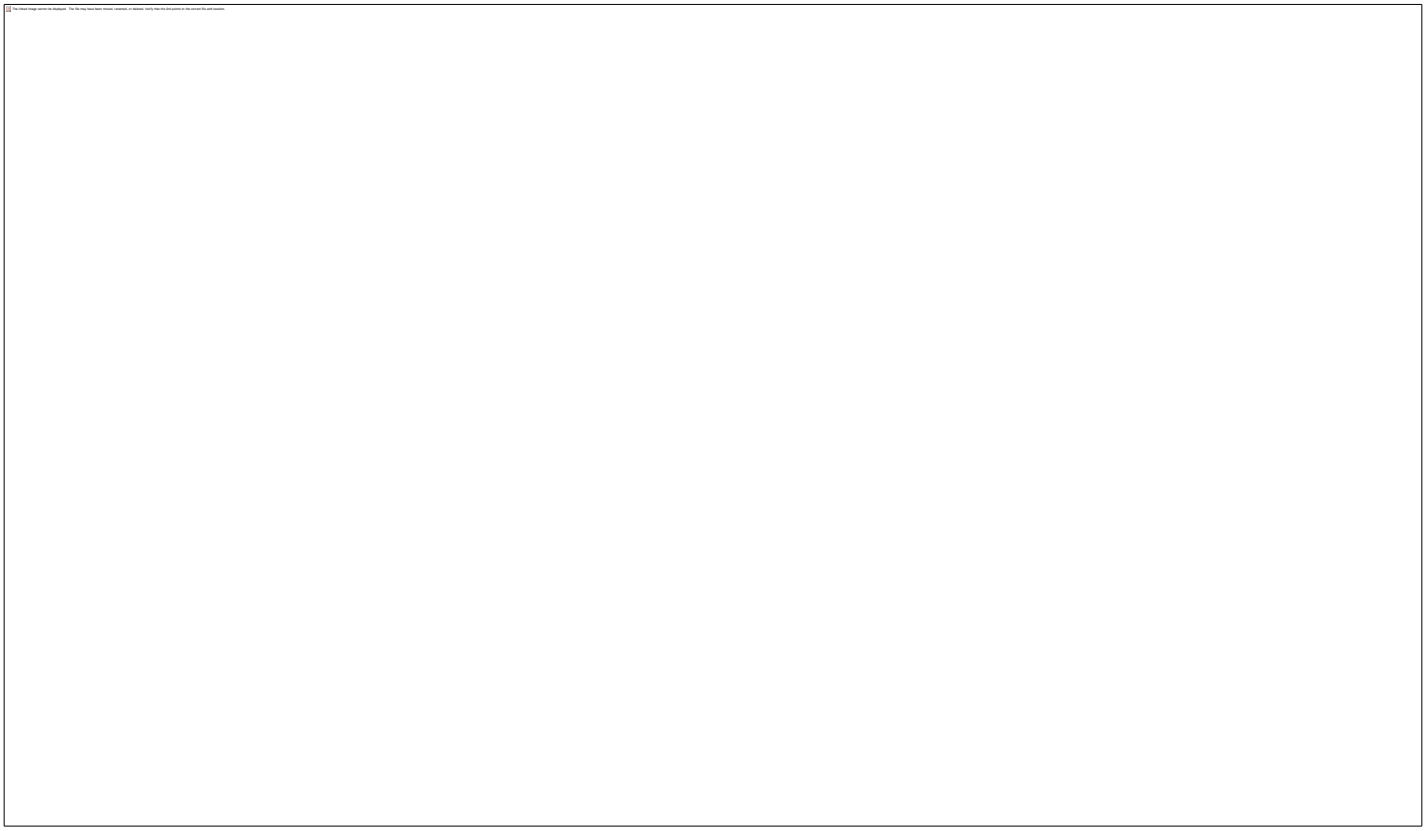
1.21.4.2 const Real MINIMUM_DISCRIMINANT = 1e-4

Referenced by TauSolver::TauSolver().

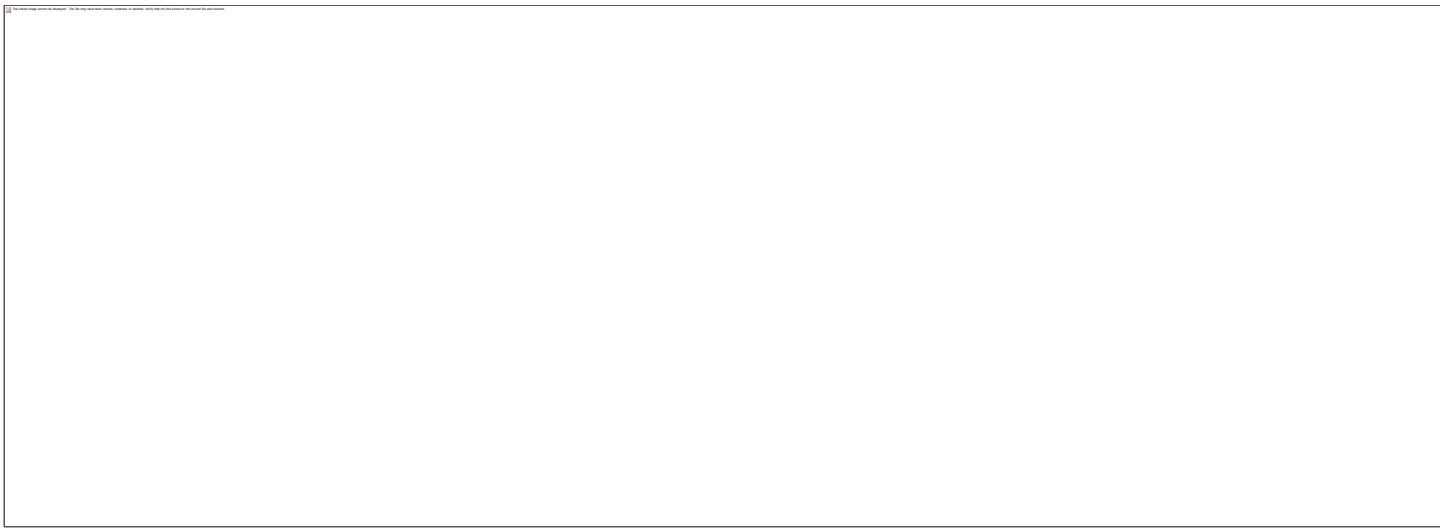
1.21.4.3

1.22 /_cuda/channelflow-0.9.22/channelflow/tausolver.h File Reference

#include "channelflow/mathdefs.h"
#include "channelflow/helmholtz.h"
#include "channelflow/vector.h"
#include "channelflow/chebyshev.h"
Include dependency graph for tausolver.h:



This graph shows which files directly or indirectly include this file:



1.22.1 Classes

- class TauSolver

1.22.2 Functions

- Real divcheck (std::string &label, int kx, int kz, Real Lx, Real Lz, const ComplexChebyCoeff &u, const ComplexChebyCoeff &v, const ComplexChebyCoeff &w, bool verbose=false)

1.22.3 Function Documentation

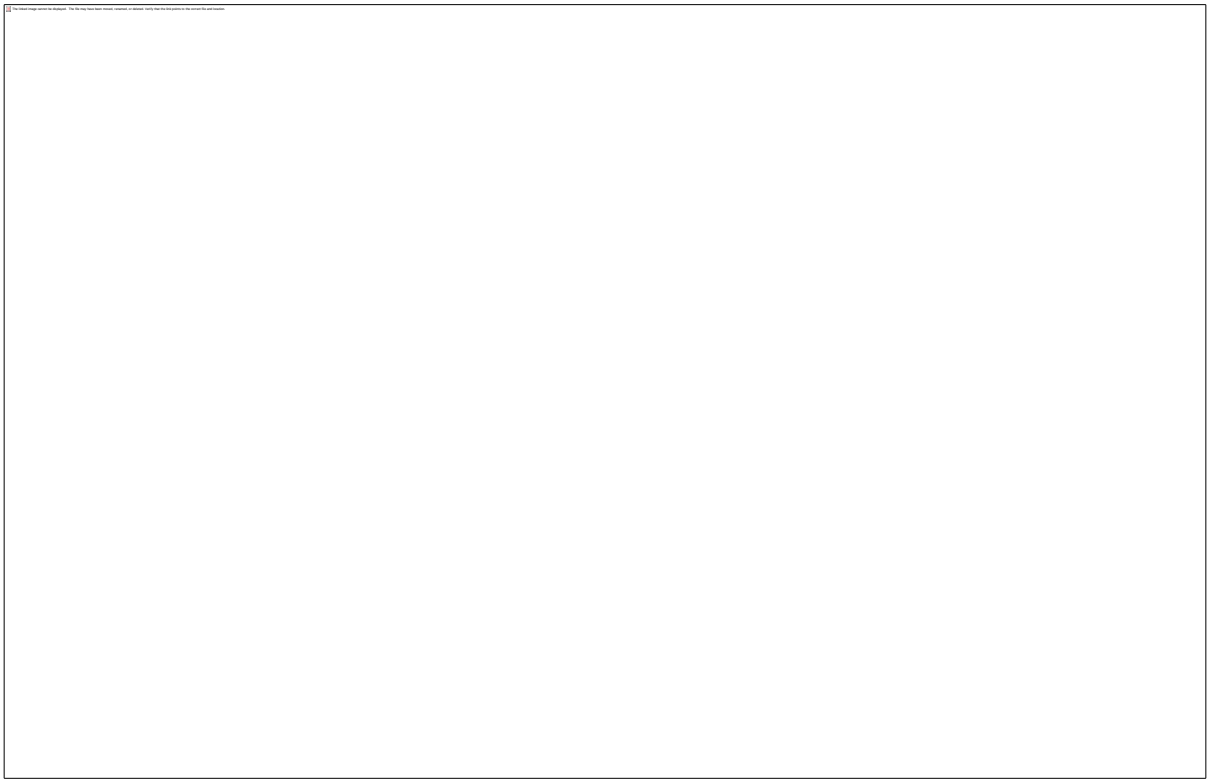
1.22.3.1 Real divcheck (std::string & *label*, int *kx*, int *kz*, Real *Lx*, Real *Lz*, const ComplexChebyCoeff & *u*, const ComplexChebyCoeff & *v*, const ComplexChebyCoeff & *w*, bool *verbose* = false)

References abs2(), diff(), I(), ComplexChebyCoeff::length(), and pi.

```
50                                          {
51
52  // Verify divergence
53  int N = u.length();
54  if (v.length() != N || w.length() != N) {
55    cout << "divcheck length problem!" << endl;
56    exit(1);
57  }
58
59  ComplexChebyCoeff tmp = diff(v);
60  Real diverge=0.0;
61  for (int n=N-1; n>=0; --n)
62    diverge += abs2(tmp[n] + (pi*2*(kxLx*u[n] + kzLz*w[n]))*I);
```

```
63   diverge = sqrt(diverge);
64
65   if (diverge > 1e-13 || verbose) {
66     cout << label << endl;
67     cout << "kx, kz == " << kx << ", " << kz << endl;
68     cout << "kxLx, kzLz == " << kxLx << ", " << kzLz << endl;
69     cout << "divergence == " << diverge << endl;
70   }
71   return diverge;
72 }
```

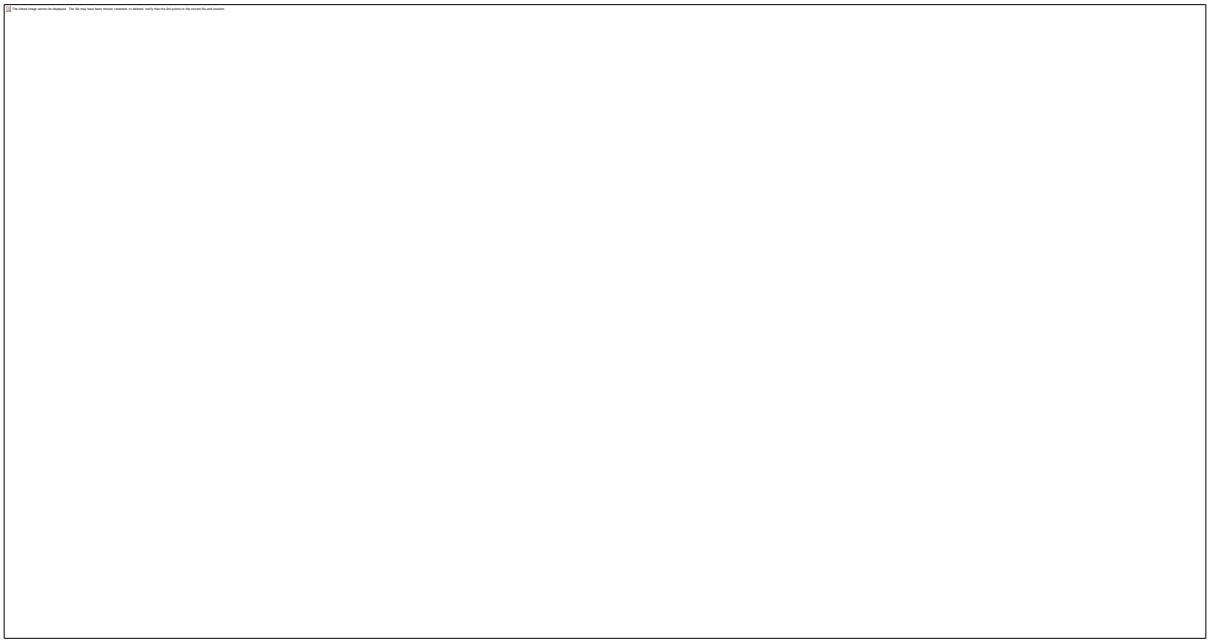
Here is the call graph for this function:



1.22.3.2

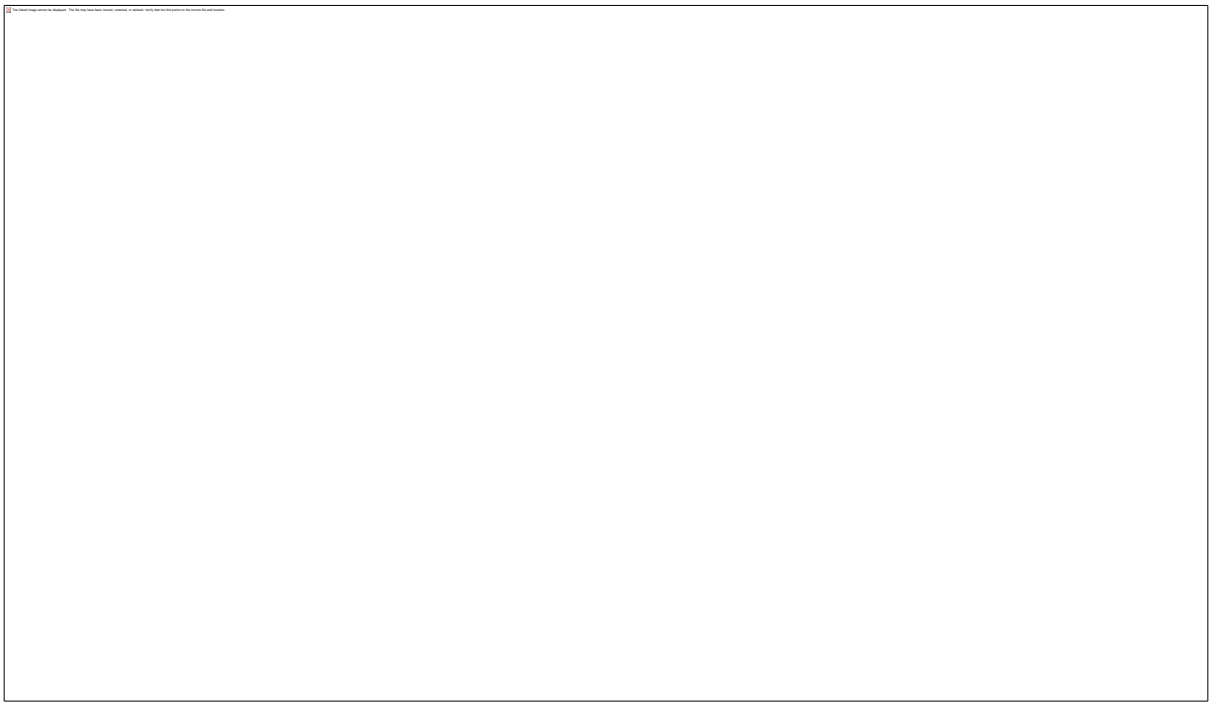
1.23 /_cuda/channelflow-0.9.22/channelflow/turbstats.cpp File Reference

#include <fstream>
#include <iomanip>
#include "channelflow/turbstats.h"
Include dependency graph for turbstats.cpp:

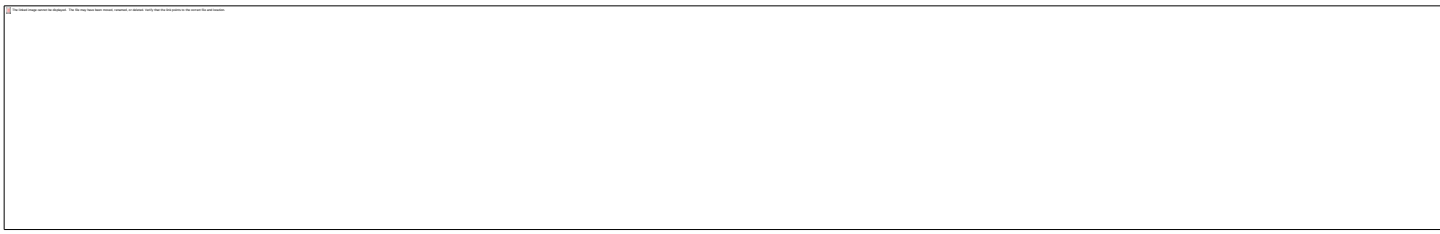


1.24 /_cuda/channelflow-0.9.22/channelflow/turbstats.h File Reference

#include "channelflow/mathdefs.h"
#include "channelflow/chebyshev.h"
#include "channelflow/basisfunc.h"
#include "channelflow/flowfield.h"
Include dependency graph for turbstats.h:



This graph shows which files directly or indirectly include this file:

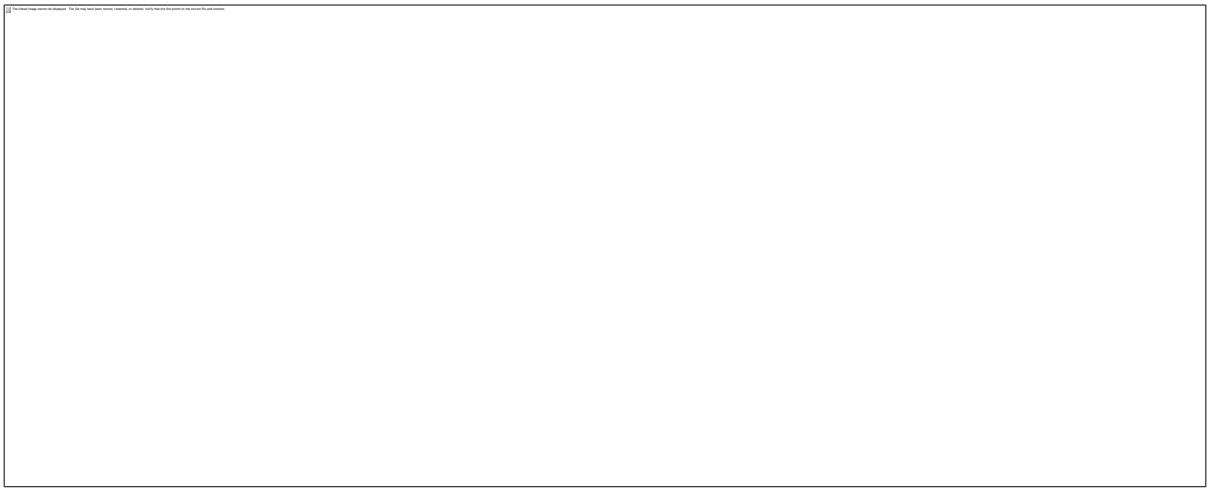


1.24.1 Classes

- class TurbStats

1.25 /_cuda/channelflow-0.9.22/channelflow/utilfuncs.cpp File Reference

#include "channelflow/utilfuncs.h"
Include dependency graph for utilfuncs.cpp:



1.25.1 Functions

- Real polyInterp (const array< Real > &fa, const array< Real > &xa, Real x)
- Real secantSearch (Real a, Real b, array< Real > &fn, const array< Real > &xn, Real feps, int maxsteps)
- Real bisectSearch (Real a, Real b, array< Real > &fn, const array< Real > &xn, Real feps, int maxsteps)
- BasisFunc makeroll (Real Lx, Real Lz, Real a, Real b, int Ny)
- BasisFunc makewave (Real Lx, Real Lz, Real a, Real b, int Ny)
- BasisFunc makemeen (Real Lx, Real Lz, Real a, Real b, int Ny)
- BasisFunc makewobl (Real Lx, Real Lz, Real a, Real b, int Ny)
- BasisFunc makeprtb (Real Lx, Real Lz, Real a, Real b, int Ny)

1.25.2 Function Documentation

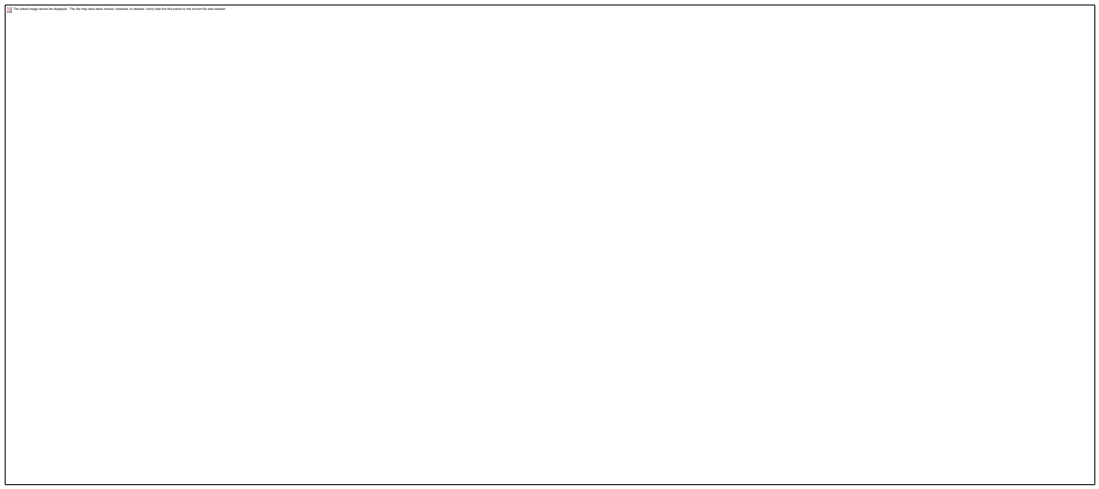
1.25.2.1 Real bisectSearch (Real *a*, Real *b*, array< Real > & *fn*, const array< Real > & *xn*, Real *feps*, int *maxsteps*)

References abs(), cferror(), and polyInterp().

```
105                                     {
106   Real c = 0.5*(a+b)/2; // GCC-3.3.5 complains if not explicitly init'ed
107   Real fc;
108   Real fa = polyInterp(fn, xn, a);
109   Real fb = polyInterp(fn, xn, b);
110   if (fa*fb > 0)
```

```
111     cferror("bisectSearch(a,b,fn,xn) : a and b don't bracket a zero");
112
113   for (int n=0; n<maxsteps; ++n) {
114     c = 0.5*(a+b);
115     fc = polyInterp(fn,xn,c);
116     if (abs(fc) < feps)
117       break;
118     if (fc*fa > 0) {
119       a = c;
120       fa = fc;
121     }
122     else {
123       b = c;
124       fb = fc;
125     }
126   }
127   return c;
128 }
```

Here is the call graph for this function:



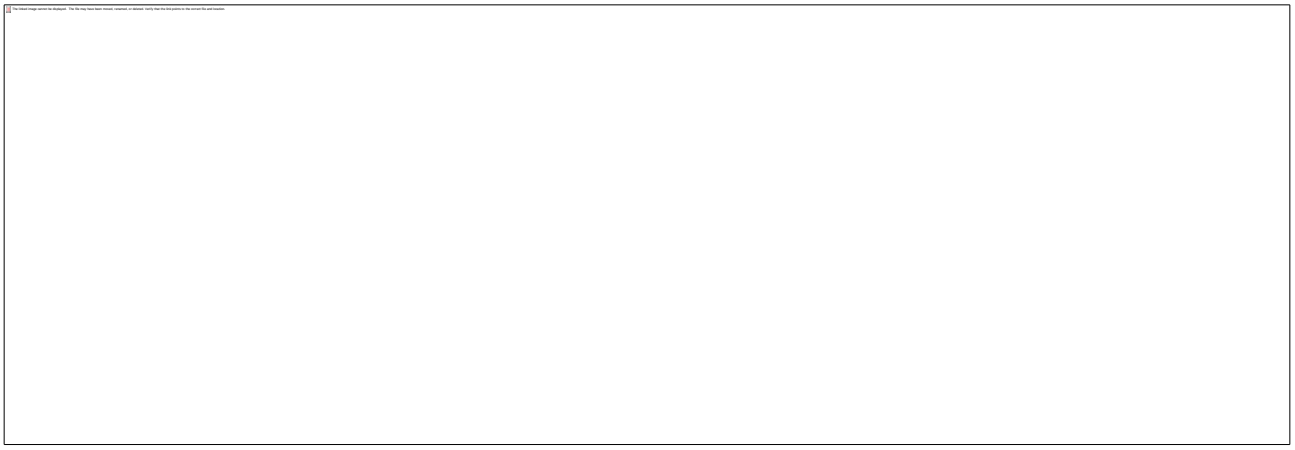
1.25.2.2 BasisFunc makemeen (Real *Lx*, Real *Lz*, Real *a*, Real *b*, int *Ny*)

References chebypoints(), L2Norm(), BasisFunc::makeSpectral(), Physical, pi, ComplexChebyCoeff::re, and BasisFunc::u().

```
272                                         {
273
274   BasisFunc rtn(Ny, 0, 0, Lx, Lz, a, b, Physical);
275   Vector y = chebypoints(Ny,-1,1);
276   for (int n=0; n<Ny; ++n)
277     rtn.u().re[n] =  -sin(pi*y[n]);
```

```
278   rtn.makeSpectral();
279   rtn *= 1.0/L2Norm(rtn);
280   return rtn;
281 }
```

Here is the call graph for this function:



1.25.2.3 BasisFunc makeprtb (Real *Lx*, Real *Lz*, Real *a*, Real *b*, int *Ny*)

References chebypoints(), diff(), ComplexChebyCoeff::im, L2Norm(), BasisFunc::makeSpectral(), Physical, pi, ComplexChebyCoeff::re, BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

```
320                                                 {
321   BasisFunc rtn(Ny, 1, 1, Lx, Lz, a, b, Physical);
322
323   Vector y = chebypoints(Ny,-1,1);
324   for (int n=0; n<Ny; ++n) {
325     Real yp = 1 + 2*(y[n]-b)/(b-a);
326     Real p = 1-yp*yp;
327     Real pp = p*p;
328     rtn.u().re[n] = p*(1+yp);
329     rtn.v().im[n] = pp*(1+yp);
330   }
331   rtn.makeSpectral();
332   ChebyCoeff tmp;
333
334   // Build up Re(w) = -Lz/Lx  Re(u) - Lz/2pikz Im(v')
335   //                = -Lz/Lx [Re(u) + Lx/2pikz Im(v')]
336   diff(rtn.v().im, tmp);
337   rtn.w().re = tmp;
338   rtn.w().re *= Lx/(2*pi);
339   rtn.w().re += rtn.u().re;
```

```
340   rtn.w().re *= -Lz/Lx;
341
342   rtn *= 1.0/L2Norm(rtn);
343   return rtn;
344 }
```

Here is the call graph for this function:

1.25.2.4 BasisFunc makeroll (Real *Lx*, Real *Lz*, Real *a*, Real *b*, int *Ny*)

References cferror(), chebypoints(), diff(), ComplexChebyCoeff::im, L2Norm(), ChebyCoeff::makeSpectral(), Physical, pi, pow(), ComplexChebyCoeff::re, ChebyCoeff::setState(), ChebyCoeff::setToZero(), Spectral, BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

```
218                                         {
219   if (Ny < 5)
220     cferror("Ny too small. Set Ny >= 5");
221
222   BasisFunc rtn(Ny, 0, 1, Lx, Lz, a, b, Spectral);
223
224   rtn.v().re[0] =  3.0/32.0;
225   rtn.v().re[2] = -1.0/8.0;
226   rtn.v().re[4] =  1.0/32.0;
227
228   ChebyCoeff tmp = diff(rtn.v().re);
229   rtn.w().im = tmp;
230   rtn.w().im *= Lz/(2*pi);
231
232   tmp.setToZero();
233   tmp.setState(Physical);
234   Vector y = chebypoints(Ny,-1,1);
235   for (int n=0; n<Ny; ++n) {
236     Real yp = 1 + 2*(y[n]-b)/(b-a);
237     tmp[n] = 2*(pow(yp,6) - 0.7*pow(yp,2) - 0.3);
238   }
239   tmp.makeSpectral();
240   rtn.u().re = tmp;
241
242   rtn *= -1.0/L2Norm(rtn);
243   return rtn;
244 }
```

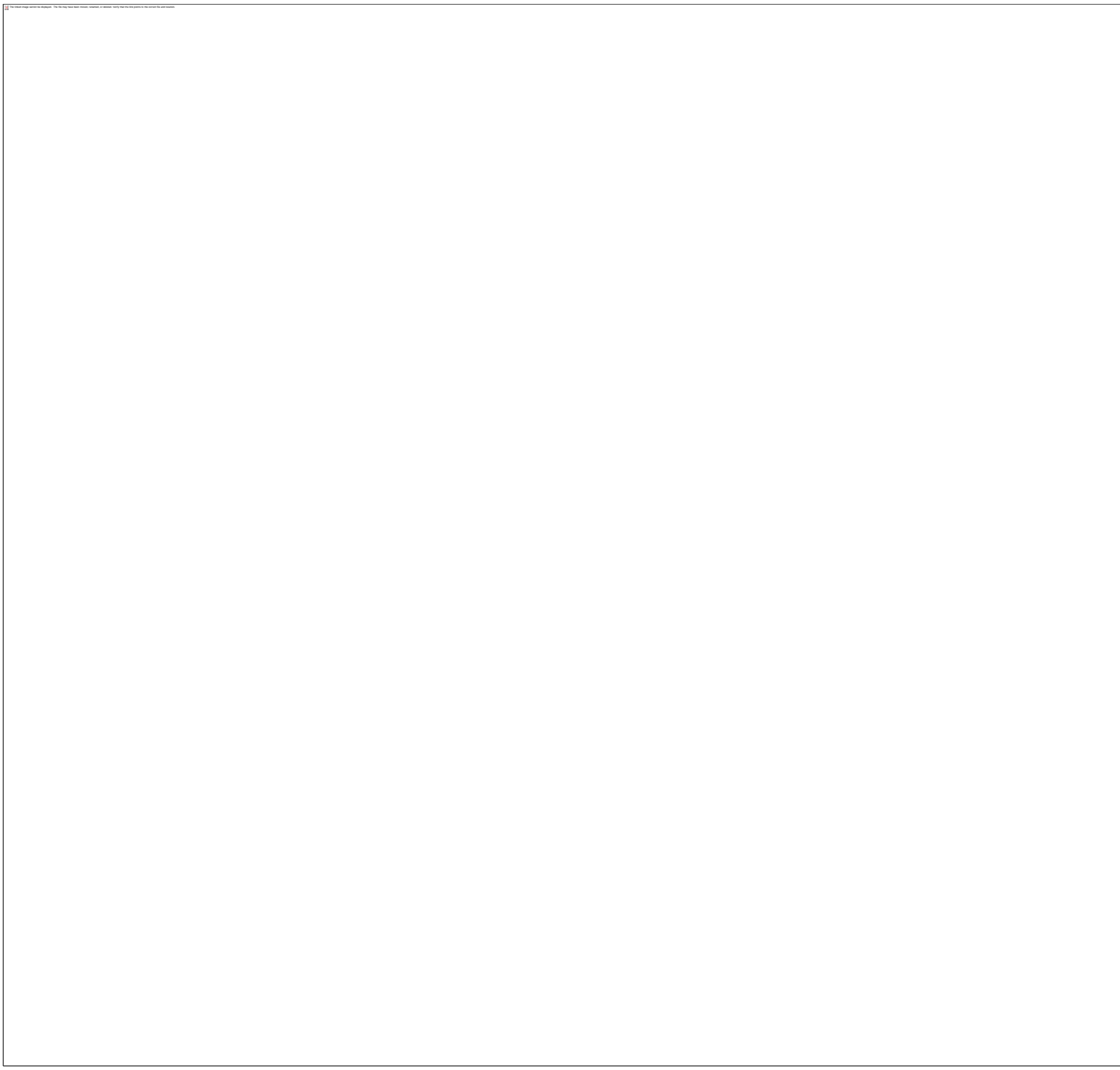
Here is the call graph for this function:

1.25.2.5 BasisFunc makewave (Real *Lx*, Real *Lz*, Real *a*, Real *b*, int *Ny*)

References chebypoints(), diff(), ComplexChebyCoeff::im, L2Norm(), BasisFunc::makeSpectral(), Physical, pi, ComplexChebyCoeff::re, BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

```
246                                                 {
247   BasisFunc rtn(Ny, 1, 0, Lx, Lz, a, b, Physical);
248
249   Vector y = chebypoints(Ny,-1,1);
250   for (int n=0; n<Ny; ++n) {
251     Real yp = 1 + 2*(y[n]-b)/(b-a);
252     Real p = 1-yp*yp;
253     Real pp = p*p;
254     rtn.w().re[n] = 0.7*p*sin(pi*yp);
255     rtn.w().im[n] = p*cos(pi*yp);
256     rtn.v().re[n] = 0.04*pp*sin(pi*yp);
257     rtn.v().im[n] = 0.04*pp*cos(pi*yp);
258   }
259   rtn.makeSpectral();
260   ChebyCoeff tmp = diff(rtn.v().re);
261   rtn.u().im = tmp;
262   rtn.u().im *= Lx/(2*pi);
263
264   tmp = diff(rtn.v().im);
265   rtn.u().re = tmp;
266   rtn.u().re *= -Lx/(2*pi);
267
268   rtn *= 1.0/L2Norm(rtn);
269   return rtn;
270 }
```

Here is the call graph for this function:



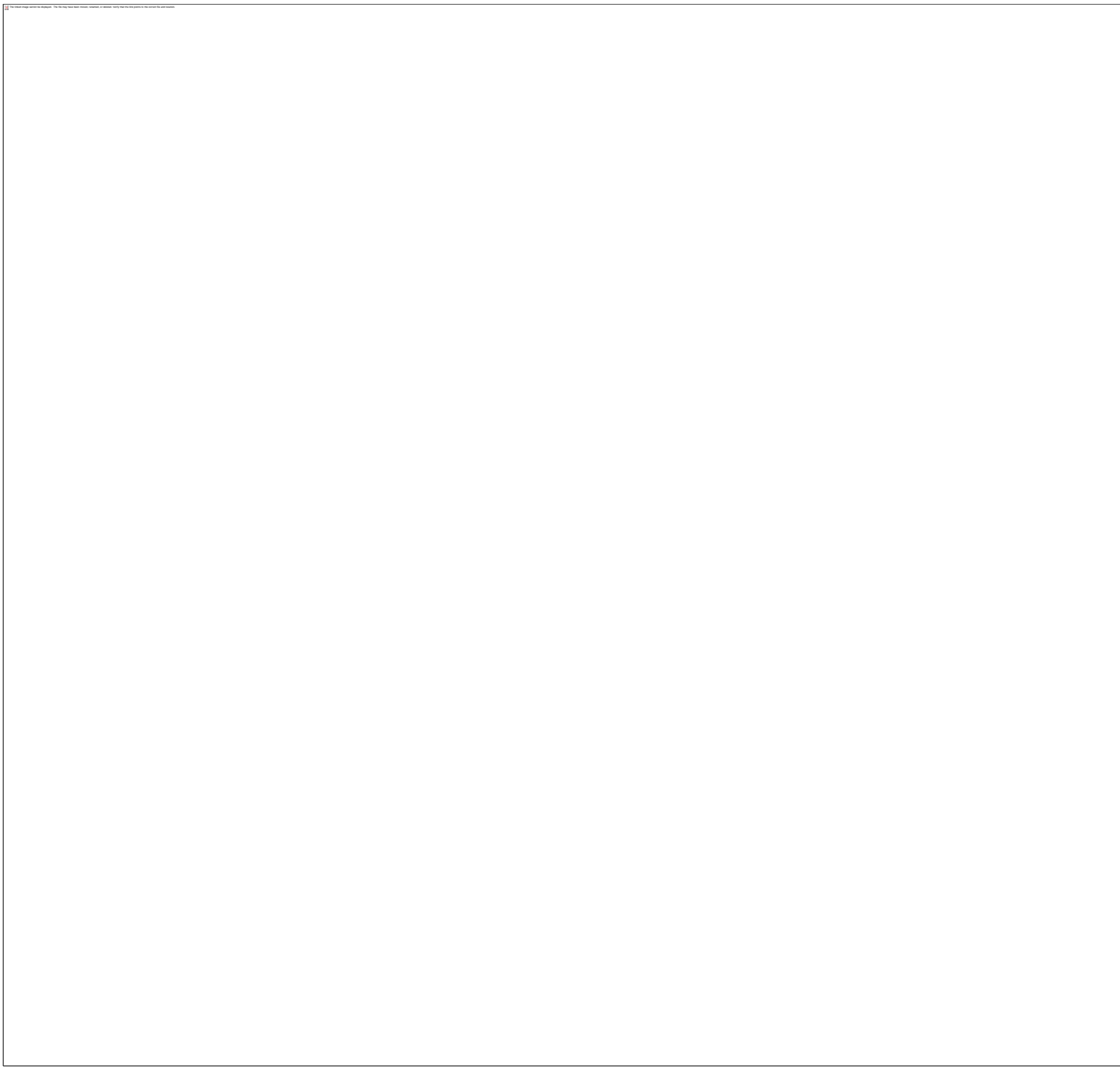
1.25.2.6 BasisFunc makewobl (Real *Lx*, Real *Lz*, Real *a*, Real *b*, int *Ny*)

References chebypoints(), diff(), ComplexChebyCoeff::im, L2Norm(), BasisFunc::makeSpectral(), Physical, pi, ComplexChebyCoeff::re, BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

```
283                                         {
```

```
284   BasisFunc rtn(Ny, 1, 1, Lx, Lz, a, b, Physical);
285
286   Vector y = chebypoints(Ny,-1,1);
287   for (int n=0; n<Ny; ++n) {
288     Real yp = 1 + 2*(y[n]-b)/(b-a);
289     Real p = 1-yp*yp;
290     Real pp = p*p;
291     rtn.u().re[n] = p*cos(pi*yp);
292     rtn.u().im[n] = -p*sin(pi*yp);
293     rtn.v().re[n] = 0.2*pp*cos(pi*yp);
294     rtn.v().im[n] = -0.2*pp*sin(pi*yp);
295   }
296   rtn.makeSpectral();
297   ChebyCoeff tmp;
298
299   // Build up Re(w) = -Lz/Lx  Re(u) - Lz/2pikz Im(v')
300   //                = -Lz/Lx [Re(u) + Lx/2pikz Im(v')]
301   diff(rtn.v().im, tmp);
302   rtn.w().re = tmp;
303   rtn.w().re *= Lx/(2*pi);
304   rtn.w().re += rtn.u().re;
305   rtn.w().re *= -Lz/Lx;
306
307   // Build up Im(w) = -Lz/Lx  Im(u) + Lz/2pikz Re(v')
308   //                = -Lz/Lx [Im(u) - Lx/2pikz Re(v')]
309   diff(rtn.v().re, tmp);
310   rtn.w().im = tmp;
311   rtn.w().im *= -Lx/(2*pi);
312   rtn.w().im += rtn.u().im;
313   rtn.w().im *= -Lz/Lx;
314
315   rtn *= 1.0/L2Norm(rtn);
316   return rtn;
317 }
```

Here is the call graph for this function:



1.25.2.7 Real polyInterp (const array< Real > & *fa*, const array< Real > & *xa*, Real *x*)

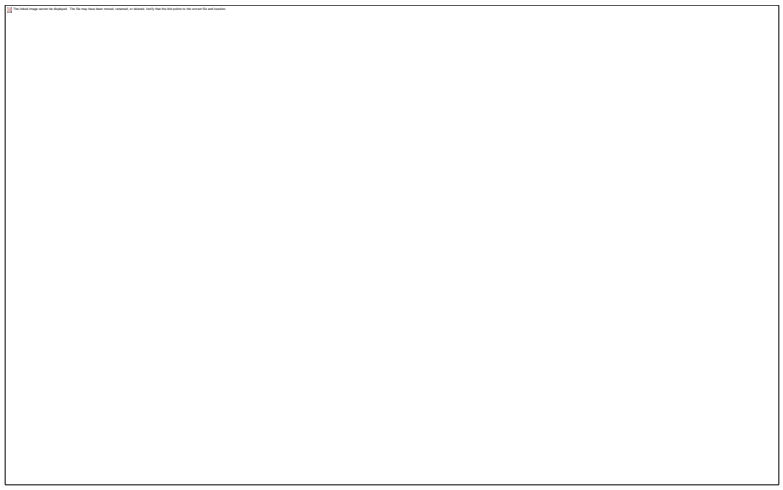
References abs(), cferror(), array< T >::N(), and xdiff().

Referenced by bisectSearch(), FieldSeries::interpolate(), and secantSearch().

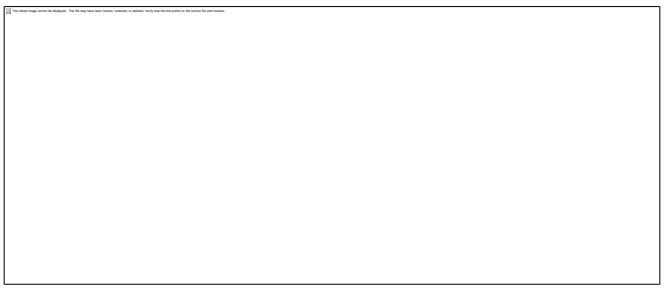
```
36                                                  {
37   assert(fa.N() == xa.N());
```

```
38   int N = fa.N();
39   array<Real> C(N);
40   array<Real> D(N);
41
42   Real xdiff = abs(x-xa[0]);
43   Real xdiff2 = 0.0;
44   Real df;
45
46   int i_closest = 0;
47   for (int i=0; i<N; ++i) {
48     if ((xdiff2 = abs(x-xa[i])) < xdiff) {
49       i_closest = i;
50       xdiff = xdiff2;
51     }
52     C[i] = fa[i];
53     D[i] = fa[i];
54   }
55
56   // Initial approx to f(x)
57   Real fx = fa[i_closest--];
58
59   for (int m=1; m<N; ++m) {
60     for (int i=0; i<N-m; ++i) {
61       Real C_dx = xa[i]-x;
62       Real D_dx = xa[i+m] - x;
63       Real C_D = C[i+1]-D[i];
64       Real denom = C_dx - D_dx;
65       if (denom == 0.0)
66         cferror("polyinterp(fa,xa,x) : two values of xa has are equal!");
67       denom = C_D/denom;
68       C[i] = C_dx*denom;
69       D[i] = D_dx*denom;
70     }
71     fx += (df=(2*(i_closest+1) < (N-m)? C[i_closest+1] : D[i_closest--]));
72   }
73   return fx;
74 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



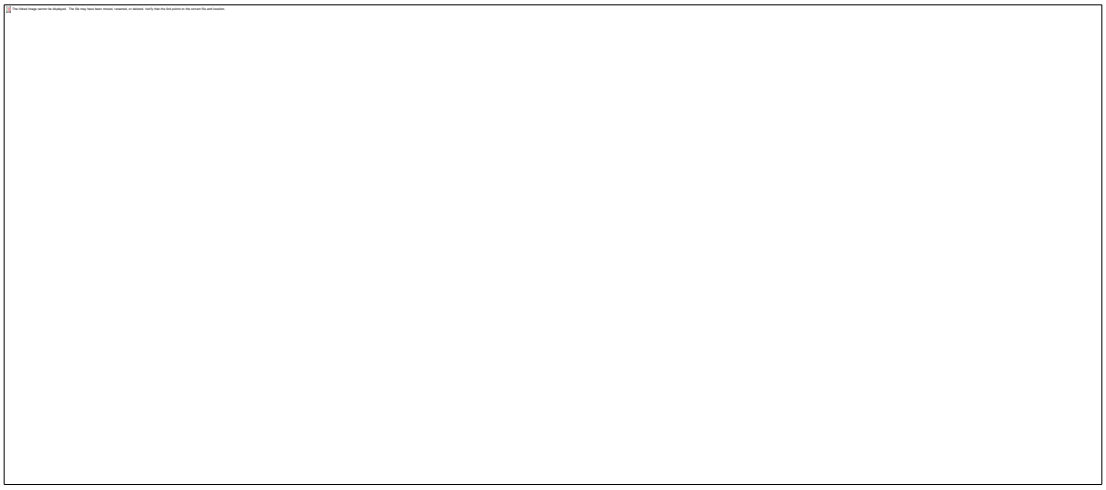
1.25.2.8 Real secantSearch (Real *a*, Real *b*, array< Real > & *fn*, const array< Real > & *xn*, Real *feps*, int *maxsteps*)

References abs(), cferror(), and polyInterp().

```
77                                      {
78   Real fa = polyInterp(fn, xn, a);
79   Real fb = polyInterp(fn, xn, b);
80   if (fa*fb > 0)
81     cferror("secantSearch(a,b,fn,xn) : a and b don't bracket a zero");
82
83   Real c = a - fa * (b-a)/(fb-fa);
84   Real fc;
85
86   for (int n=0; n<maxsteps; ++n) {
87     c = a - fa * (b-a)/(fb-fa);
88     fc = polyInterp(fn,xn,c);
89     if (abs(fc) < feps)
90       break;
91     if (fc*fa > 0) {
92       a = c;
```

```
93       fa = fc;
94     }
95     else {
96       b = c;
97       fb = fc;
98     }
99   }
100   return c;
101 }
```

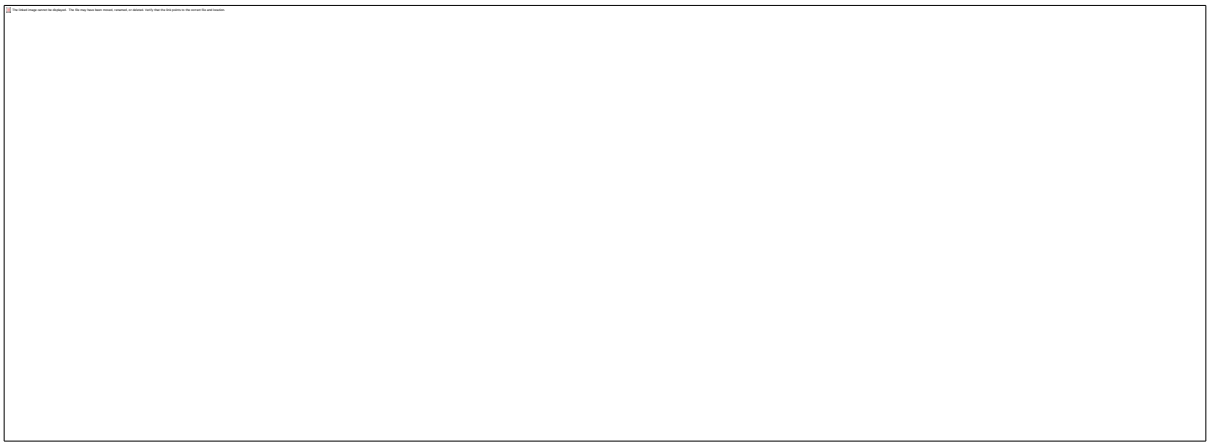
Here is the call graph for this function:



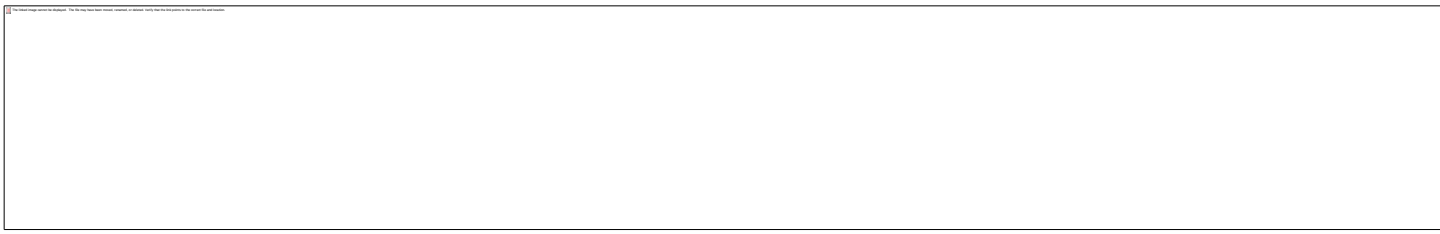
1.25.2.9

1.26 /_cuda/channelflow-0.9.22/channelflow/utilfuncs.h File Reference

#include "channelflow/mathdefs.h"
#include "channelflow/array.h"
#include "channelflow/flowfield.h"
#include "channelflow/dns.h"
Include dependency graph for utilfuncs.h:



This graph shows which files directly or indirectly include this file:



1.26.1 Classes

- class FieldSeries

1.26.2 Functions

- BasisFunc makeroll (Real Lx, Real Lz, Real a, Real b, int Ny)
- BasisFunc makewave (Real Lx, Real Lz, Real a, Real b, int Ny)
- BasisFunc makemeen (Real Lx, Real Lz, Real a, Real b, int Ny)
- BasisFunc makewobl (Real Lx, Real Lz, Real a, Real b, int Ny)
- BasisFunc makeprtb (Real Lx, Real Lz, Real a, Real b, int Ny)
- Real polyInterp (const array< Real > &fn, const array< Real > &xn, Real x)
- Real secantSearch (Real a, Real b, array< Real > &fn, const array< Real > &xn, Real feps=1e-14, int maxsteps=50)
- Real bisectSearch (Real a, Real b, array< Real > &fn, const array< Real > &xn, Real feps=1e-14, int maxsteps=50)

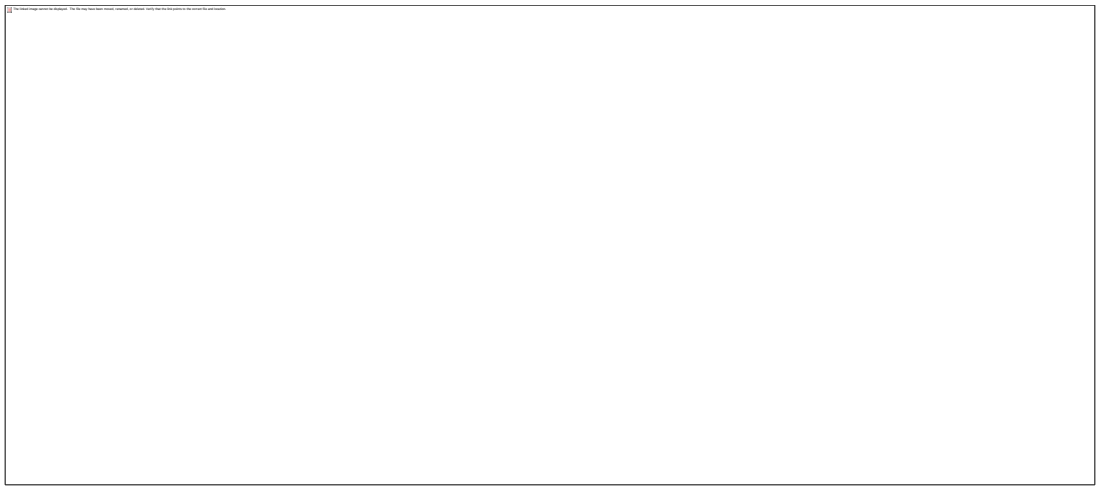
1.26.3 Function Documentation

1.26.3.1 Real bisectSearch (Real *a*, Real *b*, array< Real > & *fn*, const array< Real > & *xn*, Real *feps* = 1e-14, int *maxsteps* = 50)

References abs(), cferror(), and polyInterp().

```
105                                     {
106   Real c = 0.5*(a+b)/2; // GCC-3.3.5 complains if not explicitly init'ed
107   Real fc;
108   Real fa = polyInterp(fn, xn, a);
109   Real fb = polyInterp(fn, xn, b);
110   if (fa*fb > 0)
111     cferror("bisectSearch(a,b,fn,xn) : a and b don't bracket a zero");
112
113   for (int n=0; n<maxsteps; ++n) {
114     c = 0.5*(a+b);
115     fc = polyInterp(fn,xn,c);
116     if (abs(fc) < feps)
117       break;
118     if (fc*fa > 0) {
119       a = c;
120       fa = fc;
121     }
122     else {
123       b = c;
124       fb = fc;
125     }
126   }
127   return c;
128 }
```

Here is the call graph for this function:



1.26.3.2 BasisFunc makemeen (Real *Lx*, Real *Lz*, Real *a*, Real *b*, int *Ny*)

References chebypoints(), L2Norm(), BasisFunc::makeSpectral(), Physical, pi, ComplexChebyCoeff::re, and BasisFunc::u().

```
272                                                 {
273
274   BasisFunc rtn(Ny, 0, 0, Lx, Lz, a, b, Physical);
275   Vector y = chebypoints(Ny,-1,1);
276   for (int n=0; n<Ny; ++n)
277     rtn.u().re[n] =  -sin(pi*y[n]);
278   rtn.makeSpectral();
279   rtn *= 1.0/L2Norm(rtn);
280   return rtn;
281 }
```

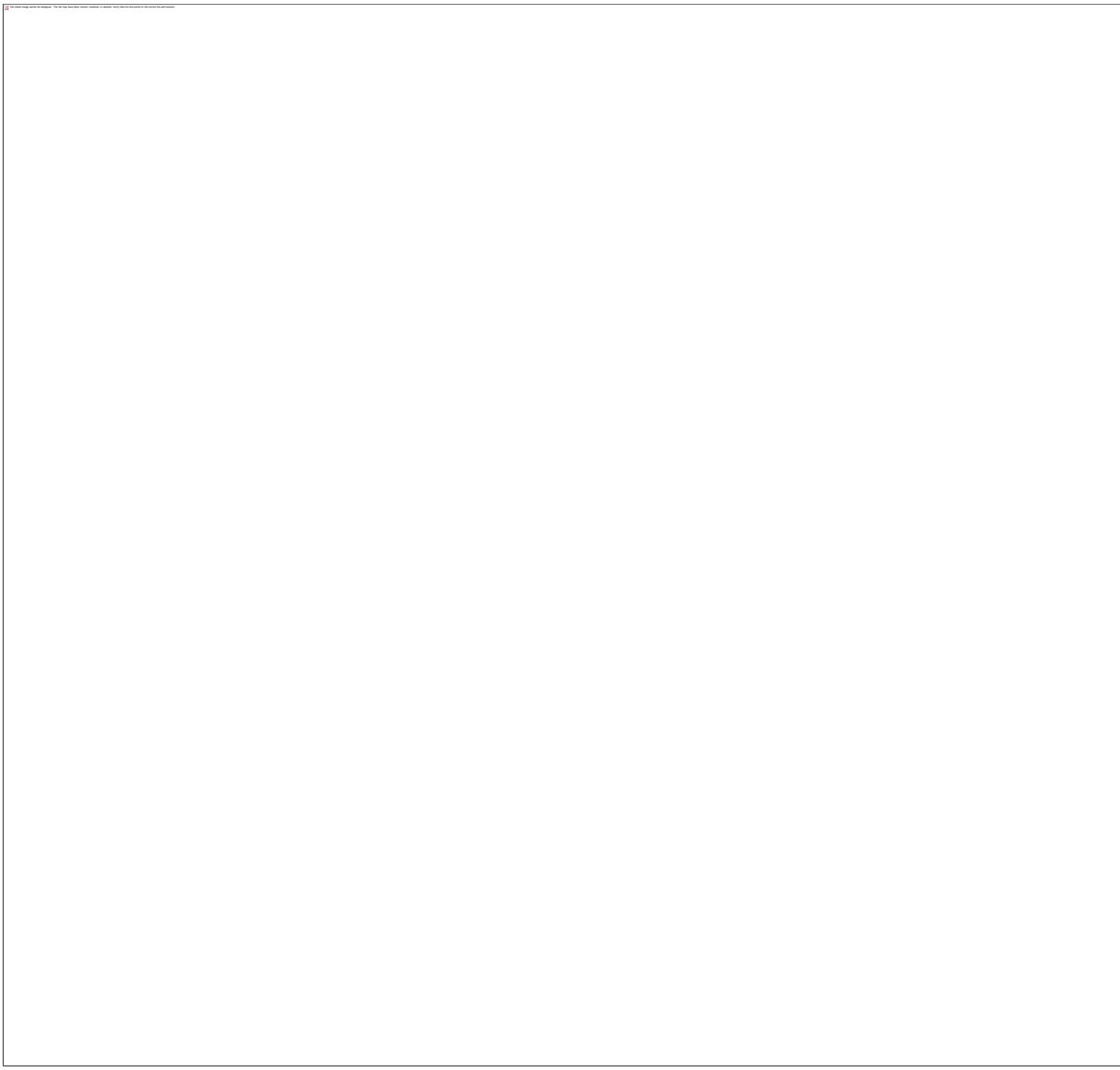
Here is the call graph for this function:

1.26.3.3 BasisFunc makeprtb (Real *Lx*, Real *Lz*, Real *a*, Real *b*, int *Ny*)

References chebypoints(), diff(), ComplexChebyCoeff::im, L2Norm(), BasisFunc::makeSpectral(), Physical, pi, ComplexChebyCoeff::re, BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

```
320                                                {
321   BasisFunc rtn(Ny, 1, 1, Lx, Lz, a, b, Physical);
322
323   Vector y = chebypoints(Ny,-1,1);
324   for (int n=0; n<Ny; ++n) {
325     Real yp = 1 + 2*(y[n]-b)/(b-a);
326     Real p = 1-yp*yp;
327     Real pp = p*p;
328     rtn.u().re[n] = p*(1+yp);
329     rtn.v().im[n] = pp*(1+yp);
330   }
331   rtn.makeSpectral();
332   ChebyCoeff tmp;
333
334   // Build up Re(w) = -Lz/Lx  Re(u) - Lz/2pikz Im(v')
335   //                = -Lz/Lx [Re(u) + Lx/2pikz Im(v')]
336   diff(rtn.v().im, tmp);
337   rtn.w().re = tmp;
338   rtn.w().re *= Lx/(2*pi);
339   rtn.w().re += rtn.u().re;
340   rtn.w().re *= -Lz/Lx;
341
342   rtn *= 1.0/L2Norm(rtn);
343   return rtn;
344 }
```

Here is the call graph for this function:



1.26.3.4 BasisFunc makeroll (Real *Lx*, Real *Lz*, Real *a*, Real *b*, int *Ny*)

References cferror(), chebypoints(), diff(), ComplexChebyCoeff::im, L2Norm(), ChebyCoeff::makeSpectral(), Physical, pi, pow(), ComplexChebyCoeff::re, ChebyCoeff::setState(), ChebyCoeff::setToZero(), Spectral, BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

```
218                                                {
219   if (Ny < 5)
220     cferror("Ny too small. Set Ny >= 5");
221
222   BasisFunc rtn(Ny, 0, 1, Lx, Lz, a, b, Spectral);
223
224   rtn.v().re[0] =  3.0/32.0;
225   rtn.v().re[2] = -1.0/8.0;
226   rtn.v().re[4] =  1.0/32.0;
227
228   ChebyCoeff tmp = diff(rtn.v().re);
229   rtn.w().im = tmp;
230   rtn.w().im *= Lz/(2*pi);
231
232   tmp.setToZero();
233   tmp.setState(Physical);
234   Vector y = chebypoints(Ny,-1,1);
235   for (int n=0; n<Ny; ++n) {
236     Real yp = 1 + 2*(y[n]-b)/(b-a);
237     tmp[n] = 2*(pow(yp,6) - 0.7*pow(yp,2) - 0.3);
238   }
239   tmp.makeSpectral();
240   rtn.u().re = tmp;
241
242   rtn *= -1.0/L2Norm(rtn);
243   return rtn;
244 }
```

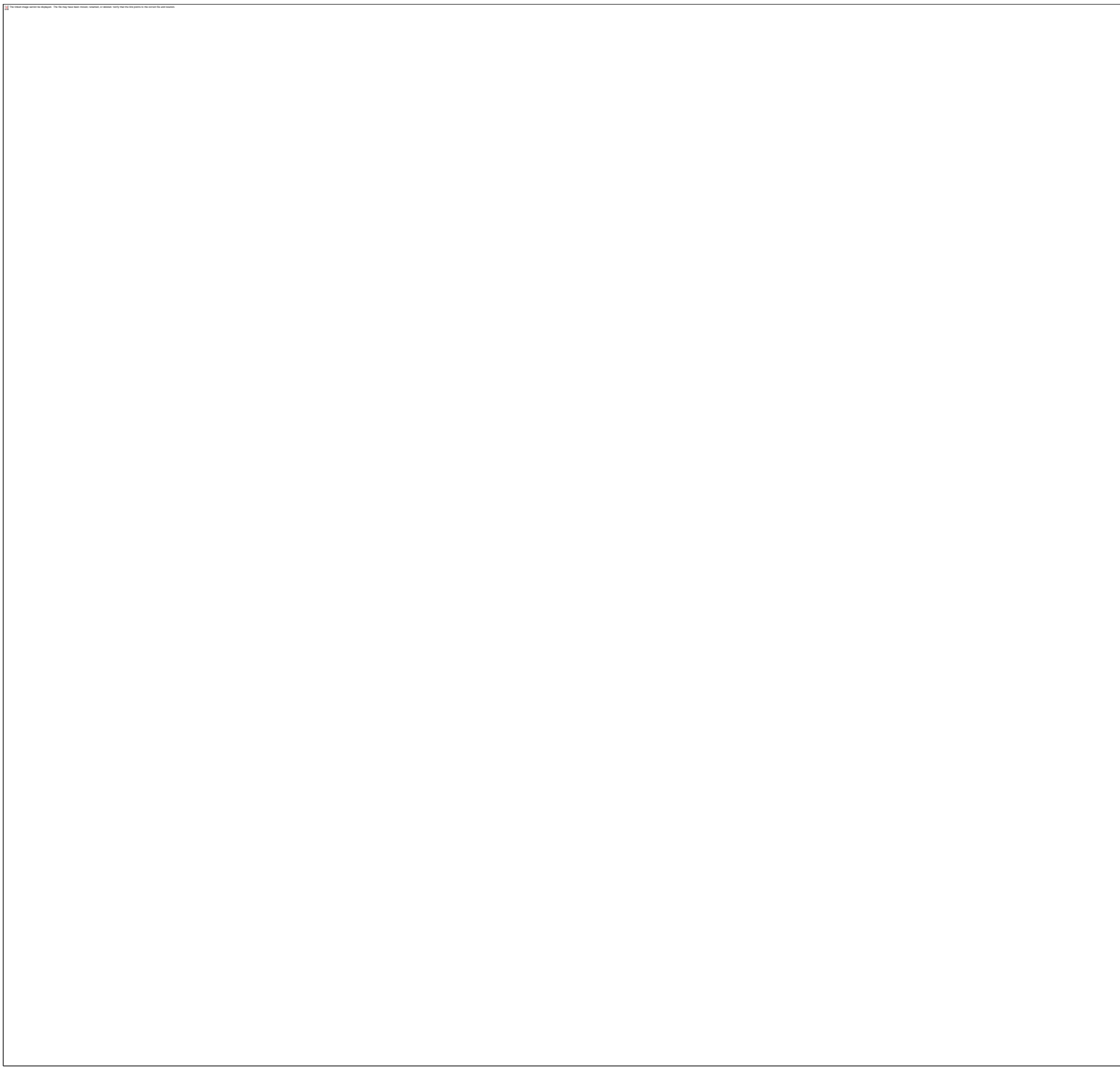
Here is the call graph for this function:

1.26.3.5 BasisFunc makewave (Real *Lx*, Real *Lz*, Real *a*, Real *b*, int *Ny*)

References chebypoints(), diff(), ComplexChebyCoeff::im, L2Norm(), BasisFunc::makeSpectral(), Physical, pi, ComplexChebyCoeff::re, BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

```
246                                                 {
247   BasisFunc rtn(Ny, 1, 0, Lx, Lz, a, b, Physical);
248
249   Vector y = chebypoints(Ny,-1,1);
250   for (int n=0; n<Ny; ++n) {
251     Real yp = 1 + 2*(y[n]-b)/(b-a);
252     Real p = 1-yp*yp;
253     Real pp = p*p;
254     rtn.w().re[n] = 0.7*p*sin(pi*yp);
255     rtn.w().im[n] = p*cos(pi*yp);
256     rtn.v().re[n] = 0.04*pp*sin(pi*yp);
257     rtn.v().im[n] = 0.04*pp*cos(pi*yp);
258   }
259   rtn.makeSpectral();
260   ChebyCoeff tmp = diff(rtn.v().re);
261   rtn.u().im = tmp;
262   rtn.u().im *= Lx/(2*pi);
263
264   tmp = diff(rtn.v().im);
265   rtn.u().re = tmp;
266   rtn.u().re *= -Lx/(2*pi);
267
268   rtn *= 1.0/L2Norm(rtn);
269   return rtn;
270 }
```

Here is the call graph for this function:



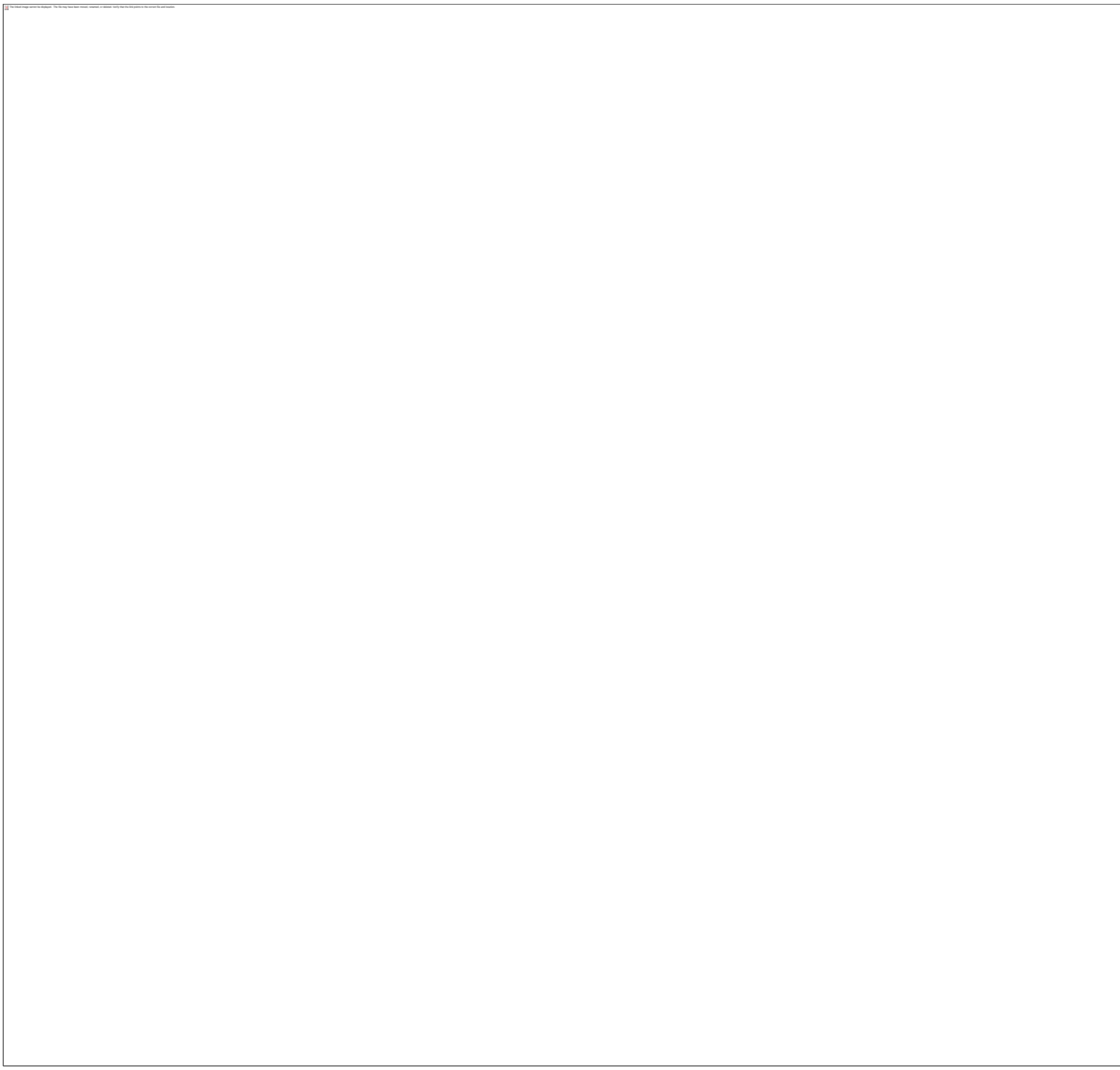
1.26.3.6 BasisFunc makewobl (Real *Lx*, Real *Lz*, Real *a*, Real *b*, int *Ny*)

References chebypoints(), diff(), ComplexChebyCoeff::im, L2Norm(), BasisFunc::makeSpectral(), Physical, pi, ComplexChebyCoeff::re, BasisFunc::u(), BasisFunc::v(), and BasisFunc::w().

```
283                                         {
```

```
284   BasisFunc rtn(Ny, 1, 1, Lx, Lz, a, b, Physical);
285
286   Vector y = chebypoints(Ny,-1,1);
287   for (int n=0; n<Ny; ++n) {
288     Real yp = 1 + 2*(y[n]-b)/(b-a);
289     Real p = 1-yp*yp;
290     Real pp = p*p;
291     rtn.u().re[n] = p*cos(pi*yp);
292     rtn.u().im[n] = -p*sin(pi*yp);
293     rtn.v().re[n] = 0.2*pp*cos(pi*yp);
294     rtn.v().im[n] = -0.2*pp*sin(pi*yp);
295   }
296   rtn.makeSpectral();
297   ChebyCoeff tmp;
298
299   // Build up Re(w) = -Lz/Lx  Re(u) - Lz/2pikz Im(v')
300   //                = -Lz/Lx [Re(u) + Lx/2pikz Im(v')]
301   diff(rtn.v().im, tmp);
302   rtn.w().re = tmp;
303   rtn.w().re *= Lx/(2*pi);
304   rtn.w().re += rtn.u().re;
305   rtn.w().re *= -Lz/Lx;
306
307   // Build up Im(w) = -Lz/Lx  Im(u) + Lz/2pikz Re(v')
308   //                = -Lz/Lx [Im(u) - Lx/2pikz Re(v')]
309   diff(rtn.v().re, tmp);
310   rtn.w().im = tmp;
311   rtn.w().im *= -Lx/(2*pi);
312   rtn.w().im += rtn.u().im;
313   rtn.w().im *= -Lz/Lx;
314
315   rtn *= 1.0/L2Norm(rtn);
316   return rtn;
317 }
```

Here is the call graph for this function:



1.26.3.7 Real polyInterp (const array< Real > & *fn*, const array< Real > & *xn*, Real *x*)

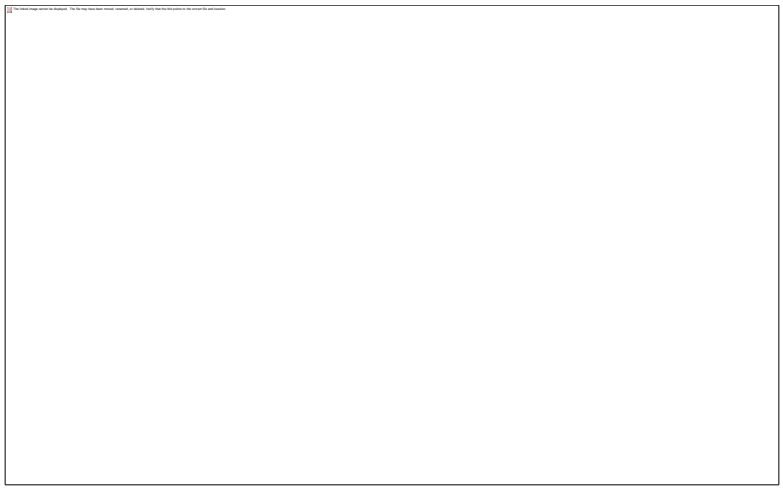
References abs(), cferror(), array< T >::N(), and xdiff().

Referenced by bisectSearch(), FieldSeries::interpolate(), and secantSearch().

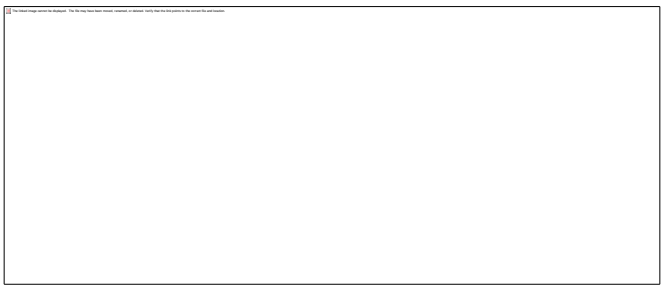
```
36                                                  {
37   assert(fa.N() == xa.N());
```

```
38   int N = fa.N();
39   array<Real> C(N);
40   array<Real> D(N);
41
42   Real xdiff = abs(x-xa[0]);
43   Real xdiff2 = 0.0;
44   Real df;
45
46   int i_closest = 0;
47   for (int i=0; i<N; ++i) {
48     if ((xdiff2 = abs(x-xa[i])) < xdiff) {
49       i_closest = i;
50       xdiff = xdiff2;
51     }
52     C[i] = fa[i];
53     D[i] = fa[i];
54   }
55
56   // Initial approx to f(x)
57   Real fx = fa[i_closest--];
58
59   for (int m=1; m<N; ++m) {
60     for (int i=0; i<N-m; ++i) {
61       Real C_dx = xa[i]-x;
62       Real D_dx = xa[i+m] - x;
63       Real C_D = C[i+1]-D[i];
64       Real denom = C_dx - D_dx;
65       if (denom == 0.0)
66         cferror("polyinterp(fa,xa,x) : two values of xa has are equal!");
67       denom = C_D/denom;
68       C[i] = C_dx*denom;
69       D[i] = D_dx*denom;
70     }
71     fx += (df=(2*(i_closest+1) < (N-m)? C[i_closest+1] : D[i_closest--]));
72   }
73   return fx;
74 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



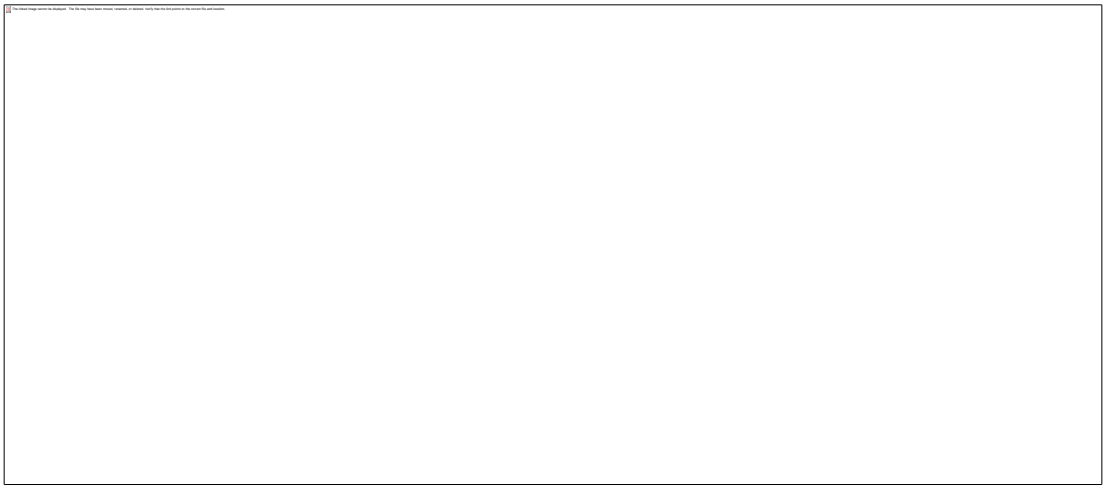
1.26.3.8 Real secantSearch (Real *a*, Real *b*, array< Real > & *fn*, const array< Real > & *xn*, Real *feps* = 1e-14, int *maxsteps* = 50)

References abs(), cferror(), and polyInterp().

```
77                                      {
78   Real fa = polyInterp(fn, xn, a);
79   Real fb = polyInterp(fn, xn, b);
80   if (fa*fb > 0)
81     cferror("secantSearch(a,b,fn,xn) : a and b don't bracket a zero");
82
83   Real c = a - fa * (b-a)/(fb-fa);
84   Real fc;
85
86   for (int n=0; n<maxsteps; ++n) {
87     c = a - fa * (b-a)/(fb-fa);
88     fc = polyInterp(fn,xn,c);
89     if (abs(fc) < feps)
90       break;
91     if (fc*fa > 0) {
92       a = c;
```

```
93       fa = fc;
94     }
95     else {
96       b = c;
97       fb = fc;
98     }
99   }
100   return c;
101 }
```

Here is the call graph for this function:



1.26.3.9

1.27 /_cuda/channelflow-0.9.22/channelflow/vector.cpp File Reference

#include <fstream>
#include <iomanip>
#include <fftw3.h>
#include "channelflow/vector.h"
Include dependency graph for vector.cpp:

1.27.1 Functions

- void swap (Vector &f, Vector &g)
- void assign (Vector &u, int uistart, int uistride, int uiend, Vector &v, int vistart, int vistride, int viend)
- Vector operator* (Real c, const Vector &v)
- Vector operator+ (const Vector &u, const Vector &v)
- Vector operator- (const Vector &u, const Vector &v)
- Vector dottimes (const Vector &u, const Vector &v)
- Vector dotdivide (const Vector &u, const Vector &v)
- Real operator* (const Vector &u, const Vector &v)
- bool operator== (const Vector &u, const Vector &v)
- Vector abs (const Vector &v)
- Real L1Norm (const Vector &v)
- Real L2Norm (const Vector &v)
- Real L2Norm2 (const Vector &v)
- Real LinfNorm (const Vector &v)
- Real L1Dist (const Vector &u, const Vector &v)
- Real L2Dist (const Vector &u, const Vector &v)
- Real L2Dist2 (const Vector &u, const Vector &v)
- Real LinfDist (const Vector &u, const Vector &v)
- Real mean (const Vector &v)

- int maxElemIndex (const Vector &v)
- ostream & operator<< (ostream &os, const Vector &a)

1.27.2 Function Documentation

1.27.2.1 Vector abs (const Vector & *v*)

References Vector::abs().

Referenced by FlowField::addPerturbations(), bisectSearch(), FlowField::CFLfactor(), chebyProfile(), chebyUprofile(), BandedTridiag::elem(), i2s(), FlowField::isAliased(), DNSAlgorithm::isAliasedMode(), L1Dist(), L1Norm(), L2Dist2(), L2Norm2(), LinfDist(), LinfNorm(), log(), polyInterp(), BandedTridiag::print(), randomProfile(), randomUprofile(), HelmholtzSolver::residual(), roundReal(), secantSearch(), TauSolver::TauSolver(), TurbStats::ustar(), TauSolver::verify(), and TauSolver::verify_P_and_v().

```
329                     {
330   Vector rtn(v);
331   rtn.abs();
332   return rtn;
333 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.27.2.2 void assign (Vector & *u*, int *uistart*, int *uistride*, int *uiend*, Vector & *v*, int *vistart*, int *vistride*, int *viend*)

```
258                                                 {
259   assert(((uistart - uiend)/uistride == (vistart - viend)/vistride));
260   int ui, vi;
261   for (ui=uistart, vi=vistart; ui<uiend; ui+=uistride, vi+=vistride)
262       u[ui] = v[vi];
263 }
```

1.27.2.3 Vector dotdivide (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
300                                         {
301   Vector w(v.length());
302   int N=u.length();
303   assert(v.length() == N);
304   for (int i=0; i<N; ++i)
305     w[i] = u[i]/v[i];
306   return w;
307 }
```

Here is the call graph for this function:

1.27.2.4 Vector dottimes (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
291                                         {
292   Vector w(v.length());
293   int N=u.length();
294   assert(v.length() == N);
295   for (int i=0; i<N; ++i)
296     w[i] = u[i] * v[i];
297   return w;
298 }
```

Here is the call graph for this function:

1.27.2.5 Real L1Dist (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
367                                     {
368   assert(u.length() == v.length());
369   Real sum = 0.0;
370   int N=v.length();
371   for (int i=0; i<N; ++i)
372     sum += fabs(u[i] - v[i]);
373   return sum;
374 }
```

Here is the call graph for this function:

1.27.2.6 Real L1Norm (const Vector & *v*)

References Vector::length().

```
335                         {
336   Real sum = 0.0;
337   int N=v.length();
338   for (int i=0; i<N; ++i)
339     sum += fabs(v[i]);
340   return sum;
341 }
```

Here is the call graph for this function:

1.27.2.7 Real L2Dist (const Vector & *u*, const Vector & *v*)

References Vector::length(), and square().

```
377                                     {
378   assert(u.length() == v.length());
```

```
379   Real sum = 0.0;
380   int N=v.length();
381   for (int i=0; i<N; ++i)
382     sum += square(v[i]-u[i]);
383   return sqrt(sum);
384 }
```

Here is the call graph for this function:

1.27.2.8 Real L2Dist2 (const Vector & *u*, const Vector & *v*)

References Vector::length(), and square().

```
386                                         {
387   assert(u.length() == v.length());
388   Real sum = 0.0;
389   int N=v.length();
390   for (int i=0; i<N; ++i)
391     sum += square(u[i]-v[i]);
392   return sum;
393 }
```

Here is the call graph for this function:

1.27.2.9 Real L2Norm (const Vector & *v*)

References Vector::length(), and square().

```
343                             {
344   Real sum = 0.0;
345   int N=v.length();
346   for (int i=0; i<N; ++i)
347     sum += square(v[i]);
```

```
348   return sqrt(sum);
349 }
```

Here is the call graph for this function:

1.27.2.10 Real L2Norm2 (const Vector & *v*)

References Vector::length(), and square().

```
351                         {
352   Real sum = 0.0;
353   int N=v.length();
354   for (int i=0; i<N; ++i)
355     sum += square(v[i]);
356   return sum;
357 }
```

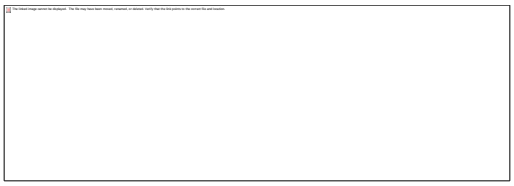
Here is the call graph for this function:

1.27.2.11 Real LinfDist (const Vector & *u*, const Vector & *v*)

References Greater(), and Vector::length().

```
395                                         {
396   assert(u.length() == v.length());
397   Real max = 0.0;
398   int N=v.length();
399   for (int i=0; i<N; ++i)
400     max = Greater(fabs(v[i]-u[i]), max);
401   return max;
402 }
```

Here is the call graph for this function:



1.27.2.12 Real LinfNorm (const Vector & *v*)

References Greater(), and Vector::length().

```
359                         {
360   Real max = 0.0;
361   int N=v.length();
362   for (int i=0; i<N; ++i)
363     max = Greater(fabs(v[i]), max);
364   return max;
365 }
```

Here is the call graph for this function:

1.27.2.13 int maxElemIndex (const Vector & *v*)

References Vector::length().

```
412                           {
413   Real max = 0.0;
414   int index = 0;
415   int N=v.length();
416   for (int i=0; i<N; ++i)
417     if (fabs(v[i]) > max) {
418       max = fabs(v[i]);
419       index = i;
420     }
421   return index;
422 }
```

Here is the call graph for this function:

1.27.2.14 Real mean (const Vector & *v*)

References Vector::length().

```
404                         {
405   Real sum = 0.0;
406   int N = v.length();
407   for (int i=0; i<N; ++i)
408     sum += v[i];
409   return sum/N;
410 }
```

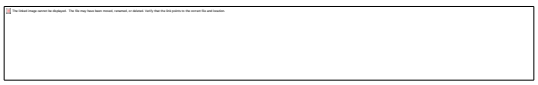
Here is the call graph for this function:

1.27.2.15 Real operator* (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
309                                     {
310   Real sum = 0.0;
311   int N=u.length();
312   assert(v.length() == N);
313   for (int i=0; i<N; ++i)
314     sum += u[i]*v[i];
315   return sum;
316 }
```

Here is the call graph for this function:



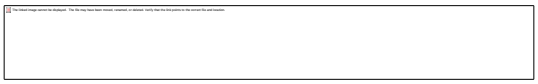
1.27.2.16 Vector operator* (Real *c*, const Vector & *v*)

References Vector::length().

```
265                                 {
266   Vector u(v.length());
```

```
267   int N=u.length();
268   for (int i=0; i<N; ++i)
269     u[i] = c*v[i];
270   return u;
271 }
```

Here is the call graph for this function:



1.27.2.17 Vector operator+ (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
273                                     {
274   Vector w(v.length());
275   int N=u.length();
276   assert(v.length() == N);
277   for (int i=0; i<N; ++i)
278     w[i] = u[i] + v[i];
279   return w;
280 }
```

Here is the call graph for this function:

1.27.2.18 Vector operator- (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
282                                     {
283   Vector w(v.length());
284   int N=u.length();
285   assert(v.length() == N);
286   for (int i=0; i<N; ++i)
287     w[i] = u[i] - v[i];
288   return w;
289 }
```

Here is the call graph for this function:

1.27.2.19 ostream& operator<< (ostream & *os*, const Vector & *a*)

References Vector::length().

```
424                                     {
425   int N = a.length();
426   char seperator=(N<10) ? ' ' : '\n';
427   for (int i=0; i<N; ++i) {
428     os << a[i];
429     os << seperator;
430     //os << a[i] << seperator;
431   }
432   return os;
433 }
```

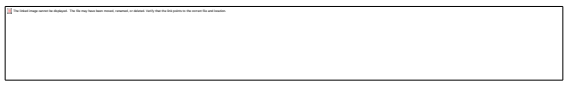
Here is the call graph for this function:

1.27.2.20 bool operator== (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
318                                     {
319   int N=u.length();
320   if (v.length() != N)
321     return false;
322   for (int i=0; i<N; ++i)
323     if (u[i] != v[i])
324       return false;
325
326   return true;
327 }
```

Here is the call graph for this function:



1.27.2.21 void swap (Vector & *f*, Vector & *g*)

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

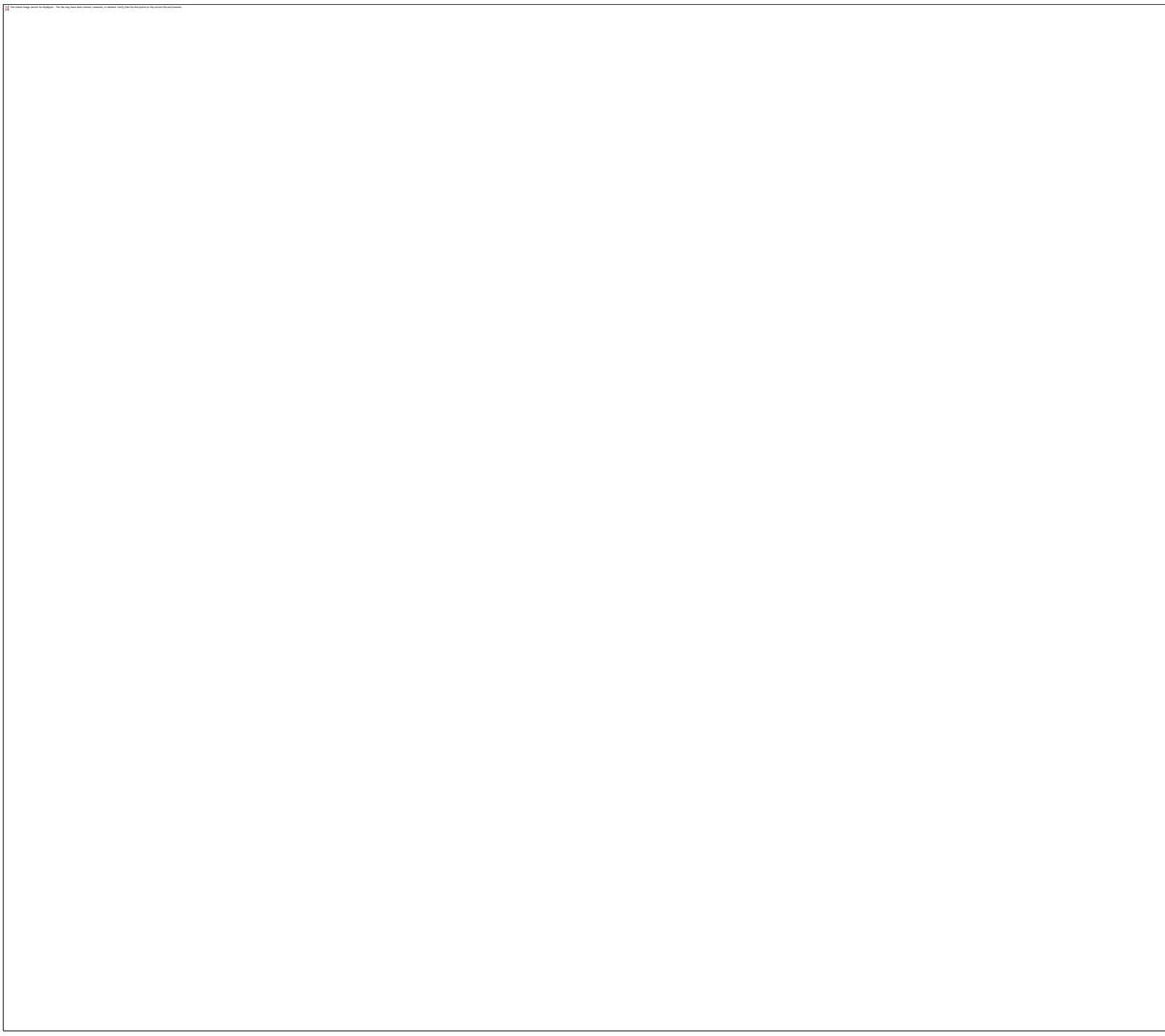
```
247                         {
248   int itmp = f.N_;
249   f.N_ = g.N_;
250   g.N_ = itmp;
251
252   Real* tmp = f.data_;
253   f.data_ = g.data_;
254   g.data_ = tmp;
255 }
```

1.27.2.22

1.28 /_cuda/channelflow-0.9.22/channelflow/vector.h File Reference

#include "channelflow/mathdefs.h"
Include dependency graph for vector.h:

This graph shows which files directly or indirectly include this file:



1.28.1 Classes

- class Vector

1.28.2 Functions

- void assign (Vector &u, int uistart, int uistride, int uiend, Vector &v, int vistart, int vistride, int viend)
- Vector operator* (Real c, const Vector &v)
- Vector operator+ (const Vector &u, const Vector &v)
- Vector operator- (const Vector &u, const Vector &v)

- Real operator* (const Vector &u, const Vector &v)
- bool operator== (const Vector &u, const Vector &v)
- Vector dottimes (const Vector &u, const Vector &v)
- Vector dotdivide (const Vector &u, const Vector &v)
- Real L1Norm (const Vector &v)
- Real L2Norm (const Vector &v)
- Real L2Norm2 (const Vector &v)
- Real LinfNorm (const Vector &v)
- Real L1Dist (const Vector &u, const Vector &v)
- Real L2Dist (const Vector &u, const Vector &v)
- Real L2Dist2 (const Vector &u, const Vector &v)
- Real LinfDist (const Vector &u, const Vector &v)
- Real mean (const Vector &v)
- int maxElemIndex (const Vector &v)
- Vector abs (const Vector &v)
- std::ostream & operator<< (std::ostream &, const Vector &a)

1.28.3 Function Documentation

1.28.3.1 Vector abs (const Vector & *v*)

References Vector::abs().

Referenced by FlowField::addPerturbations(), bisectSearch(), FlowField::CFLfactor(), chebyProfile(), chebyUprofile(), BandedTridiag::elem(), i2s(), FlowField::isAliased(), DNSAlgorithm::isAliasedMode(), L1Dist(), L1Norm(), L2Dist2(), L2Norm2(), LinfDist(), LinfNorm(), log(), polyInterp(), BandedTridiag::print(), randomProfile(), randomUprofile(), HelmholtzSolver::residual(), roundReal(), secantSearch(), TauSolver::TauSolver(), TurbStats::ustar(), TauSolver::verify(), and TauSolver::verify_P_and_v().

```
329                     {
330   Vector rtn(v);
331   rtn.abs();
332   return rtn;
333 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

1.28.3.2 void assign (Vector & *u*, int *uistart*, int *uistride*, int *uiend*, Vector & *v*, int *vistart*, int *vistride*, int *viend*)

```
258                                                 {
259   assert(((uistart - uiend)/uistride == (vistart - viend)/vistride));
260   int ui, vi;
261   for (ui=uistart, vi=vistart; ui<uiend; ui+=uistride, vi+=vistride)
262       u[ui] = v[vi];
263 }
```

1.28.3.3 Vector dotdivide (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
300                                         {
301   Vector w(v.length());
302   int N=u.length();
303   assert(v.length() == N);
304   for (int i=0; i<N; ++i)
305     w[i] = u[i]/v[i];
306   return w;
307 }
```

Here is the call graph for this function:

1.28.3.4 Vector dottimes (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
291                                         {
292   Vector w(v.length());
293   int N=u.length();
294   assert(v.length() == N);
295   for (int i=0; i<N; ++i)
296     w[i] = u[i] * v[i];
297   return w;
298 }
```

Here is the call graph for this function:

1.28.3.5 Real L1Dist (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
367                                     {
368   assert(u.length() == v.length());
369   Real sum = 0.0;
370   int N=v.length();
371   for (int i=0; i<N; ++i)
372     sum += fabs(u[i] - v[i]);
373   return sum;
374 }
```

Here is the call graph for this function:

1.28.3.6 Real L1Norm (const Vector & *v*)

References Vector::length().

```
335                         {
336   Real sum = 0.0;
337   int N=v.length();
338   for (int i=0; i<N; ++i)
339     sum += fabs(v[i]);
340   return sum;
341 }
```

Here is the call graph for this function:

1.28.3.7 Real L2Dist (const Vector & *u*, const Vector & *v*)

References Vector::length(), and square().

```
377                                     {
378   assert(u.length() == v.length());
```

```
379   Real sum = 0.0;
380   int N=v.length();
381   for (int i=0; i<N; ++i)
382     sum += square(v[i]-u[i]);
383   return sqrt(sum);
384 }
```

Here is the call graph for this function:

1.28.3.8 Real L2Dist2 (const Vector & *u*, const Vector & *v*)

References Vector::length(), and square().

```
386                                         {
387   assert(u.length() == v.length());
388   Real sum = 0.0;
389   int N=v.length();
390   for (int i=0; i<N; ++i)
391     sum += square(u[i]-v[i]);
392   return sum;
393 }
```

Here is the call graph for this function:

1.28.3.9 Real L2Norm (const Vector & *v*)

References Vector::length(), and square().

```
343                             {
344   Real sum = 0.0;
345   int N=v.length();
346   for (int i=0; i<N; ++i)
347     sum += square(v[i]);
```

```
348   return sqrt(sum);
349 }
```

Here is the call graph for this function:

1.28.3.10 Real L2Norm2 (const Vector & *v*)

References Vector::length(), and square().

```
351                         {
352   Real sum = 0.0;
353   int N=v.length();
354   for (int i=0; i<N; ++i)
355     sum += square(v[i]);
356   return sum;
357 }
```

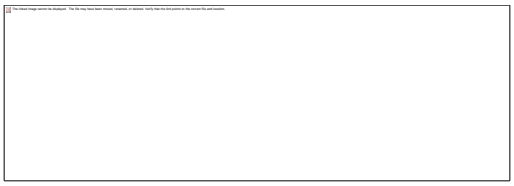
Here is the call graph for this function:

1.28.3.11 Real LinfDist (const Vector & *u*, const Vector & *v*)

References Greater(), and Vector::length().

```
395                                         {
396   assert(u.length() == v.length());
397   Real max = 0.0;
398   int N=v.length();
399   for (int i=0; i<N; ++i)
400     max = Greater(fabs(v[i]-u[i]), max);
401   return max;
402 }
```

Here is the call graph for this function:



1.28.3.12 Real LinfNorm (const Vector & *v*)

References Greater(), and Vector::length().

```
359                         {
360   Real max = 0.0;
361   int N=v.length();
362   for (int i=0; i<N; ++i)
363     max = Greater(fabs(v[i]), max);
364   return max;
365 }
```

Here is the call graph for this function:

1.28.3.13 int maxElemIndex (const Vector & *v*)

References Vector::length().

```
412                           {
413   Real max = 0.0;
414   int index = 0;
415   int N=v.length();
416   for (int i=0; i<N; ++i)
417     if (fabs(v[i]) > max) {
418       max = fabs(v[i]);
419       index = i;
420     }
421   return index;
422 }
```

Here is the call graph for this function:

1.28.3.14 Real mean (const Vector & *v*)

References Vector::length().

```
404                         {
405   Real sum = 0.0;
406   int N = v.length();
407   for (int i=0; i<N; ++i)
408     sum += v[i];
409   return sum/N;
410 }
```

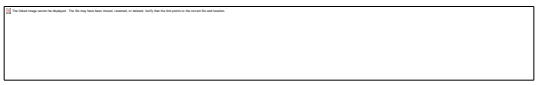
Here is the call graph for this function:

1.28.3.15 Real operator* (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
309                                     {
310   Real sum = 0.0;
311   int N=u.length();
312   assert(v.length() == N);
313   for (int i=0; i<N; ++i)
314     sum += u[i]*v[i];
315   return sum;
316 }
```

Here is the call graph for this function:



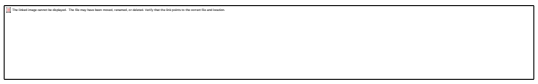
1.28.3.16 Vector operator* (Real *c*, const Vector & *v*)

References Vector::length().

```
265                                 {
266   Vector u(v.length());
```

```
267   int N=u.length();
268   for (int i=0; i<N; ++i)
269     u[i] = c*v[i];
270   return u;
271 }
```

Here is the call graph for this function:



1.28.3.17 Vector operator+ (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
273                                     {
274   Vector w(v.length());
275   int N=u.length();
276   assert(v.length() == N);
277   for (int i=0; i<N; ++i)
278     w[i] = u[i] + v[i];
279   return w;
280 }
```

Here is the call graph for this function:

1.28.3.18 Vector operator- (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
282                                     {
283   Vector w(v.length());
284   int N=u.length();
285   assert(v.length() == N);
286   for (int i=0; i<N; ++i)
287     w[i] = u[i] - v[i];
288   return w;
289 }
```

Here is the call graph for this function:

1.28.3.19 std::ostream& operator<< (std::ostream &, const Vector & *a*)

References Vector::length().

```
424                                     {
425   int N = a.length();
426   char seperator=(N<10) ? ' ' : '\n';
427   for (int i=0; i<N; ++i) {
428     os << a[i];
429     os << seperator;
430     //os << a[i] << seperator;
431   }
432   return os;
433 }
```

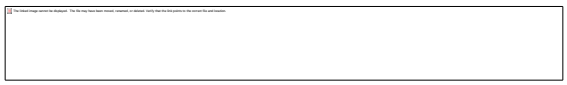
Here is the call graph for this function:

1.28.3.20 bool operator== (const Vector & *u*, const Vector & *v*)

References Vector::length().

```
318                                     {
319   int N=u.length();
320   if (v.length() != N)
321     return false;
322   for (int i=0; i<N; ++i)
323     if (u[i] != v[i])
324       return false;
325
326   return true;
327 }
```

Here is the call graph for this function:



2 Index

/_cuda/channelflow-0.9.22/channelflow/array.h, 502
/_cuda/channelflow-0.9.22/channelflow/bandedtridiag.cpp, 504
/_cuda/channelflow-0.9.22/channelflow/bandedtridiag.h, 505
/_cuda/channelflow-0.9.22/channelflow/basisfunc.cpp, 506
/_cuda/channelflow-0.9.22/channelflow/basisfunc.h, 556
/_cuda/channelflow-0.9.22/channelflow/chebyshev.cpp, 607
/_cuda/channelflow-0.9.22/channelflow/chebyshev.h, 660
/_cuda/channelflow-0.9.22/channelflow/complexdefs.h, 715
/_cuda/channelflow-0.9.22/channelflow/diffops.cpp, 717
/_cuda/channelflow-0.9.22/channelflow/diffops.h, 798
/_cuda/channelflow-0.9.22/channelflow/dns.cpp, 878
/_cuda/channelflow-0.9.22/channelflow/dns.h, 883
/_cuda/channelflow-0.9.22/channelflow/flowfield.cpp, 890
/_cuda/channelflow-0.9.22/channelflow/flowfield.h, 937
/_cuda/channelflow-0.9.22/channelflow/helmholtz.cpp, 986
/_cuda/channelflow-0.9.22/channelflow/helmholtz.h, 987
/_cuda/channelflow-0.9.22/channelflow/mathdefs.cpp, 988
/_cuda/channelflow-0.9.22/channelflow/mathdefs.h, 1001
/_cuda/channelflow-0.9.22/channelflow/poissonsolver.cpp, 1024
/_cuda/channelflow-0.9.22/channelflow/poissonsolver.h, 1025
/_cuda/channelflow-0.9.22/channelflow/tausolver.cpp, 1026
/_cuda/channelflow-0.9.22/channelflow/tausolver.h, 1029
/_cuda/channelflow-0.9.22/channelflow/turbstats.cpp, 1032
/_cuda/channelflow-0.9.22/channelflow/turbstats.h, 1033
/_cuda/channelflow-0.9.22/channelflow/utilfuncs.cpp, 1034
/_cuda/channelflow-0.9.22/channelflow/utilfuncs.h, 1045
/_cuda/channelflow-0.9.22/channelflow/vector.cpp, 1056
/_cuda/channelflow-0.9.22/channelflow/vector.h, 1065
~array
  array, 10
~BandedTridiag
  BandedTridiag, 19
~ChebyCoeff
  ChebyCoeff, 74
~ChebyTransform
  ChebyTransform, 106
~CNABstyleDNS
  CNABstyleDNS, 120
~DNS
  DNS, 158
~DNSAlgorithm
  DNSAlgorithm, 175
~FlowField
  FlowField, 211
~MultistepDNS
  MultistepDNS, 348
~PoissonSolver
  PoissonSolver, 362
~PPEDNS
  PPEDNS, 382
~PressureSolver
  PressureSolver, 395
~RungeKuttaDNS
  RungeKuttaDNS, 408
~Vector

Vector, 491
a
BasisFunc, 41
ChebyCoeff, 74
ComplexChebyCoeff, 134
DNSAlgorithm, 176
FlowField, 211
a_
BandedTridiag, 31
BasisFunc, 65
ChebyCoeff, 104
DNSAlgorithm, 184
FlowField, 318
HelmholtzSolver, 334
PoissonSolver, 372
TauSolver, 455
A_
RungeKuttaDNS, 414
abs
Vector, 491
vector.cpp, 1056
vector.h, 1066
abs2
mathdefs.h, 1003
add
ComplexChebyCoeff, 135
addData
TurbStats, 475
addPerturbation
FlowField, 213
addPerturbations
FlowField, 215, 216
addProfile
FlowField, 217, 218, 219
adjust
TimeStep, 468
advance
CNABstyleDNS, 121
DNS, 158
DNSAlgorithm, 176
MultistepDNS, 348
PPEDNS, 382
RungeKuttaDNS, 408
Ae_
HelmholtzSolver, 335
alpha_
CNABstyleDNS, 128
MultistepDNS, 356
PPEDNS, 390
Alternating
diffops.h, 799
Alternating_
diffops.h, 799
Ao_
HelmholtzSolver, 335
array, 6
~array, 10
array, 9
binaryDump, 10
binaryLoad, 10
data_, 14
fill, 11
length, 11
N, 11
N_, 14
operator!=, 11
operator[], 12
operator=, 12
operator==, 12
pointer, 13
resize, 13
save, 14
subvector, 14
array.h
operator<<, 503
Real, 503
REAL_OUTPUT_DIGITS, 503
asciiSave
FlowField, 220
assertState
FlowField, 221
assign
vector.cpp, 1058
vector.h, 1067
assignOrrSommField
flowfield.cpp, 891
flowfield.h, 938
Attributes, 15
Attributes, 15
kxmax, 16
kzmax, 16
Lx, 16

Lz, 16
operator=, 15
setAttributes, 16
b
BasisFunc, 41
ChebyCoeff, 76
ComplexChebyCoeff, 136
DNSAlgorithm, 176
FlowField, 222
b_
BasisFunc, 65
ChebyCoeff, 104
DNSAlgorithm, 185
FlowField, 318
HelmholtzSolver, 335
PoissonSolver, 372
TauSolver, 455
B_
RungeKuttaDNS, 414
band
BandedTridiag, 20
BandedTridiag, 17
~BandedTridiag, 19
a_, 31
band, 20
BandedTridiag, 17, 18
d_, 31
diag, 20
elem, 21, 22
lodiag, 23
M_, 31
Mbar_, 32
multiply, 23
multiplyStrided, 24
numNonzeros, 25
numrows, 25
numRows, 25
operator=, 25
operator==, 26
print, 26
save, 26
test, 27
UL_, 32
ULdecomp, 27
ULprint, 28
ULsolve, 29
ULsolveStrided, 29
updiag, 30, 31
bandedtridiag.cpp
EPSILON, 504
bandedtridiag.h
Real, 505
BaseTask, 33
BaseTask, 33
Run, 33
BasisFunc, 35
a, 41
a_, 65
b, 41
b_, 65
BasisFunc, 38, 39, 40
bcNorm, 42
binaryDump, 43
binaryLoad, 43
chebyfft, 43, 44
component, 44
congruent, 44
conjugate, 45
fill, 45
geometryCongruent, 45
ichebyfft, 46
interpolate, 47
kx, 47
kx_, 65
kz, 48
kz_, 65
lambda, 48
lambda_, 65
Lx, 48
Lx_, 66
Lz, 50
Lz_, 66
makePhysical, 51, 52
makeSpectral, 52, 53
makeState, 53
Nd, 54
Nd_, 66
Ny, 55
Ny_, 66
operator*=, 55
operator[], 56
operator+=, 56

operator-=, 56
randomize, 56
reflect, 58
resize, 58
rmsval, 59
save, 59
setBounds, 60
setkxkz, 60
setRmsval, 60
setState, 60
setToZero, 61
state, 61
state_, 66
u, 62
u_, 66
v, 63
w, 64
basisfunc.cpp
bcDist2, 507
bcNorm, 508, 509
bcNorm2, 510
chebyDist, 511
chebyDist2, 512
chebyInnerProduct, 513
chebyNorm, 514
chebyNorm2, 515
cross, 516
curl, 517
dist, 519
dist2, 520
divDist, 520
divDist2, 522
divergence, 524
divNorm, 526
divNorm2, 528
dot, 530
dotgrad, 531, 533
gradient, 536, 538
innerProduct, 540
L2Dist, 540
L2Dist2, 541
L2InnerProduct, 542
L2Norm, 543
L2Norm2, 544
laplacian, 545
norm, 547
norm2, 548
operator!=, 548
operator==, 548
ubcFix, 549
vbcFix, 550, 551
xdiff, 552
ydiff, 553
zdiff, 555
basisfunc.h
bcDist, 557
bcDist2, 557
bcNorm, 558
bcNorm2, 559
chebyDist, 560
chebyDist2, 561
chebyInnerProduct, 562
chebyNorm, 563
chebyNorm2, 564
cross, 565
curl, 566
dist, 569
dist2, 570
divDist, 570
divDist2, 572
divergence, 574
divNorm, 576
divNorm2, 578
dot, 580
dotgrad, 581, 583
gradient, 586, 588
innerProduct, 590
L2Dist, 590
L2Dist2, 591
L2InnerProduct, 592
L2Norm, 593
L2Norm2, 594
laplacian, 595
norm, 598
norm2, 599
operator!=, 599
operator==, 599
ubcFix, 600
vbcFix, 601, 602
xdiff, 603
ydiff, 604
zdiff, 606

bcDist
  basisfunc.h, 557
  flowfield.cpp, 893
  flowfield.h, 940
bcDist2
  basisfunc.cpp, 507
  basisfunc.h, 557
  flowfield.cpp, 893
  flowfield.h, 941
bcNorm
  BasisFunc, 42
  basisfunc.cpp, 508, 509
  basisfunc.h, 558
  flowfield.cpp, 894
  flowfield.h, 942
bcNorm2
  basisfunc.cpp, 510
  basisfunc.h, 559
  flowfield.cpp, 895
  flowfield.h, 943
Be_
  HelmholtzSolver, 335
beta
  HelmholtzSolver, 326
beta_
  CNABstyleDNS, 128
  MultistepDNS, 356
  PPEDNS, 390
binaryDump
  array, 10
  BasisFunc, 43
  ChebyCoeff, 78
  ComplexChebyCoeff, 136
binaryLoad
  array, 10
  BasisFunc, 43
  ChebyCoeff, 78
  ComplexChebyCoeff, 137
binarySave
  FlowField, 224
bisectSearch
  utilfuncs.cpp, 1034
  utilfuncs.h, 1046
Bo_
  HelmholtzSolver, 335
bulkReynolds
  TurbStats, 477
BulkVelocity
  dns.h, 885
c
  HelmholtzSolver, 326
C_
  RungeKuttaDNS, 414
call_fftw
  FlowField, 225
cdata_
  FlowField, 319
centerlineReynolds
  TurbStats, 477
cferror
  mathdefs.cpp, 989
  mathdefs.h, 1004
CFL
  DNS, 159
  DNSAlgorithm, 177
  TimeStep, 469
cfl_
  DNSAlgorithm, 185
CFL_
  TimeStep, 470
CFLfactor
  FlowField, 225, 226
CFLmax_
  TimeStep, 470
CFLmin_
  TimeStep, 470
cfpause
  mathdefs.cpp, 990
  mathdefs.h, 1005
changeBaseFlow
  dns.cpp, 878
  dns.h, 885
channelflowVersion
  mathdefs.cpp, 991
  mathdefs.h, 1006
cheby_c
  chebyshev.h, 662
ChebyCoeff, 67
  ~ChebyCoeff, 74
  a, 74
  a_, 104
  b, 76

b_, 104
binaryDump, 78
binaryLoad, 78
ChebyCoeff, 72, 73
chebyfft, 79
ChebyTransform, 104, 107
congruent, 80
eval, 80, 81
eval_a, 82
eval_b, 83
fill, 84
ichebyfft, 85
interpolate, 86
L, 86
makePhysical, 88
makeSpectral, 88, 89
makeState, 90
mean, 91
N, 92
numModes, 93
operator*=, 94, 95
operator+=, 95
operator-=, 95
randomize, 96
reflect, 96
save, 97
setBounds, 98
setState, 100
setToZero, 102
state, 102
state_, 104
swap, 104
chebyDist
basisfunc.cpp, 511
basisfunc.h, 560
chebyshev.cpp, 609
chebyshev.h, 663
chebyDist2
basisfunc.cpp, 512
basisfunc.h, 561
chebyshev.cpp, 610
chebyshev.h, 664
chebyEval
chebyshev.h, 665
chebyfft
BasisFunc, 43, 44
ChebyCoeff, 79
ComplexChebyCoeff, 137
chebyfft_y
FlowField, 227
chebyInnerProduct
basisfunc.cpp, 513
basisfunc.h, 562
chebyshev.cpp, 611, 612
chebyshev.h, 665, 666
chebyIP
chebyshev.cpp, 612
chebyshev.h, 667
chebyNorm
basisfunc.cpp, 514
basisfunc.h, 563
chebyshev.cpp, 613
chebyshev.h, 667
CHEBYNORM
chebyshev.h, 662
chebyNorm2
basisfunc.cpp, 515
basisfunc.h, 564
chebyshev.cpp, 613, 614
chebyshev.h, 667, 668
chebypoints
chebyshev.cpp, 614
chebyshev.h, 668
chebyProfile
flowfield.cpp, 896
flowfield.h, 944
chebyshev
chebyshev.cpp, 615, 616
chebyshev.h, 669, 670
chebyshev.cpp
chebyDist, 609
chebyDist2, 610
chebyInnerProduct, 611, 612
chebyIP, 612
chebyNorm, 613
chebyNorm2, 613, 614
chebypoints, 614
chebyshev, 615, 616
diff, 616, 617, 618, 619, 620, 621, 622
diff2, 625, 626, 627, 628, 629
dist, 631, 632
dist2, 632, 633

fftw_loadwisdom, 634
fftw_savewisdom, 634
gaussLegendreQuadrature, 634
innerProduct, 635, 636
integrate, 636, 637, 638, 639
L1Dist, 640, 641
L1Norm, 642, 643
L2Dist, 644, 645
L2Dist2, 645, 646
L2InnerProduct, 646, 647
L2Norm, 648
L2Norm2, 649
legendre, 650
LinfDist, 651
LinfNorm, 652
norm, 653, 654
norm2, 655
operator-, 657
operator!=, 656
operator*, 656
operator+, 656, 657
operator<<, 657
operator==, 657
periodicpoints, 658
QUADRATURE_EPSILON, 659
swap, 658
chebyshev.h
cheby_c, 662
chebyDist, 663
chebyDist2, 664
chebyEval, 665
chebyInnerProduct, 665, 666
chebyIP, 667
chebyNorm, 667
CHEBYNORM, 662
chebyNorm2, 667, 668
chebypoints, 668
chebyshev, 669, 670
diff, 670, 671, 672, 673, 674, 675, 676
diff2, 679, 680, 681, 682, 683
dist, 685, 686
dist2, 687
fftw_loadwisdom, 687
fftw_savewisdom, 687
gaussLegendreQuadrature, 688
Im, 688
innerProduct, 689, 690
integrate, 690, 691, 692, 693
L1Dist, 694, 695
L1Norm, 696, 697
L2Dist, 698, 699
L2Dist2, 699, 700
L2InnerProduct, 700, 701
L2Norm, 702
L2NORM, 662
L2Norm2, 703
legendre, 704
LinfDist, 705
LinfNorm, 706
norm, 707, 708
norm2, 709
NormType, 662
operator-, 711
operator!=, 710
operator*, 710
operator+, 710, 711
operator<<, 711
operator==, 711
periodicpoints, 712
Re, 712
ChebyTransform, 105
~ChebyTransform, 106
ChebyCoeff, 104, 107
ChebyTransform, 105
ComplexChebyCoeff, 107
cosfftw_plan_, 107
flags_, 107
length, 106
N, 106
N_, 107
operator=, 106
chebyUprofile
flowfield.cpp, 900
flowfield.h, 948
chebyVprofile
flowfield.cpp, 901
flowfield.h, 949
clone
CNABstyleDNS, 125
DNSAlgorithm, 177
MultistepDNS, 352
PPEDNS, 386

RungeKuttaDNS, 413
cmndline, 108
cmndline, 109
GetInstance, 110
getRun, 110
m_args, 112
m_nmbThreads, 113
m_run, 113
nThreads, 111
parse, 111
pInstance, 113
show, 112
cmplx
FlowField, 228, 229
CNAB2
dns.h, 885
CNABstyleDNS, 114
~CNABstyleDNS, 120
advance, 121
alpha_, 128
beta_, 128
clone, 125
CNABstyleDNS, 118, 119
fj_, 128
fj1_, 128
full, 125
full_, 128
gamma_, 129
Nsubsteps_, 129
operator=, 125
push, 125
reset_dt, 127
tausolver_, 129
zeta_, 129
CNFE1
dns.h, 885
CNRK2
dns.h, 885
coeff
FlowField, 232
Complex
mathdefs.h, 1003
complex_flatten
FlowField, 233
ComplexChebyCoeff, 130
a, 134
add, 135
b, 136
binaryDump, 136
binaryLoad, 137
chebyfft, 137
ChebyTransform, 107
ComplexChebyCoeff, 133, 134
congruent, 138
conjugate, 138
eval, 138
eval_a, 138
eval_b, 139
fill, 140
ichebyfft, 140
im, 153
interpolate, 141
L, 141
length, 141
makePhysical, 142
makeSpectral, 142, 143
makeState, 143
mean, 144
N, 144
numModes, 145
operator*=, 145, 146
operator[], 147
operator+=, 146
operator-=, 147
randomize, 147
re, 153
reflect, 147
resize, 148
save, 148
set, 149
setBounds, 150
setState, 151
setToZero, 151
state, 152
sub, 153
swap, 153
complexdefs.h
exp, 715
log, 716
component
BasisFunc, 44
congruent

BasisFunc, 44
ChebyCoeff, 80
ComplexChebyCoeff, 138
FlowField, 235
PoissonSolver, 362
conjugate
BasisFunc, 45
ComplexChebyCoeff, 138
constraint
DNSFlags, 192
Convection
diffops.h, 799
convectionNL
diffops.cpp, 718
diffops.h, 800
cosfftw_plan_
ChebyTransform, 107
count_
TurbStats, 485
countdown_
MultistepDNS, 356
PPEDNS, 390
CPU
dns.h, 884
CreateThreads
ThreadPool, 461
cross
basisfunc.cpp, 516
basisfunc.h, 565
diffops.cpp, 721, 722
diffops.h, 803, 804
cube
mathdefs.h, 1006
curl
basisfunc.cpp, 517
basisfunc.h, 566
diffops.cpp, 724
diffops.h, 805, 808
d_
BandedTridiag, 31
data_
array, 14
Vector, 500
dealias_xz
DNSFlags, 191
dealias_y
DNSFlags, 192
dealiased_
FlowField, 319
dealiasing
DNSFlags, 192
Dealiasing
dns.h, 884
DealiasXYZ
dns.h, 884
DealiasXZ
dns.h, 884
DealiasY
dns.h, 884
DestroyThreadPool
ThreadPool, 461
diag
BandedTridiag, 20
diff
chebyshev.cpp, 616, 617, 618, 619, 620, 621, 622
chebyshev.h, 670, 671, 672, 673, 674, 675, 676
diffops.cpp, 727
diffops.h, 808, 811
diff2
chebyshev.cpp, 625, 626, 627, 628, 629
chebyshev.h, 679, 680, 681, 682, 683
diffops.cpp
convectionNL, 718
cross, 721, 722
curl, 724
diff, 727
div, 731, 733
divergenceNL, 736
dot, 739, 740
energy, 741, 742, 743, 745
grad, 747, 749
lapl, 752, 753
linearizedNL, 756
navierstokesNL, 759
norm, 762, 763
norm2, 764, 765
operator<<, 766
outer, 767
rotationalNL, 770
skewsymmetricNL, 774

skewsymmetricNL_GPU, 778
skewsymmetricNL_THREAD, 782
up2q, 786
uq2p, 787
xdiff, 788
ydiff, 791
zdiff, 794
diffops.h
Alternating, 799
Alternating_, 799
Convection, 799
convectionNL, 800
cross, 803, 804
curl, 805, 808
diff, 808, 811
div, 811, 815
Divergence, 799
divergenceNL, 817
dot, 819, 820
energy, 821, 823, 825, 826
grad, 827, 831
i3j, 833
lapl, 833, 836
Linearized, 799
linearizedNL, 837
matlabSave, 840
navierstokesNL, 840
NonlinearMethod, 799
norm, 843, 845
norm2, 845, 846
operator<<, 847
outer, 847, 849
Rotational, 799
rotationalNL, 850
SkewSymmetric, 799
skewsymmetricNL, 854
skewsymmetricNL_GPU, 858
skewsymmetricNL_THREAD, 862
up2q, 866
uq2p, 867
xdiff, 868, 871
ydiff, 871, 874
zdiff, 874, 877
dist
basisfunc.cpp, 519
basisfunc.h, 569
chebyshev.cpp, 631, 632
chebyshev.h, 685, 686
dist2
basisfunc.cpp, 520
basisfunc.h, 570
chebyshev.cpp, 632, 633
chebyshev.h, 687
div
diffops.cpp, 731, 733
diffops.h, 811, 815
div_nonl_
PressureSolver, 401
divcheck
tausolver.cpp, 1026
tausolver.h, 1030
divDist
basisfunc.cpp, 520
basisfunc.h, 570
flowfield.cpp, 903
flowfield.h, 951
divDist2
basisfunc.cpp, 522
basisfunc.h, 572
flowfield.cpp, 906
flowfield.h, 954
divergence
basisfunc.cpp, 524
basisfunc.h, 574
Divergence
diffops.h, 799
divergenceNL
diffops.cpp, 736
diffops.h, 817
divNorm
basisfunc.cpp, 526
basisfunc.h, 576
flowfield.cpp, 908
flowfield.h, 956
divNorm2
basisfunc.cpp, 528
basisfunc.h, 578
flowfield.cpp, 910
flowfield.h, 958
DNS, 155
~DNS, 158
advance, 158

CFL, 159
DNS, 157, 158
dPdx, 159
dPdxRef, 160
dt, 160
flags, 160
full, 160
init_algorithm_, 166
main_algorithm_, 166
newAlgorithm, 161
Ninitsteps, 161
operator=, 162
order, 162
push, 162
reset_dPdx, 163
reset_dt, 163
reset_time, 164
reset_Ubulk, 165
time, 165
timestepping, 165
Ubulk, 165
UbulkRef, 166
dns.cpp
changeBaseFlow, 878
EPSILON, 882
operator<<, 880, 881
start_thread, 882
dns.h
BulkVelocity, 885
changeBaseFlow, 885
CNAB2, 885
CNFE1, 885
CNRK2, 885
CPU, 884
Dealiasing, 884
DealiasXYZ, 884
DealiasXZ, 884
DealiasY, 884
GPU, 884
HowToRun, 884
MeanConstraint, 885
NoDealiasing, 884
operator<<, 887, 888
PPEQ2, 885
PressureGradient, 885
PrintAll, 885
PrintTicks, 885
PrintTime, 885
SBDF1, 885
SBDF2, 885
SBDF3, 885
SBDF4, 885
Silent, 885
SMRK2, 885
THREAD, 884
TimeStepMethod, 885
Verbosity, 885
VerifyTauSolve, 885
DNSAlgorithm, 167
~DNSAlgorithm, 175
a, 176
a_, 184
advance, 176
b, 176
b_, 185
CFL, 177
cfl_, 185
clone, 177
DNSAlgorithm, 172, 173
dPdx, 177
dPdxAct_, 185
dPdxRef, 177
dPdxRef_, 185
dt, 177
dt_, 185
flags, 178
flags_, 185
full, 178
isAliasedMode, 178
kxd_max_, 185
kxmaxDealiased, 179
kzd_max_, 185
kzmaxDealiased, 179
Lx, 179
Lx_, 186
Lz, 180
Lz_, 186
Mx, 180
Mx_, 186
My, 180
Mz, 180
Mz_, 186

Ninitsteps, 180
Ninitsteps_, 186
nu, 180
nu_, 186
Nx, 181
Nx_, 186
Ny, 181
Ny_, 186
Nyd_, 187
Nz, 181
Nz_, 187
operator=, 181
order, 181
order_, 187
Pk_, 187
push, 182
Pyk_, 187
reset_dPdx, 182
reset_dt, 182
reset_time, 183
reset_Ubulk, 183
Rxk_, 187
Ryk_, 187
Rzk_, 187
t_, 187
time, 183
timestepping, 183
tmp_, 188
Ubase, 184
Ubase_, 188
Ubaseyy_, 188
Ubulk, 184
UbulkAct_, 188
UbulkBase_, 188
UbulkRef, 184
UbulkRef_, 188
uk_, 188
vk_, 188
wk_, 189
DNSFlags, 190
constraint, 192
dealias_xz, 191
dealias_y, 192
dealiasing, 192
DNSFlags, 190
initstepping, 192
nonlinearity, 192
taucorrection, 192
timestepping, 192
verbosity, 193
dot
basisfunc.cpp, 530
basisfunc.h, 580
diffops.cpp, 739, 740
diffops.h, 819, 820
dotdivide
Vector, 493
vector.cpp, 1058
vector.h, 1067
dotgrad
basisfunc.cpp, 531, 533
basisfunc.h, 581, 583
dottimes
Vector, 493
vector.cpp, 1058
vector.h, 1068
dPdx
DNS, 159
DNSAlgorithm, 177
dPdxAct_
DNSAlgorithm, 185
dPdxRef
DNS, 160
DNSAlgorithm, 177
dPdxRef_
DNSAlgorithm, 185
drand48
mathdefs.cpp, 989
dt
DNS, 160
DNSAlgorithm, 177
TimeStep, 470
dT
TimeStep, 470
dt_
DNSAlgorithm, 185
dT_
TimeStep, 471
dUdy
TurbStats, 478
dudy_a
FlowField, 237

dudy_b
  FlowField, 237
dump
  FlowField, 238
Dx
  FlowField, 238, 239
Dz
  FlowField, 239, 240
elem
  BandedTridiag, 21, 22
emptiness_
  FieldSeries, 198
energy
  diffops.cpp, 741, 742, 743, 745
  diffops.h, 821, 823, 825, 826
  FlowField, 241
EPSILON
  bandedtridiag.cpp, 504
  dns.cpp, 882
  flowfield.cpp, 936
  helmholtz.cpp, 986
  tausolver.cpp, 1028
eta_
  MultistepDNS, 356
  PPEDNS, 390
eval
  ChebyCoeff, 80, 81
  ComplexChebyCoeff, 138
eval_a
  ChebyCoeff, 82
  ComplexChebyCoeff, 138
eval_b
  ChebyCoeff, 83
  ComplexChebyCoeff, 139
Even
  mathdefs.h, 1003
executeTask
  ThreadPool, 462
exp
  complexdefs.h, 715
f
  skewsymmetricNL_task, 419
  skewsymmetricNL_task3, 428
  skewsymmetricNL_task4, 433
f_
  FieldSeries, 199
  MultistepDNS, 356
  PPEDNS, 390
fftw_initialize
  FlowField, 242
fftw_loadwisdom
  chebyshev.cpp, 634
  chebyshev.h, 687
fftw_savewisdom
  chebyshev.cpp, 634
  chebyshev.h, 687
FieldSeries, 194
  emptiness_, 198
  f_, 199
  FieldSeries, 196
  full, 196
  interpolate, 197
  push, 198
  t_, 199
fieldstate
  mathdefs.h, 1003
fill
  array, 11
  BasisFunc, 45
  ChebyCoeff, 84
  ComplexChebyCoeff, 140
fj_
  CNABstyleDNS, 128
fj1_
  CNABstyleDNS, 128
flags
  DNS, 160
  DNSAlgorithm, 178
flags_
  ChebyTransform, 107
  DNSAlgorithm, 185
flatten
  FlowField, 244
FlowField, 200
  ~FlowField, 211
  a, 211
  a_, 318
  addPerturbation, 213
  addPerturbations, 215, 216
  addProfile, 217, 218, 219
  asciiSave, 220
  assertState, 221

b, 222
b_, 318
binarySave, 224
call_fftw, 225
cdata_, 319
CFLfactor, 225, 226
chebyfft_y, 227
cmplx, 228, 229
coeff, 232
complex_flatten, 233
congruent, 235
dealiased_, 319
dudy_a, 237
dudy_b, 237
dump, 238
Dx, 238, 239
Dz, 239, 240
energy, 241
fftw_initialize, 242
flatten, 244
FlowField, 205, 206, 207, 209
geomCongruent, 245
ichebyfft_y, 246
interpolate, 248
irealfft_xz, 250
isAliased, 251
isNull, 252
kx, 252
kxmax, 254
kxmaxDealiased, 255
kxmin, 255
kz, 255
kzmax, 257
kzmaxDealiased, 257
kzmin, 258
Lx, 258
Lx_, 319
Ly, 260
Lz, 260
Lz_, 319
makePhysical, 262
makePhysical_xz, 262
makePhysical_y, 263
makeSpectral, 264
makeSpectral_xz, 266
makeSpectral_y, 267
makeState, 268
mx, 270
Mx, 271
My, 272
mz, 273
Mz, 274
Nd, 276
Nd_, 319
numXgridpts, 278
numXmodes, 278
numYgridpts, 278
numYmodes, 278
numZgridpts, 279
numZmodes, 279
Nx, 279
Nx_, 319
Ny, 281
Ny_, 319
Nz, 283
Nz_, 320
Nzpad_, 320
Nzpad2_, 320
operator(), 285
operator*=, 286
operator+=, 286, 287, 288
operator=, 290
operator-=, 288, 289
optimizeFFTW, 290
perturb, 291
print, 291, 292
profile, 293
rdata_, 320
realfft_xz, 295
rescale, 296
resize, 296
saveDivSpectrum, 298
saveProfile, 301
saveSlice, 302
saveSpectrum, 303, 304
scratch_, 320
setDealiased, 306
setState, 307
setToZero, 308
swap, 318
translate, 310
vectorDim, 311

x, 312
xgridpts, 312
xz_iplan_, 320
xz_plan_, 320
xzstate, 313
xzstate_, 320
y, 315
y_plan_, 320
ygridpts, 315
ystate, 316
ystate_, 321
z, 318
zgridpts, 318
flowfield.cpp
assignOrrSommField, 891
bcDist, 893
bcDist2, 893
bcNorm, 894
bcNorm2, 895
chebyProfile, 896
chebyUprofile, 900
chebyVprofile, 901
divDist, 903
divDist2, 906
divNorm, 908
divNorm2, 910
EPSILON, 936
L1Dist, 912
L1Norm, 913
L2Dist, 915
L2Dist2, 916, 918
L2InnerProduct, 919, 921
L2Norm, 923, 924
L2Norm2, 924, 926
LinfDist, 927
LinfNorm, 928
randomProfile, 929
randomUprofile, 933
randomVprofile, 934
swap, 935
flowfield.h
assignOrrSommField, 938
bcDist, 940
bcDist2, 941
bcNorm, 942
bcNorm2, 943
chebyProfile, 944
chebyUprofile, 948
chebyVprofile, 949
divDist, 951
divDist2, 954
divNorm, 956
divNorm2, 958
L1Dist, 960
L1Norm, 961
L2Dist, 963
L2Dist2, 964, 966
L2InnerProduct, 967, 969
L2Norm, 971, 972
L2Norm2, 972, 974
LinfDist, 975
LinfNorm, 976
randomProfile, 977
randomUprofile, 981
randomVprofile, 982
swap, 983
zero_last_mode, 984
full
CNABstyleDNS, 125
DNS, 160
DNSAlgorithm, 178
FieldSeries, 196
MultistepDNS, 352
PPEDNS, 386
full_
CNABstyleDNS, 128
gamma_
CNABstyleDNS, 129
gaussLegendreQuadrature
chebyshev.cpp, 634
chebyshev.h, 688
geomCongruent
FlowField, 245
PoissonSolver, 363
geometryCongruent
BasisFunc, 45
GetInstance
cmndline, 110
ThreadPool, 463
getRun
cmndline, 110
GPU

dns.h, 884
grad
diffops.cpp, 747, 749
diffops.h, 827, 831
grad_u
skewsymmetricNL_task, 419
gradient
basisfunc.cpp, 536, 538
basisfunc.h, 586, 588
Greater
mathdefs.h, 1006
helmholtz.cpp
EPSILON, 986
helmholtz_
PoissonSolver, 372
PPEDNS, 390
HelmholtzSolver, 322
a_, 334
Ae_, 335
Ao_, 335
b_, 335
Be_, 335
beta, 326
Bo_, 335
c, 326
HelmholtzSolver, 324, 325
lambda, 327
lambda_, 335
N_, 335
nEvenModes_, 335
nModes_, 335
nOddModes_, 335
nu_, 336
residual, 327, 328
solve, 329, 331
verify, 332
HowToRun
dns.h, 884
I
mathdefs.h, 1007
i00_
TauSolver, 455
i01_
TauSolver, 455
i10_
TauSolver, 455
i11_
TauSolver, 455
i2s
mathdefs.cpp, 991
mathdefs.h, 1007
i3j
diffops.h, 833
ichebyfft
BasisFunc, 46
ChebyCoeff, 85
ComplexChebyCoeff, 140
ichebyfft_y
FlowField, 246
im
ComplexChebyCoeff, 153
Im
chebyshev.h, 688
mathdefs.h, 1008
influenceCorrection
TauSolver, 441
init_algorithm_
DNS, 166
initstepping
DNSFlags, 192
innerProduct
basisfunc.cpp, 540
basisfunc.h, 590
chebyshev.cpp, 635, 636
chebyshev.h, 689, 690
integrate
chebyshev.cpp, 636, 637, 638, 639
chebyshev.h, 690, 691, 692, 693
interpolate
BasisFunc, 47
ChebyCoeff, 86
ComplexChebyCoeff, 141
FieldSeries, 197
FlowField, 248
irealfft_xz
FlowField, 250
isAliased
FlowField, 251
isAliasedMode
DNSAlgorithm, 178
isNull
FlowField, 252

isPowerOfTwo
  mathdefs.cpp, 991
  mathdefs.h, 1008
kappa2_
  TauSolver, 456
kprime_func
  tausolver.cpp, 1027
kronecker
  mathdefs.h, 1008
kx
  BasisFunc, 47
  FlowField, 252
  TauSolver, 442
kx_
  BasisFunc, 65
  TauSolver, 456
kxd_max_
  DNSAlgorithm, 185
kxmax
  Attributes, 16
  FlowField, 254
  skewsymmetricNL_task3, 428
  skewsymmetricNL_task4, 434
kxmaxDealiased
  DNSAlgorithm, 179
  FlowField, 255
kxmin
  FlowField, 255
kz
  BasisFunc, 48
  FlowField, 255
  TauSolver, 442
kz_
  BasisFunc, 65
  TauSolver, 456
kzd_max_
  DNSAlgorithm, 185
kzmax
  Attributes, 16
  FlowField, 257
  skewsymmetricNL_task3, 428
  skewsymmetricNL_task4, 434
kzmaxDealiased
  DNSAlgorithm, 179
  FlowField, 257
kzmin
  FlowField, 258
L
  ChebyCoeff, 86
  ComplexChebyCoeff, 141
L1Dist
  chebyshev.cpp, 640, 641
  chebyshev.h, 694, 695
  flowfield.cpp, 912
  flowfield.h, 960
  vector.cpp, 1058
  vector.h, 1068
L1Norm
  chebyshev.cpp, 642, 643
  chebyshev.h, 696, 697
  flowfield.cpp, 913
  flowfield.h, 961
  vector.cpp, 1059
  vector.h, 1068
L2Dist
  basisfunc.cpp, 540
  basisfunc.h, 590
  chebyshev.cpp, 644, 645
  chebyshev.h, 698, 699
  flowfield.cpp, 915
  flowfield.h, 963
  vector.cpp, 1059
  vector.h, 1069
L2Dist2
  basisfunc.cpp, 541
  basisfunc.h, 591
  chebyshev.cpp, 645, 646
  chebyshev.h, 699, 700
  flowfield.cpp, 916, 918
  flowfield.h, 964, 966
  vector.cpp, 1059
  vector.h, 1069
L2InnerProduct
  basisfunc.cpp, 542
  basisfunc.h, 592
  chebyshev.cpp, 646, 647
  chebyshev.h, 700, 701
  flowfield.cpp, 919, 921
  flowfield.h, 967, 969
L2Norm
  basisfunc.cpp, 543
  basisfunc.h, 593

chebyshev.cpp, 648
chebyshev.h, 702
flowfield.cpp, 923, 924
flowfield.h, 971, 972
vector.cpp, 1060
vector.h, 1069
L2NORM
chebyshev.h, 662
L2Norm2
basisfunc.cpp, 544
basisfunc.h, 594
chebyshev.cpp, 649
chebyshev.h, 703
flowfield.cpp, 924, 926
flowfield.h, 972, 974
vector.cpp, 1060
vector.h, 1070
lambda
BasisFunc, 48
HelmholtzSolver, 327
TauSolver, 442
lambda_
BasisFunc, 65
HelmholtzSolver, 335
TauSolver, 456
lapl
diffops.cpp, 752, 753
diffops.h, 833, 836
laplacian
basisfunc.cpp, 545
basisfunc.h, 595
legendre
chebyshev.cpp, 650
chebyshev.h, 704
length
array, 11
ChebyTransform, 106
ComplexChebyCoeff, 141
Vector, 493
lesser
mathdefs.h, 1008
Limits, 337
Limits, 337
m_maxI, 338
m_maxJ, 338
m_maxNx, 338
m_maxNy, 338
m_maxNz, 338
operator=, 337
setLimits, 338
Linearized
diffops.h, 799
linearizedNL
diffops.cpp, 756
diffops.h, 837
LinfDist
chebyshev.cpp, 651
chebyshev.h, 705
flowfield.cpp, 927
flowfield.h, 975
vector.cpp, 1060
vector.h, 1070
LinfNorm
chebyshev.cpp, 652
chebyshev.h, 706
flowfield.cpp, 928
flowfield.h, 976
vector.cpp, 1061
vector.h, 1071
load
mathdefs.cpp, 991, 992
mathdefs.h, 1011
lodiag
BandedTridiag, 23
log
complexdefs.h, 716
Lx
Attributes, 16
BasisFunc, 48
DNSAlgorithm, 179
FlowField, 258
skewsymmetricNL_task3, 429
skewsymmetricNL_task4, 434
Lx_
BasisFunc, 66
DNSAlgorithm, 186
FlowField, 319
PoissonSolver, 372
Ly
FlowField, 260
Lz
Attributes, 16

BasisFunc, 50
DNSAlgorithm, 180
FlowField, 260
skewsymmetricNL_task3, 429
skewsymmetricNL_task4, 434
Lz_
BasisFunc, 66
DNSAlgorithm, 186
FlowField, 319
PoissonSolver, 372
M_
BandedTridiag, 31
m_args
cmndline, 112
m_attr
skewsymmetricNL_task, 420
m_cond
ThreadPool, 466
m_cond_done
ThreadPool, 466
m_lim
ThreadPool, 466
m_map
ThreadPool, 467
m_maxI
Limits, 338
m_maxJ
Limits, 338
m_maxNx
Limits, 338
m_maxNy
Limits, 338
m_maxNz
Limits, 338
m_mutex
ThreadPool, 467
m_mutex_done
ThreadPool, 467
m_nmbThreads
cmndline, 113
ThreadPool, 467
M_PI
mathdefs.h, 1002
m_run
cmndline, 113
m_step
skewsymmetricNL_task, 420
m_stop_threads
ThreadPool, 467
m_task
ThreadPool, 467
m_total_running_threads
ThreadPool, 467
m_work
ThreadPool, 467
main_algorithm_
DNS, 166
makemeen
utilfuncs.cpp, 1035
utilfuncs.h, 1046
makePhysical
BasisFunc, 51, 52
ChebyCoeff, 88
ComplexChebyCoeff, 142
FlowField, 262
makePhysical_xz
FlowField, 262
makePhysical_y
FlowField, 263
makeprtb
utilfuncs.cpp, 1036
utilfuncs.h, 1047
makeroll
utilfuncs.cpp, 1037
utilfuncs.h, 1048
makeSpectral
BasisFunc, 52, 53
ChebyCoeff, 88, 89
ComplexChebyCoeff, 142, 143
FlowField, 264
makeSpectral_xz
FlowField, 266
makeSpectral_y
FlowField, 267
makeState
BasisFunc, 53
ChebyCoeff, 90
ComplexChebyCoeff, 143
FlowField, 268
makewave
utilfuncs.cpp, 1039
utilfuncs.h, 1050

makewobl
  utilfuncs.cpp, 1040
  utilfuncs.h, 1051
mathdefs.cpp
  cferror, 989
  cfpause, 990
  channelflowVersion, 991
  drand48, 989
  i2s, 991
  isPowerOfTwo, 991
  load, 991, 992
  operator<<, 992
  operator>>, 993
  pow, 993
  pythag, 994
  randomComplex, 995
  randomReal, 995
  read, 996, 997
  roundReal, 997
  save, 998
  write, 998, 999, 1000
mathdefs.h
  abs2, 1003
  cferror, 1004
  cfpause, 1005
  channelflowVersion, 1006
  Complex, 1003
  cube, 1006
  Even, 1003
  fieldstate, 1003
  Greater, 1006
  I, 1007
  i2s, 1007
  Im, 1008
  isPowerOfTwo, 1008
  kronecker, 1008
  lesser, 1008
  load, 1011
  M_PI, 1002
  nr_sign, 1011
  Odd, 1003
  operator<<, 1011, 1012
  operator>>, 1012
  parity, 1003
  Physical, 1003
  pi, 1022
  pow, 1012
  pythag, 1013
  randomComplex, 1014
  randomReal, 1014
  Re, 1015
  read, 1015, 1016
  readLittleEndian, 1017
  Real, 1003
  REAL_DIGITS, 1022
  REAL_IOWIDTH, 1023
  roundReal, 1017
  save, 1017, 1018
  Spectral, 1003
  square, 1018
  swap, 1020
  write, 1020, 1021, 1022
matlabSave
  diffops.h, 840
maxElemIndex
  vector.cpp, 1061
  vector.h, 1071
Mbar_
  BandedTridiag, 32
mean
  ChebyCoeff, 91
  ComplexChebyCoeff, 144
  vector.cpp, 1062
  vector.h, 1071
MeanConstraint
  dns.h, 885
MINIMUM_DISCRIMINANT
  tausolver.cpp, 1028
modularSubvector
  Vector, 495
msave
  TurbStats, 478
multiply
  BandedTridiag, 23
multiplyStrided
  BandedTridiag, 24
MultistepDNS, 340
  ~MultistepDNS, 348
  advance, 348
  alpha_, 356
  beta_, 356
  clone, 352

countdown_, 356
eta_, 356
f_, 356
full, 352
MultistepDNS, 344, 345
operator=, 352
push, 352
reset_dt, 355
tausolver_, 357
u_, 357
mut
skewsymmetricNL_task, 420
mx
FlowField, 270
Mx
DNSAlgorithm, 180
FlowField, 271
skewsymmetricNL_task3, 429
skewsymmetricNL_task4, 434
Mx_
DNSAlgorithm, 186
PoissonSolver, 372
My
DNSAlgorithm, 180
FlowField, 272
skewsymmetricNL_task3, 429
skewsymmetricNL_task4, 434
My_
PoissonSolver, 372
mz
FlowField, 273
Mz
DNSAlgorithm, 180
FlowField, 274
skewsymmetricNL_task3, 429
skewsymmetricNL_task4, 434
Mz_
DNSAlgorithm, 186
PoissonSolver, 372
n
TimeStep, 470
N
array, 11
ChebyCoeff, 92
ChebyTransform, 106
ComplexChebyCoeff, 144
Vector, 495
n_
TimeStep, 471
N_
array, 14
ChebyTransform, 107
HelmholtzSolver, 335
TauSolver, 456
Vector, 501
n_func
tausolver.cpp, 1028
navierstokesNL
diffops.cpp, 759
diffops.h, 840
Nb_
TauSolver, 456
Nd
BasisFunc, 54
FlowField, 276
Nd_
BasisFunc, 66
FlowField, 319
PoissonSolver, 372
nEvenModes_
HelmholtzSolver, 335
newAlgorithm
DNS, 161
Ninitsteps
DNS, 161
DNSAlgorithm, 180
Ninitsteps_
DNSAlgorithm, 186
nmax_
TimeStep, 471
nmin_
TimeStep, 471
nModes_
HelmholtzSolver, 335
nOddModes_
HelmholtzSolver, 335
NoDealiasing
dns.h, 884
nonl_
PressureSolver, 401
nonl_method_
PressureSolver, 401

nonlinearity
DNSFlags, 192
NonlinearMethod
diffops.h, 799
norm
basisfunc.cpp, 547
basisfunc.h, 598
chebyshev.cpp, 653, 654
chebyshev.h, 707, 708
diffops.cpp, 762, 763
diffops.h, 843, 845
norm2
basisfunc.cpp, 548
basisfunc.h, 599
chebyshev.cpp, 655
chebyshev.h, 709
diffops.cpp, 764, 765
diffops.h, 845, 846
NormType
chebyshev.h, 662
nr_sign
mathdefs.h, 1011
Nsubsteps_
CNABstyleDNS, 129
RungeKuttaDNS, 414
nThreads
cmndline, 111
nu
DNSAlgorithm, 180
TauSolver, 443
nu_
DNSAlgorithm, 186
HelmholtzSolver, 336
PressureSolver, 401
TauSolver, 456
TurbStats, 485
numModes
ChebyCoeff, 93
ComplexChebyCoeff, 145
numNonzeros
BandedTridiag, 25
numrows
BandedTridiag, 25
numRows
BandedTridiag, 25
numXgridpts
FlowField, 278
numXmodes
FlowField, 278
numYgridpts
FlowField, 278
numYmodes
FlowField, 278
numZgridpts
FlowField, 279
numZmodes
FlowField, 279
Nx
DNSAlgorithm, 181
FlowField, 279
Nx_
DNSAlgorithm, 186
FlowField, 319
Ny
BasisFunc, 55
DNSAlgorithm, 181
FlowField, 281
Ny_
BasisFunc, 66
DNSAlgorithm, 186
FlowField, 319
Nyd_
DNSAlgorithm, 187
Nz
DNSAlgorithm, 181
FlowField, 283
Nz_
DNSAlgorithm, 187
FlowField, 320
Nzpad_
FlowField, 320
Nzpad2_
FlowField, 320
Odd
mathdefs.h, 1003
operator-
chebyshev.cpp, 657
chebyshev.h, 711
vector.cpp, 1063
vector.h, 1072
operator Real
TimeStep, 470

operator!=
array, 11
basisfunc.cpp, 548
basisfunc.h, 599
chebyshev.cpp, 656
chebyshev.h, 710
operator()
FlowField, 285
Vector, 495
operator*
chebyshev.cpp, 656
chebyshev.h, 710
vector.cpp, 1062
vector.h, 1072
operator*=
BasisFunc, 55
ChebyCoeff, 94, 95
ComplexChebyCoeff, 145, 146
FlowField, 286
Vector, 495
operator/=
Vector, 496
operator[]
array, 12
BasisFunc, 56
ComplexChebyCoeff, 147
Vector, 497
operator+
chebyshev.cpp, 656, 657
chebyshev.h, 710, 711
vector.cpp, 1062
vector.h, 1072
operator+=
BasisFunc, 56
ChebyCoeff, 95
ComplexChebyCoeff, 146
FlowField, 286, 287, 288
Vector, 496
operator<<
array.h, 503
chebyshev.cpp, 657
chebyshev.h, 711
diffops.cpp, 766
diffops.h, 847
dns.cpp, 880, 881
dns.h, 887, 888
mathdefs.cpp, 992
mathdefs.h, 1011, 1012
vector.cpp, 1063
vector.h, 1073
operator=
array, 12
Attributes, 15
BandedTridiag, 25
ChebyTransform, 106
CNABstyleDNS, 125
DNS, 162
DNSAlgorithm, 181
FlowField, 290
Limits, 337
MultistepDNS, 352
PoissonSolver, 364
PPEDNS, 386
RungeKuttaDNS, 413
Vector, 497
operator-=
BasisFunc, 56
operator-=
ChebyCoeff, 95
operator-=
ComplexChebyCoeff, 147
operator-=
FlowField, 288
operator-=
FlowField, 288
operator-=
FlowField, 289
operator-=
FlowField, 289
operator-=
Vector, 496
operator-=
Vector, 496
operator==
array, 12
BandedTridiag, 26
basisfunc.cpp, 548
basisfunc.h, 599
chebyshev.cpp, 657
chebyshev.h, 711
vector.cpp, 1063
vector.h, 1073

operator>>
  mathdefs.cpp, 993
  mathdefs.h, 1012
optimizeFFTW
  FlowField, 290
order
  DNS, 162
  DNSAlgorithm, 181
order_
  DNSAlgorithm, 187
outer
  diffops.cpp, 767
  diffops.h, 847, 849
P_0_
  TauSolver, 456
P_minus_
  TauSolver, 456
P_plus_
  TauSolver, 456
parabolicReynolds
  TurbStats, 479
parity
  mathdefs.h, 1003
parse
  cmndline, 111
periodicpoints
  chebyshev.cpp, 658
  chebyshev.h, 712
perturb
  FlowField, 291
Physical
  mathdefs.h, 1003
pi
  mathdefs.h, 1022
pInstance
  cmndline, 113
  ThreadPool, 467
Pk_
  DNSAlgorithm, 187
pointer
  array, 13
  Vector, 497
PoissonSolver, 358
  ~PoissonSolver, 362
  a_, 372
  b_, 372
  congruent, 362
  geomCongruent, 363
  helmholtz_, 372
  Lx_, 372
  Lz_, 372
  Mx_, 372
  My_, 372
  Mz_, 372
  Nd_, 372
  operator=, 364
  PoissonSolver, 360, 361, 362
  solve, 365, 366
  verify, 367, 370
polyInterp
  utilfuncs.cpp, 1042
  utilfuncs.h, 1053
pow
  mathdefs.cpp, 993
  mathdefs.h, 1012
PPEDNS, 374
  ~PPEDNS, 382
  advance, 382
  alpha_, 390
  beta_, 390
  clone, 386
  countdown_, 390
  eta_, 390
  f_, 390
  full, 386
  helmholtz_, 390
  operator=, 386
  PPEDNS, 378, 379
  psolve_, 390
  push, 386
  reset_dt, 389
  u_, 390
PPEQ2
  dns.h, 885
PressureGradient
  dns.h, 885
pressureHelmholtz_
  TauSolver, 456
PressureSolver, 391
  ~PressureSolver, 395
  div_nonl_, 401
  nonl_, 401

nonl_method_, 401
nu_, 401
PressureSolver, 393, 394
solve, 395
tmp_, 401
trans_, 401
U_, 401
verify, 398
print
BandedTridiag, 26
FlowField, 291, 292
PrintAll
dns.h, 885
PrintTicks
dns.h, 885
PrintTime
dns.h, 885
profile
FlowField, 293
psolve_
PPEDNS, 390
push
CNABstyleDNS, 125
DNS, 162
DNSAlgorithm, 182
FieldSeries, 198
MultistepDNS, 352
PPEDNS, 386
Pyk_
DNSAlgorithm, 187
pythag
mathdefs.cpp, 994
mathdefs.h, 1013
Qj_
RungeKuttaDNS, 415
Qj1_
RungeKuttaDNS, 414
QUADRATURE_EPSILON
chebyshev.cpp, 659
randomComplex
mathdefs.cpp, 995
mathdefs.h, 1014
randomize
BasisFunc, 56
ChebyCoeff, 96
ComplexChebyCoeff, 147
Vector, 498
randomProfile
flowfield.cpp, 929
flowfield.h, 977
randomReal
mathdefs.cpp, 995
mathdefs.h, 1014
randomUprofile
flowfield.cpp, 933
flowfield.h, 981
randomVprofile
flowfield.cpp, 934
flowfield.h, 982
rdata_
FlowField, 320
re
ComplexChebyCoeff, 153
Re
chebyshev.h, 712
mathdefs.h, 1015
read
mathdefs.cpp, 996, 997
mathdefs.h, 1015, 1016
readLittleEndian
mathdefs.h, 1017
Real
array.h, 503
bandedtridiag.h, 505
mathdefs.h, 1003
REAL_DIGITS
mathdefs.h, 1022
REAL_IOWIDTH
mathdefs.h, 1023
REAL_OUTPUT_DIGITS
array.h, 503
realfft_xz
FlowField, 295
reflect
BasisFunc, 58
ChebyCoeff, 96
ComplexChebyCoeff, 147
rescale
FlowField, 296
reset
TurbStats, 480
reset_dPdx

DNS, 163
DNSAlgorithm, 182
reset_dt
CNABstyleDNS, 127
DNS, 163
DNSAlgorithm, 182
MultistepDNS, 355
PPEDNS, 389
RungeKuttaDNS, 413
reset_time
DNS, 164
DNSAlgorithm, 183
reset_Ubulk
DNS, 165
DNSAlgorithm, 183
residual
HelmholtzSolver, 327, 328
resize
array, 13
BasisFunc, 58
ComplexChebyCoeff, 148
FlowField, 296
Vector, 498
rmsval
BasisFunc, 59
Rotational
diffops.h, 799
rotationalNL
diffops.cpp, 770
diffops.h, 850
roundReal
mathdefs.cpp, 997
mathdefs.h, 1017
Run
BaseTask, 33
skewsymmetricNL_task, 418
skewsymmetricNL_task2, 423
skewsymmetricNL_task3, 427
skewsymmetricNL_task4, 432
ThreadPool, 463
RungeKuttaDNS, 402
~RungeKuttaDNS, 408
A_, 414
advance, 408
B_, 414
C_, 414
clone, 413
Nsubsteps_, 414
operator=, 413
Qj_, 415
Qj1_, 414
reset_dt, 413
RungeKuttaDNS, 406, 407
tausolver_, 415
Rxk_
DNSAlgorithm, 187
Ryk_
DNSAlgorithm, 187
Rzk_
DNSAlgorithm, 187
save
array, 14
BandedTridiag, 26
BasisFunc, 59
ChebyCoeff, 97
ComplexChebyCoeff, 148
mathdefs.cpp, 998
mathdefs.h, 1017, 1018
Vector, 500
saveDivSpectrum
FlowField, 298
saveProfile
FlowField, 301
saveSlice
FlowField, 302
saveSpectrum
FlowField, 303, 304
SBDF1
dns.h, 885
SBDF2
dns.h, 885
SBDF3
dns.h, 885
SBDF4
dns.h, 885
scratch_
FlowField, 320
secantSearch
utilfuncs.cpp, 1043
utilfuncs.h, 1054
set
ComplexChebyCoeff, 149

setAttributes
  Attributes, 16
  skewsymmetricNL_task, 419
setBounds
  BasisFunc, 60
  ChebyCoeff, 98
  ComplexChebyCoeff, 150
setDealiased
  FlowField, 306
setkxkz
  BasisFunc, 60
setLimits
  Limits, 338
  ThreadPool, 464
setRmsval
  BasisFunc, 60
setState
  BasisFunc, 60
  ChebyCoeff, 100
  ComplexChebyCoeff, 151
  FlowField, 307
setStep
  skewsymmetricNL_task, 419
setTask
  ThreadPool, 464
setToZero
  BasisFunc, 61
  ChebyCoeff, 102
  ComplexChebyCoeff, 151
  FlowField, 308
  Vector, 500
show
  cmndline, 112
sigma0_Nb_
  TauSolver, 457
sigma0_Nb1_
  TauSolver, 457
Silent
  dns.h, 885
SkewSymmetric
  diffops.h, 799
skewsymmetricNL
  diffops.cpp, 774
  diffops.h, 854
skewsymmetricNL_GPU
  diffops.cpp, 778
  diffops.h, 858
skewsymmetricNL_task, 416
  f, 419
  grad_u, 419
  m_attr, 420
  m_step, 420
  mut, 420
  Run, 418
  setAttributes, 419
  setStep, 419
  skewsymmetricNL_task, 418
  u, 420
skewsymmetricNL_task2, 421
  Run, 423
  skewsymmetricNL_task2, 423
  tmp, 424
  u, 424
  uu, 424
skewsymmetricNL_task3, 425
  f, 428
  kxmax, 428
  kzmax, 428
  Lx, 429
  Lz, 429
  Mx, 429
  My, 429
  Mz, 429
  Run, 427
  skewsymmetricNL_task3, 427
  u, 429
  uu, 429
skewsymmetricNL_task4, 430
  f, 433
  kxmax, 434
  kzmax, 434
  Lx, 434
  Lz, 434
  Mx, 434
  My, 434
  Mz, 434
  Run, 432
  skewsymmetricNL_task4, 432
  tmpProfile, 434
  tmpProfile_y, 434
  u, 434
  uu, 434

skewsymmetricNL_THREAD
  diffops.cpp, 782
  diffops.h, 862
SMRK2
  dns.h, 885
solve
  HelmholtzSolver, 329, 331
  PoissonSolver, 365, 366
  PressureSolver, 395
  TauSolver, 443, 444
solve_P_and_v
  TauSolver, 446
Spectral
  mathdefs.h, 1003
square
  mathdefs.h, 1018
start_thread
  dns.cpp, 882
state
  BasisFunc, 61
  ChebyCoeff, 102
  ComplexChebyCoeff, 152
state_
  BasisFunc, 66
  ChebyCoeff, 104
sub
  ComplexChebyCoeff, 153
subvector
  array, 14
  Vector, 500
swap
  ChebyCoeff, 104
  chebyshev.cpp, 658
  ComplexChebyCoeff, 153
  FlowField, 318
  flowfield.cpp, 935
  flowfield.h, 983
  mathdefs.h, 1020
  Vector, 500
  vector.cpp, 1064
t_
  DNSAlgorithm, 187
  FieldSeries, 199
taucorrection
  DNSFlags, 192
tauCorrection_
  TauSolver, 457
tauDist
  TauSolver, 447, 448
tauNorm
  TauSolver, 448, 449
TauSolver, 436
  a_, 455
  b_, 455
  i00_, 455
  i01_, 455
  i10_, 455
  i11_, 455
  influenceCorrection, 441
  kappa2_, 456
  kx, 442
  kx_, 456
  kz, 442
  kz_, 456
  lambda, 442
  lambda_, 456
  N_, 456
  Nb_, 456
  nu, 443
  nu_, 456
  P_0_, 456
  P_minus_, 456
  P_plus_, 456
  pressureHelmholtz_, 456
  sigma0_Nb_, 457
  sigma0_Nb1_, 457
  solve, 443, 444
  solve_P_and_v, 446
  tauCorrection_, 457
  tauDist, 447, 448
  tauNorm, 448, 449
  TauSolver, 439
  two_pi_kxLx_, 457
  two_pi_kzLz_, 457
  v_0_, 457
  v_minus_, 457
  v_plus_, 457
  velocityHelmholtz_, 457
  verify, 449, 452
  verify_P_and_v, 453
tausolver.cpp
  divcheck, 1026

EPSILON, 1028
kprime_func, 1027
MINIMUM_DISCRIMINANT, 1028
n_func, 1028
tausolver.h
divcheck, 1030
tausolver_
CNABstyleDNS, 129
MultistepDNS, 357
RungeKuttaDNS, 415
test
BandedTridiag, 27
THREAD
dns.h, 884
ThreadPool, 458
CreateThreads, 461
DestroyThreadPool, 461
executeTask, 462
GetInstance, 463
m_cond, 466
m_cond_done, 466
m_lim, 466
m_map, 467
m_mutex, 467
m_mutex_done, 467
m_nmbThreads, 467
m_stop_threads, 467
m_task, 467
m_total_running_threads, 467
m_work, 467
pInstance, 467
Run, 463
setLimits, 464
setTask, 464
ThreadPool, 460
WaitToComplete, 465
WakeUp, 466
time
DNS, 165
DNSAlgorithm, 183
TimeStep, 468
adjust, 468
CFL, 469
CFL_, 470
CFLmax_, 470
CFLmin_, 470
dt, 470
dT, 470
dT_, 471
n, 470
n_, 471
nmax_, 471
nmin_, 471
operator Real, 470
TimeStep, 468
TimeStepMethod
dns.h, 885
timestepping
DNS, 165
DNSAlgorithm, 183
DNSFlags, 192
tmp
skewsymmetricNL_task2, 424
tmp_
DNSAlgorithm, 188
PressureSolver, 401
tmpProfile
skewsymmetricNL_task4, 434
tmpProfile_y
skewsymmetricNL_task4, 434
trans_
PressureSolver, 401
translate
FlowField, 310
TurbStats, 472
addData, 475
bulkReynolds, 477
centerlineReynolds, 477
count_, 485
dUdy, 478
msave, 478
nu_, 485
parabolicReynolds, 479
reset, 480
TurbStats, 474, 475
U, 480
U_, 485
ubase, 481
Ubase, 481
ubase_, 485
Ubase_, 485
ustar, 481

uu, 482
uu_, 485
uv, 483
uv_, 485
uw, 483
uw_, 485
vv, 483
vv_, 485
vw, 483
vw_, 485
ww, 484
ww_, 486
yplus, 484
two_pi_kxLx_
TauSolver, 457
two_pi_kzLz_
TauSolver, 457
u
BasisFunc, 62
skewsymmetricNL_task, 420
skewsymmetricNL_task2, 424
skewsymmetricNL_task3, 429
skewsymmetricNL_task4, 434
U
TurbStats, 480
u_
BasisFunc, 66
MultistepDNS, 357
PPEDNS, 390
U_
PressureSolver, 401
TurbStats, 485
ubase
TurbStats, 481
Ubase
DNSAlgorithm, 184
TurbStats, 481
ubase_
TurbStats, 485
Ubase_
DNSAlgorithm, 188
TurbStats, 485
Ubaseyy_
DNSAlgorithm, 188
ubcFix
basisfunc.cpp, 549
basisfunc.h, 600
Ubulk
DNS, 165
DNSAlgorithm, 184
UbulkAct_
DNSAlgorithm, 188
UbulkBase_
DNSAlgorithm, 188
UbulkRef
DNS, 166
DNSAlgorithm, 184
UbulkRef_
DNSAlgorithm, 188
uk_
DNSAlgorithm, 188
UL_
BandedTridiag, 32
ULdecomp
BandedTridiag, 27
ULprint
BandedTridiag, 28
ULsolve
BandedTridiag, 29
ULsolveStrided
BandedTridiag, 29
up2q
diffops.cpp, 786
diffops.h, 866
updiag
BandedTridiag, 30, 31
uq2p
diffops.cpp, 787
diffops.h, 867
ustar
TurbStats, 481
utilfuncs.cpp
bisectSearch, 1034
makemeen, 1035
makeprtb, 1036
makeroll, 1037
makewave, 1039
makewobl, 1040
polyInterp, 1042
secantSearch, 1043
utilfuncs.h
bisectSearch, 1046

makemeen, 1046
makeprtb, 1047
makeroll, 1048
makewave, 1050
makewobl, 1051
polyInterp, 1053
secantSearch, 1054
uu
skewsymmetricNL_task2, 424
skewsymmetricNL_task3, 429
skewsymmetricNL_task4, 434
TurbStats, 482
uu_
TurbStats, 485
uv
TurbStats, 483
uv_
TurbStats, 485
uw
TurbStats, 483
uw_
TurbStats, 485
v
BasisFunc, 63
v_0_
TauSolver, 457
v_minus_
TauSolver, 457
v_plus_
TauSolver, 457
vbcFix
basisfunc.cpp, 550, 551
basisfunc.h, 601, 602
Vector, 487
~Vector, 491
abs, 491
data_, 500
dotdivide, 493
dottimes, 493
length, 493
modularSubvector, 495
N, 495
N_, 501
operator(), 495
operator*=, 495
operator/=, 496
operator[], 497
operator+=, 496
operator=, 497
operator-=, 496
pointer, 497
randomize, 498
resize, 498
save, 500
setToZero, 500
subvector, 500
swap, 500
Vector, 489, 490
vector.cpp
abs, 1056
assign, 1058
dotdivide, 1058
dottimes, 1058
L1Dist, 1058
L1Norm, 1059
L2Dist, 1059
L2Dist2, 1059
L2Norm, 1060
L2Norm2, 1060
LinfDist, 1060
LinfNorm, 1061
maxElemIndex, 1061
mean, 1062
operator-, 1063
operator*, 1062
operator+, 1062
operator<<, 1063
operator==, 1063
swap, 1064
vector.h
abs, 1066
assign, 1067
dotdivide, 1067
dottimes, 1068
L1Dist, 1068
L1Norm, 1068
L2Dist, 1069
L2Dist2, 1069
L2Norm, 1069
L2Norm2, 1070
LinfDist, 1070
LinfNorm, 1071

maxElemIndex, 1071
mean, 1071
operator-, 1072
operator*, 1072
operator+, 1072
operator<<, 1073
operator==, 1073
vectorDim
FlowField, 311
velocityHelmholtz_
TauSolver, 457
verbosity
DNSFlags, 193
Verbosity
dns.h, 885
verify
HelmholtzSolver, 332
PoissonSolver, 367, 370
PressureSolver, 398
TauSolver, 449, 452
verify_P_and_v
TauSolver, 453
VerifyTauSolve
dns.h, 885
vk_
DNSAlgorithm, 188
vv
TurbStats, 483
vv_
TurbStats, 485
vw
TurbStats, 483
vw_
TurbStats, 485
w
BasisFunc, 64
WaitToComplete
ThreadPool, 465
WakeUp
ThreadPool, 466
wk_
DNSAlgorithm, 189
write
mathdefs.cpp, 998, 999, 1000
mathdefs.h, 1020, 1021, 1022
ww
TurbStats, 484
ww_
TurbStats, 486
x
FlowField, 312
xdiff
basisfunc.cpp, 552
basisfunc.h, 603
diffops.cpp, 788
diffops.h, 868, 871
xgridpts
FlowField, 312
xz_iplan_
FlowField, 320
xz_plan_
FlowField, 320
xzstate
FlowField, 313
xzstate_
FlowField, 320
y
FlowField, 315
y_plan_
FlowField, 320
ydiff
basisfunc.cpp, 553
basisfunc.h, 604
diffops.cpp, 791
diffops.h, 871, 874
ygridpts
FlowField, 315
yplus
TurbStats, 484
ystate
FlowField, 316
ystate_
FlowField, 321
z
FlowField, 318
zdiff
basisfunc.cpp, 555
basisfunc.h, 606
diffops.cpp, 794
diffops.h, 874, 877
zero_last_mode
flowfield.h, 984

zeta_
  CNABstyleDNS, 129
zgridpts
  FlowField, 318