Channelflow Users' Manual
Release 0.9.18

John F. Gibson

February 13, 2009

Contents

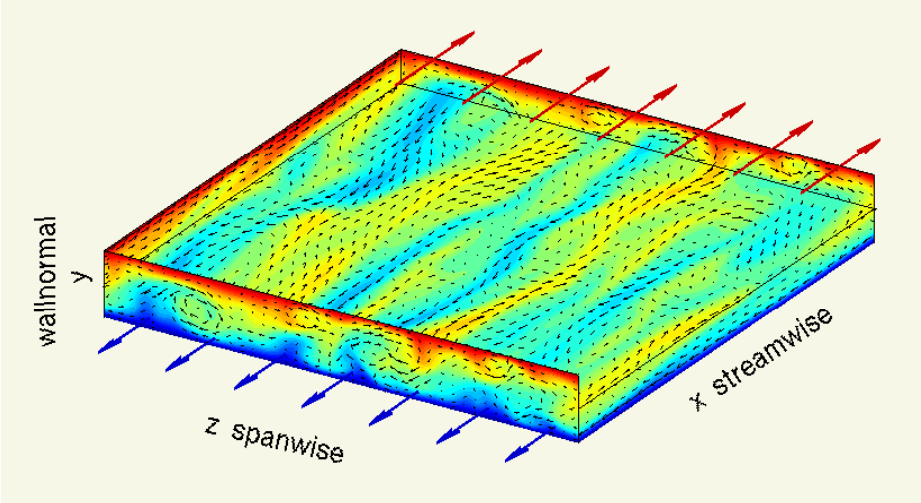

Figure 1: **Plane Couette flow simulated with Channelflow.** The flow is driven by the motion of the upper and lower walls at $y = \pm 1$, which travel with equal speeds in the $\pm x$-directions. Arrows indicate in-plane velocities; the colormap indicates the speed of the fluid in the x direction: red/blue is $u = \pm 1$. The top half of the fluid is cut away to show the flow at the $y = 0$ midplane. The flow domain is $[L_x, L_y, L_z] = [15, 2, 15]$ with periodic boundary conditions in x and z. The Reynolds number is 400, based on channel half-height and half the relative wall velocity. The flow was integrated on a $96 \times 33 \times 128$ grid with a fourth-order backwards differentiation timestepping and a timestep of $dt = 0.02$. Animations of this and other flows are available at `http://cns.physics.gatech.edu/~gibson/PCF-movies`.

1 Introduction

Channelflow is a direct numerical simulator for incompressible fluid flow on a periodic, rectangular, wall-bounded domain. Channelflow uses spectral discretization in spatial directions (Fourier x Chebyshev x Fourier), finite-differencing in time, and primitive variables (3d velocity and pressure) to integrate the incompressible Navier-Stokes equations. The mathematics are based on the spectral channel-flow algorithm in Section 7.3 of *Spectral Methods in Fluid Dynamics* by Canuto, Hussaini, Quarteroni, and Zang ([**?**]). Channelflow is written in C++ and designed to be easy to use, easy to understand, modular, extensible, and fast. Channelflow is documented, licensed under the GNU GPL version 2, and available for download at

`http://www.channelflow.org/download`

1.1 Design

Channelflow is written as a set of C++ classes that represent the major components of spectral channel-flow simulation. The Channelflow class library provides a high-level representation for expressing and performing spectral channel-flow simulations. In Channelflow's high-level syntax, fluids simulation programs are short, readable, and easily modifiable. Channelflow falls short of a providing a *language* for spectral simulation, due to the scope of the problem domain and to the difficulty of presenting a clean syntax through C++ class

libraries. But Channelflow should be good enough for a general use by fluids researchers who need a fast, simple, and extensible way to simulate channel flows.

Channelflow's classes are designed to be modular. Instances of classes behave as independent objects with automatic memory management. Auxiliary fields and computations can be added to a program with a few lines of code. In Channelflow, even the DNS algorithm is an object. This greatly increases the flexibility of DNS computations. For example, a DNS object can be reparameterized and restarted multiple times within a single program, multiple independent DNS computations can run side-by-side within the same program, and DNS computations can run as small components within a larger, more complex computations. As a result, comparative calculations that formerly required coordination of several programs through shell scripts and saved data files can be done within single Channelflow program. In this way Channelflow opens the way to a new class of computations that were not practically possible with previous codes.

Channelflow uses object-oriented programming and data abstraction to maximize the organization and readability of its library code, as well. Channelflow defines about a dozen C++ classes that act as abstract data types for the major components of spectral channel-flow simulation, as outlined by CHQZ. Each class forms a level of abstraction in which a set of mathematical operations are performed in terms of lower-level abstractions, from time-stepping equations at the top to linear algebra at the bottom. The Channelflow library code thus naturally reflects mathematical algorithm, both in overall structure and line-by-line. One can look at any part of the code and quickly understand what role it plays in the overall algorithm. One can learn the algorithm in stages, either top-down or bottom-up, by focusing on one level of abstraction at a time.

Thus Channelflow has three main benefits:

- **Ease of use:** Channelflow's high-level syntax allows simple, rapid development of particular channel-flow simulations.
- **Modularity:** Its modularity allows a broader range of channel-flow computations.
- **Intelligibility:** Its library code is organized and documented in a way that makes learning the details easy.

Additional benefits are

- **Extensibility:** Channelflow is adaptable to new needs. For example, it is fairly easy to add a new time-stepping algorithm or method of calculating the nonlinear term. It would not be difficult to implement an
- **Speed:** Channelflow is as fast as comparable Fortran codes
- **Verifiability:** Channelflow contains a test suite that verifies the correct behavior of major classes.
- **Documentation:** The documentation describes how to use the software and precisely specifies the mathematics of the algorithm.

1.2 Drawbacks and rough edges

The main potential drawbacks to using Channelflow have to do with C++. C++ is a complex language that takes some getting used to. There will be some learning overhead for those who are not familiar with it. How much overhead depends on how deeply one wants to delve. Only very basic knowledge of C++ is necessary for reparameterizing or modifying the example programs. Modification of library code will require a fair amount of experience. Secondly, C++ compilers vary in their implementation of the language. Channelflow avoids the most complex aspects of the C++ language to minimize portability problems and

learning overhead, but one can probably expect a few problems compilation errors on new platforms. Channelflow was developed on GNU/Linux with gcc-3.2. Dietmar Rempfer ported earlier versions of Channelflow to MS-Windows and Visual C++, and the source code includes his modifications as `#ifdefs`. Increasing Channelflow's portability is a major goal for future releases.

The following were not pressing concerns during the initial development of Channelflow, but are getting more attention in preparation for public release.

- **Memory footprint:** Channelflow is memory-efficient with large objects, such as flow fields that scale as N^3, but it is less careful with small things, like making extra copies of parameters in order to eliminate global variables. The wasted memory turns out to be negligible. See Section **??** for the details.

- **Import/export methods:** Most Channelflow modules have ASCII or binary input and output methods. Some ASCII output is designed to be readable by Matlab. Matlab scripts are provided for reading this data into Matlab. There are not yet import/export methods for the file formats of other channel-flow codes or for tools like Fluent and Tecplot. It should be very easy to write these, if you know the format.

- **Consistent nomenclature and syntax:** A number of inconsistencies in this regard have become apparent during preparation of the documentation. For example, `Real unorm = L2Norm(u)` computes the L^2-norm of FlowField `u`, but `Real udiv = u.divergence()` computes divergence. Some class names could be changed.

- **Coverage of problem domain:** Channelflow aims to provide elemental differential and algebraic operations for all its objects in order to allow easy computation of arbitrary quantities. But so far these operations have been written and tested on an as-needed basis, so they are probably incomplete.

- **Graphical user interface:** A basic GUI for setting parameters, driving simulations, and plotting results would be quite useful.

- **Packaging:** At this point it might be necessary to edit the Makefile to get Channelflow to compile. Ideally, Channelflow should have an autoconf system (./configure; make) and be distributed in RPM and Debian apt packages.

Help with any of these issues would be greatly appreciated.

2 Quick start

Channelflow's C++ classes are compiled into software libraries. Under normal circumstances, users of Channelflow should not need to modify the Channelflow library code. Channelflow programs, on the other hand, are relatively short sequences of statements that use the library classes in particular ways to solve particular problems. The Channelflow distribution includes several example programs that show how this is done. Two of these are presented as annotated examples in Section 2.2. Refer to the `examples` directory for other examples.

Most common simulation needs, such as the extraction of data or statistics from the integration of an autonomous flow, can probably be satisfied by modifications of the example programs. Complex problems and unusual needs will call for novel arrangements of the classes and possibly modification to the libraries.

2.1 Compilation

On a Unix system, the following commands unpack the source distribution and compile the libraries and a simple example program:

```
birbal$ tar xvpfz channelflow-0.9.17.tgz
birbal$ cd channelflow-0.9.17/src
birbal$ make libs
birbal$ cd ../examples/couette
birbal$ make couette.x
birbal$ ./couette.x
```

To change the flow and integration parameters, edit `couette.cpp` and recompile. The other subdirectories of `channelflow-0.9.17/examples` have example programs for channel flow, Poisseuille flow, Orr-Sommerfeld eigenfunctions, and the decay of a sinusoidal perturbation.

2.2 Example programs

This section presents several annotated Channelflow programs. The programs are listed and the text steps explains what's happening, line-by-line. The example programs are included in the Channelflow distribution package in the `examples` directory. See Section **??** for information on compilation and execution.

Before launching into the examples, a few brief statements about the the most important Channelflow classes: The **FlowField** class represents Fourier $\times$ Chebyshev $\times$ Fourier expansions of vector fields on three-dimensional periodic domains. The **DNS** class represents a complete Navier-Stokes approximation method, that is, a spatial discretization, a time-stepping algorithm, their parameters, and the subsidiary data structures necessary to solve the discrete equations. **ChebyCoeff, ComplexChebyCoeff, BasisFunc** represent Chebyshev expansions of real, complex, and vector-valued functions on one-dimensional finite domains.

2.2.1 couette.cpp: a simple program for plane Couette flow

Code listing 2.1 shows the main body of a simple Channelflow program, couette.cpp. The program integrates a plane Couette flow with a linear base velocity profile, using 3rd-order Runge-Kutta timestepping, constant pressure gradient, and a fixed time step. The initial fluctuating velocity field consists of small perturbations in the first few Fourier modes. The complete program is included in the Channelflow distribution package in the `examples/couette` directory.

Code listing 2.1 couette.cpp: a simple Channelflow program (line numbers added)

```
1   (skip inclusion of header files)
2
3   int main() {
4
5     // Definition of numerical parameters Nx,Ny, etc.
6     (skipped to save space)
7
8     // Construct base flow for plane Couette: U(y) = y
9     ChebyCoeff U(Ny,a,b,Physical);
10    Vector y = chebypoints(Ny, a,b);
11    for (int ny=0; ny<Ny; ++ny)
12      U[ny] = y[ny];
13    U.save("U");
14    y.save("y");
15
16    // Construct data fields: 3d velocity and 1d pressure
17    FlowField u(Nx,Ny,Nz,3,Lx,Lz,a,b);
18    FlowField q(Nx,Ny,Nz,1,Lx,Lz,a,b);
19
20    // Perturb velocity field
21    u.addPerturbations(kxmax,kzmax,magnitude,decay);
22
23    // Construct a DNS object
24    DNSFlags flags;
25    flags.timestepping = RK3;  // use 3rd order Runge-Kutta method
26    flags.constraint = BulkVelocity; // enforce bulk velocity
27
28    DNS dns(u, U, nu, dt, flags);
29
30    // Timestepping loop
31    for (Real t=0.0; t<T; t += n*dt) {
32      cout << "=========================================" << endl;
33      cout << "         t == " << t << endl;
34      cout << "       CFL == " << dns.CFL() << endl;
35      cout << "L2Norm2(u) == " << L2Norm2(u) << endl;
36
37      // Save the kx=1,kz=2 Fourier profile
38      ComplexChebyCoeff uprofile12 = u.profile(1,2,0);
39      uprofile12.makePhysical();
40      uprofile12.save("uprofile12"+i2s(int(t)));
41
42      // Take n steps of length dt
43      dns.advance(u, q, n);
44    }
45    u.binarySave("u");
46    q.binarySave("q");
47  }
```

line	meaning
1–7	The code listing skips the header-file inclusion statements and parameter definitions to save space. The parameter definitions take the form `int Nx=32; Real Lx = 2*pi;` etc.
8–14	The base flow `U` is declared as a variable of type ChebyCoeff and the y-gridpoints `y` are set as a real-valued Vector. The for-loop sets the base flow to a linear profile, $U(y) = y$. Both `U` and `y` are set and saved to disk in an ASCII Matlab-readable format.
17–18	The fluctuating velocity `u` and modified pressure `q` fields are allocated and initialized to zero. The FlowField constructor allocates memory for 3d and 1d $N_x \times N_y \times N_z$ grids, respectively. The domain of each field is set to $[0, L_x] \times [a, b] \times [0, L_z]$.
21	Random divergence-free perturbations are added to Fourier modes with `|kx| < kxmax` and `|kz| < kxmax`. The `magnitude` and `decay` parameters determine the spectral characteristics of the perturbations' Chebyshev expansions along y.
24–26	The next few statements construct a DNSFlags object and modify a few of its default values.
28	The DNS constructor allocates and initializes data needed for time-stepping calculations, based on the initial velocity field, the base flow, the viscosity, the timestep, and the flags.
31–35	A for-loop advances time from `T0` to `T1` in steps of length `n*dt`. At each step, the time the CFL number, and the L_2-norm of the velocity field are printed out.
38–40	The Fourier profile $\tilde{u}_{12}(y)$ is extracted from velocity field `u`, transformed from spectral representation to physical gridpoint values, and then saved to disk in ASCII Matlab-readable form, with filenames indicating the integration time.
43	The DNS object `dns` advances the velocity and pressure fields `n` steps of length `dt`.
45–46	After the time-stepping loop finishes, the velocity and modified pressure fields are saved to disk in binary form and the main program.

Of course, what's notable about couette.cpp is what *doesn't* appear, for example, allocation of arrays, Fourier transforms, calculation of nonlinear terms, influence-matrix calculations, and solution of linear algebra problems. These operations are carried out internally by the objects to which they pertain. Most of the work occurs within DNS construction (line 28, `DNS(u,U,nu,dt,flags)`), and the DNS advance function (line 43, `dns.advance(u,q,n)`).

2.2.2 couette2.cpp: variable time-stepping, statistics, and start-up from saved fields

The couette2.cpp example program adds to couette.cpp variable time-stepping, simple statistics, and start-up from saved field to couette.cpp. The statistics calculated in couette2.cpp are the mean-velocity profile and the mean drag on the lower wall. Code listings 2.2 and 2.3 show the program in its entirety. The program is included in the Channelflow source distribution at `examples/couette/couette2.cpp`.

Code listing 2.2 couette2.cpp: variable time-stepping, statistics, and start-up from saved fields

```
1  #include <iostream>
2  #include <iomanip>
3  #include "vector.h"
4  #include "chebyshev.h"
5  #include "flowfield.h"
6  #include "nsintegrator.h"
7
8  int main() {
9
10   // Define flow parameters
11   const Real Reynolds = 400.0;
12   const Real nu = 1.0/Reynolds;
13
14   // Define integration parameters
15   const Real dtmax = 0.15;
16   const Real dtmin = 0.05;
17   const Real CFLmax = 0.90;
18   const Real CFLmin = 0.5;
19   const Real dT  = 1.0;     // plot interval
20   const Real T0  = 100.0;    // start time
21   const Real T1  = 200.0;    // end time
22
23   // Load velocity, modified pressure, and base flow from disk.
24   FlowField u("u100");
25   FlowField q("q100");
26   ChebyCoeff U("U");
27
28   // Get y-domain information from velocity field.
29   Real a = u.a();
30   Real b = u.b();
31   int Ny = u.Ny();
32
33   // Construct a DNS object
34   DNSFlags flags;
35   flags.timestepping = RK3;
36   flags.constraint = PressureGradient;
37
38   TimeStep dt((dtmax+dtmin)/2, dtmin, dtmax, dT, CFLmin, CFLmax);
39   DNS dns(u, U, nu, dt, flags, T0);
40
41   ChebyCoeff u00mean(Ny,a,b,Spectral);
42   Real dragmean = 0.0;
43   int count = 0;
```

Code listing 2.3 couette2.cpp cont'd: variable time-stepping, statistics, and start-up from saved fields

```
44
45    for (Real t=T0; t<T1; t += dT) {
46
47      // Get kx=kz=0 Fourier component u00(y) and compute drag
48      ChebyCoeff u00 = Re(u.profile(0,0,0));
49      ChebyCoeff du00dy = diff(u00);
50      Real drag = nu*du00dy.eval_a();
51
52      u00mean += u00;
53      dragmean += drag;
54      ++count;
55
56      // Save stuff
57      string time = i2s(int(t));
58      u00.save("uprofile00_"+time);
59      Re(u.profile(1,2,0)).save("uprofile12_"+time);
60
61      cout << "========================================" << endl;
62      cout << "         t == " << t << endl;
63      cout << "        dt == " << dt << endl;
64      cout << "       CFL == " << dns.CFL() << endl;
65      cout << "L2Norm2(u) == " << L2Norm2(u) << endl;
66      cout << "      drag == " << drag << endl;
67
68      // Take n steps of length dt
69      dns.advance(u, q, dt.n());
70
71      // Adjust timestep if CFL number is too large or too small.
72      if (dt.adjust(dns.CFL())) {
73        cout << "adjusting timestep" << endl;
74        dns.reset(nu, dt);
75      }
76    }
77
78    // Compute means
79    dragmean /= count;
80    u00mean /= count;
81
82    // Fourier-transform u00mean, save, and print
83    u00mean.makePhysical();
84    u00mean.save("u00mean");
85    cout << "mean drag == " << dragmean << endl;
86  }
```

line	meaning
1-6	These header-file inclusion statements declare standard C++ I/O classes and a number of Channelflow classes.
11-21	Definitions of flow and integration parameters.
24-26	Load the velocity, the modified pressure, and the base flow that were saved to disk in `couette.cpp`. Channelflow's binary storage format for FlowFields includes data such as the gridsize, the domain, and the Physical/Spectral state of the data, in addition to the data itself. Thus the FlowField `u` at line 24 is reconstructed in exactly the same state as the FlowField saved at at line 45 in `couette.cpp`. `ChebyCoeff` uses an ASCII, Matlab-readable file format, with parametric information stored in a comment line.
29-31	Extract information about the y-domain from FlowField `u`.
34-36	Set a few flags for the DNS.
38	Construct a TimeStep object for variable time-stepping. The initial timestep is set halfway between its minimum and maximum bounds. The timestep `dt` will vary during the integration to keep the CFL number and the timestep between the given bounds, but always as a whole-number fraction of the plot interval `dT`, i.e. $dt = dT/n$ for some integer n.
39	Construct a DNS based on the velocity field `u`, the base flow `U`, viscosity `nu`, TimeStep `dt`, and starting time `T0`. The starting time of `couette2.cpp` equals the end time of `couette.cpp`.
41-43	Construct variables for accumulating sums for the calculation of the mean drag and the mean $k_x, k_z = 0, 0$ Fourier profile.
45	Begin time-stepping loop. Note that time increases by the plot interval `dT` each pass through the loop.
48-50	Extract Fourier profile $\tilde{u}_{00}(y)$, compute $\mathrm{Re}(\partial\tilde{u}_{00}/\partial y)$, and $\nu\,\mathrm{Re}(\partial\tilde{u}_{00}/\partial y|_{y=a})$. Line 50 uses a special efficient function for evaluating ChebyCoeffs at an endpoint.
52-54	The current values of $\tilde{u}_{00}(y)$ and the drag are added into their summation variables.
57-59	Save the current $\tilde{u}_{00}(y)$ and $\tilde{u}_{12}(y)$ profiles to disk, with file names that indicate the integration time. Line 59 illustrates how to save a profile to disk without the use of a temporary ChebyCoeff variable.
61-66	Print interesting information.
69	Advance `n` timesteps of length `dt`.
72-74	Check if the CFL condition is out of bounds and adjust if necessary. If adjustment occurs, the `dt.adjust` function returns `true`, and the DNS object `dns` is recalibrated for the new timestepping interval.
79-80	Divide the sums by the number of samples to get the means.
83-85	Transform `u00mean` to gridpoint values, save to disk, and print the mean drag.

3 Guide to main classes

This section is a user's guide to the behavior and meaning of the main Channelflow classes. The goal is to discuss how to use and control the main classes in top-level Channelflow programs. Parts of the DNS class are in full mathematical detail in Section 4. At this point the documentation falls short of an exhaustive reference manual. Please consult the header files and source code for more information.

3.1 ChebyCoeff

3.1.1 Description

The ChebyCoeff class represents real-valued Chebyshev expansions of functions on the domain $[a, b]$, of the form

$$f(y) = \sum_{n=0}^{N-1} \widehat{f}_n \bar{T}_n(y) \tag{1}$$

where $\bar{T}_n(y)$ is the nth Chebyshev polynomial rescaled to the interval $y \in [a, b]$. That is,

$$\bar{T}_n(y) = T_n\left(\frac{2y - (b+a)}{b-a}\right) \tag{2}$$

ChebyCoeffs are on a general domain $[a, b]$ instead of the usual $[-1, 1]$ to facilitate programs that involve more than one approximation domain. In general the right-hand side of eqn. 1 is an approximation of a function f. For simplicity, we treat the function and its expansion as identically equal.

The spectral coefficients of a function f can be computed efficiently from the function values taken at a discrete set of Chebyshev gridpoints. Let

$$y_n = \frac{b+a}{2} + \frac{b-a}{2}\cos\left(\frac{n\pi}{N-1}\right), \quad n \in [0, N-1] \tag{3}$$

and let $f_n = f(y_n)$. Then a fast cosine transform can be used to transform the function values $\{f_0, f_1, \ldots, f_{N-1}\}$ into the spectral coefficients $\{\widehat{f}_0, \widehat{f}_1, \ldots, \widehat{f}_{N-1}\}$, and vice versa. The main point of the ChebyCoeff class is to store an array of these function values or Chebyshev spectral coefficients, transform back and forth between them, and approximate properties of the function $f(y)$ based on the spectral expansion 1.

3.1.2 Data access, transforms, and state

The ChebyCoeff class has a data array that stores either function values or spectral coefficients and a flag that indicates which state the data array is in. For the ChebyCoeff object `f`, elements of the data array are accessed with the square-bracket operator, e.g. `f[n]`. The function `f.state()` returns `Physical` if the array represents function values and `Spectral` if spectral coefficients. The Physical/Spectral state is set at construction time and toggled when the ChebyCoeff's transform functions are called. For example, the following block of code constructs a length-N ChebyCoeff object `f` on the domain $[-1, 1]$, sets the function values $\{f_n\}$ to $\{\sin(\pi y_n)\}$, transforms the ChebyCoeff from `Physical` to `Spectral`, and then prints the zeroth spectral coefficient.

```
Vector y = chebypoints(N,-1,1);
ChebyCoeff f(N,-1,1,Physical);
```

```
for (int n=0; n<N; ++n)
  f[n] = sin(pi*y[n]);

if (f.state() == Physical)  // will be true
  f.chebyfft();             // transform f from Physical to Spectral
cout << f[0] << endl;       // print coeff of T_0

if (f.state() == Spectral)  // will be true
  f.ichebyfft();            // transform f from Spectral to Physical
cout << f[0] << endl;       // print value of sin(pi*y[0])
```

Note that the return-value of `f.state()` is a variable of type `fieldstate`, with two possible values: `Physical` and `Spectral`.

Several other forms of transform function are provided for convenience and efficiency. The functions

```
f.makeSpectral();  // if f.state()!=Spectral, transform to Spectral
f.makePhysical();  // if f.state()!=Physical, transform to Physical
```

are state-checking versions of `f.chebyfft()` and `f.ichebyfft()`. Using these forms eliminates the possibility of performing the same transform twice in a row. A third form performs the transform specified by an argument. If `s` be a variable of type `fieldstate`, then

```
f.makeState(s);  // if f.state()!=s, transform to state s
```

transforms `f` to that state `s`.

Each of the transforms discussed so far has a more efficient form that takes a `ChebyTransform` argument. The `ChebyTransform` class is described in Section 3.3. For now, suffice it to say that some common work can be factored out of multiple calls to `ChebyCoeff` transforms of equal length by constructing a `ChebyTransform` object and passing it to the `ChebyCoeff` transform functions, as in

```
int N = f.length();
ChebyTransform trans(N);

f.chebyfft(trans);
f.ichebyfft(trans);
f.makeSpectral(trans);
f.makePhysical(trans);
f.makestate(trans, s);
```

These forms are the preferred forms for `ChebyCoeff` transforms. They should be used in frequently repeated calculations. The forms without `ChebyTransform` arguments are conveniences for use when efficiency is not an issue.

For a more complete discussion of the numerics of Chebyshev approximation, cosine transforms, and the FFT, see *Numerical Recipes in C* ([**?**]). But don't implement Numerical Recipe's algorithm! It's less numerically stable than FFTW's algorithm (termed RDEFT, for "Discrete Real Even Fourier Transform").

3.1.3 Input/output

`ChebyCoeff` I/O is done with a `save` function and a constructor that both take a filename argument. For example, given a `ChebyCoeff f`,

```
string filebase = "foo";
f.save(foo);

ChebyCoeff g(filebase);
```

saves `f` to disk in file `foo.asc` and then constructs `g` based on the data stored in the file. At after construction, `g` will be identical to `f`, with the same length, bounds, state, and data.

The ASCII file format for `ChebyCoeff` is

```
% N a b s
f[0]
f[1]
.
.
.
f[N-1]
```

where N is the integer expansion length, a and b are the double-precision domain bounds, s is a character `P` or `S`, indicating the `Physical` or `Spectral` state, and the $f[n]$ are double-precision values of function values or spectral coefficients. The `%` character marks the first line as a comment in Matlab, so that the file can be read into Matlab as an $N \times 1$ matrix with the command `load foo.asc`.

3.1.4 Other functions

`ChebyCoeff` provides a number of other functions for arithmetical operations, computing norms, derivatives, etc. Note that most of these functions operate on spectral coefficients and so require the `ChebyCoeff` to be in `Spectral` state. Please refer to the header files for a complete list of functions. A few quick examples:

```
ChebyCoeff f(N,a,b,Spectral);
ChebyCoeff g(N,a,b,Spectral);

f.randomize(magn, decay); // set f[n] = magn*random()*pow(decay,n)
g.randomize(magn, decay); // ditto for g

f += g;                   // add g to f;
Real x = L2Dist2(f,g);    // 1/(b-a) Integral_a^b (f-g)^2 dy
Real y = chebyNorm2(f);   // 2/(b-a) Integral_a^b f^2/sqrt((y-a)(b-y)) dy

ChebyCoeff dfdy = diff(f);   // compute derivative of f
ChebyCoeff F = integrate(f); // compute integral of f, set F.mean() to 0

Real f_a = f.eval_a();       // return function value at lower bound
Real f_m = f.eval((b+a)/2);  // return function value at midpoint
```

3.2 ComplexChebyCoeff

`ComplexChebyCoeff` represents complex-valued Chebyshev expansions of the form of eqn. 1 and follows the same syntax as `ChebyCoeff` in almost all respects. There's just one thing to watch out for: *you can't assign into* `f[n]`! Or rather, you can assign into `f[n]`, and it will compile and run with no complaints, but it won't have any effect on the value of `f[n]`. To set the value of `f[n]`, use

```
f.set(n, z);  // CORRECT: sets f[n] to z
```

and not

```
f[n] = z;     // INCORRECT: doesn't change f[n]
```

This behavior is due to a bad design decision that I hope to correct before the channelflow-1.0.0 release. The `f[n]` syntax works just fine for extracting spectral coefficients or function values.

```
Complex z = f[n];  // OK: sets z to f[n]
```

The `ComplexChebyCoeff` I/O methods follow the same syntax as `ChebyCoeff`, but the ASCII file format has two columns for the real and imaginary parts of the data

```
% N a b s
Re(f[0]) Im(f[0])
Re(f[1]) Im(f[1])
.
.
.
Re(f[N-1]) Im(f[N-1)
```

Other numerical functions are the usual complex generalizations. For example, `L2InnerProduct2(f,g)` computes $1/(b-a)\int_a^b fg^* dy$.

3.3 ChebyTransform

`ChebyTransform` is a wrapper class for the discrete cosine transforms of the elegant and powerful FFTW3 library, by Matteo Frigo and Steven G. Johnson (see `www.fftw.org` and [**?**]). In order to use `ChebyTransform` well, one should know a few things about FFTW. FFTW uses code generation and run-time profiling to find the optimal FFT algorithm for given transform length on a given processor. Once that optimal FFT is found, it can be reused as many times as needed. Thus optimal use of FFTW consists of a relatively high-cost "learning" phase and repeated execution of the optimal FFT algorithm. If only a single transform needs to be calculated, a good algorithm can be estimated with heuristics or the results of previous learning. FFTW's accumulated learning is called "wisdom." Wisdom can be saved to disk and recalled in subsequent runs.

The `ChebyTransform` class does FFTW's learning or estimating during construction and repeated execution in calls to its transform functions. The `ChebyTransform` constructor takes an integer `N` argument that specifies the transform length and an optional integer flag argument that specifies how FFTW should learn or estimate. The default behavior is wisdom-based estimation rather than learning. For example,

```
ChebyTransform trans(N);
```

constructs a `ChebyTransform` with wisdom-based estimation of the optimal FFT for length N.

```
ChebyTransform trans(N, FFTW_MEASURE);
```

performs the high-cost "learning" phase to find a nearly optimal length-N transform.

```
ChebyTransform trans(N, FFTW_MEASURE | FFTW_WISDOM);
```

learns the optimal transform and adds remembers it to improve any subsequent estimates. The `FFTW_PATIENT` flags can be substituted in place of `FFTW_MEASURE` to run a more exhaustive search and find a more nearly optimal algorithm. For exact details on FFTW flags, see the FFTW documentation. Channelflow provides two methods for saving FFTW wisdom to disk and rereading it, `fftw_loadwisdom()` and `fftw_savewisdom()`, both taking an optional filename argument. The filename defaults to `~/.fft-wisdom` if left unspecified.

I generally load wisdom at the beginning of my programs and save it at the end. However, I recommend reading the FFTW documentation on wisdom. You can undermine the performance of Channelflow by developing wisdom on one machine architecture and using it on another.

3.4 FlowField

The FlowField class represents vector-valued Fourier $\times$ Chebyshev $\times$ Fourier expansions whose mathematical form is

$$\mathbf{u}(\mathbf{x}) = \sum_{k_x=-N_x/2+1}^{N_x/2} \sum_{n_y=0}^{N_y-1} \sum_{k_z=-N_z/2+1}^{N_z/2} \widehat{\widetilde{\mathbf{u}}}_{k_x,n_y,k_z} \bar{T}_{n_y}(y)\, e^{2\pi i(k_x x/L_x + k_z z/L_z)} \tag{4}$$

where N_x and N_z are even, $\mathbf{x} = (x, y, z)$, and $\bar{T}_m$ is the mth Chebyshev polynomial rescaled for the domain $y \in [a, b]$. For N_x odd, the sum is over $k_x = -N_x/2 + 1$ to $N_x/2 - 1$, and likewise for N_z. The double tilde/hat notation on the spectral coefficients indicates that the coefficients result from a combined Fourier transform in xz and a Chebyshev transform in y.

The primary function of FlowField, like ChebyCoeff, is to store data in arrays, transform it back and forth from Spectral and Physical representations, allow access to the physical data or spectral coefficients, and perform mathemtical operations related to the spectral expansions. FlowField, however, is considerably more complicated, so we discuss it in greater detail.

3.4.1 A few examples of use

The following code snippet declares a 3d FlowField on (N_x, N_y, N_z) gridpoints, on the box $[0, L_x] \times [a, b] \times [0, L_z]$, sets the field to physical values provided by an external function, and then transforms the field to a spectral representation

```
FlowField u(Nx,Ny,Nz,3,Lx,Lz,a,b,Physical,Physical);

Vector x = u.xgridpts();
Vector y = u.ygridpts();
Vector z = u.zgridpts();

for (int i=0; i<3; ++i)
  for (int ny=0; ny<Ny; ++ny)
    for (int nx=0; nx<Nx; ++nx)
      for (int nz=0; nz<Nz; ++nz)
        u(nx,ny,nz,i) = f(x(nx), y(ny), z(nz), i);

u.makeSpectral();
```

The next code snippet declares a 3d FlowField in the default `Spectral, Spectral` state, assigns successively smaller random values to the Chebyshev coefficients of the $(k_x, k_z) = (-1, 4)$ Fourier mode,

computes the curl of the field, ouputs the L_2 norm of the curl, and then transforms and prints the curl's physical gridpoint values.

```
FlowField f(Nx,Ny,Nz,3,Lx,Lz,a,b);

int kx = -1;
int kz =  4;
int mx = u.mx(kx);
int mz = u.mz(kz);
Real magn = 1.0;
Real decay = 0.5;

for (int my=0; my<f.My(); ++my) {
  for (int i=0; i<f.Nd(); ++i)
    f.cmplx(mx,my,mz,i) = magn*randomComplex();
  magn *= decay;
}

FlowField g = curl(f);
cout << "L2Norm(curl u) == " << L2Norm(g) << endl;

g.makePhysical();
for (int i=0; i<g.Nd(); ++i)
  for (int nx=0; nx<g.Nx(); ++nx)
    for (int ny=0; ny<g.Ny(); ++ny)
      for (int nz=0; nz<g.Nz(); ++nz)
        cout << g(nx,ny,nz,i) << ' ';
```

3.4.2 Mode numbers, wave numbers, and grid points

Fun facts about FlowFields:

- FlowFields are allocated in terms of their physical grid sizes $N_x \times N_y \times N_z$ and vector dimension N_d.
- Physical gridpoint data is real-valued, indexed by **gridpoint indices** (n_x, n_y, n_z), and accessed with syntax `u(nx,ny,nz,i)`.
- Spectral coefficient data is complex-valued, indexed by **mode numbers** (m_x, m_y, m_z), and accessed with syntax `u.cmplx(mx,my,mz,i)`.
- However, FlowField's xz and y transforms are independent, so that mixed Physcial,Spectral and Spectral,Physical states are possible, too.
- `u.Nx()`, `u.Ny()`, and `u.Nz()` indicate the **number of gridpoints** in x, y, and z for a given FlowField.
- `u.Mx()`, `u.My()`, and `u.Mz()` indicate the **number of modes** in x, y, and z for a given FlowField.
- The ranges of gridpoint indices are $0 \leq n_x < N_x$, etc.
- The ranges of mode numbers are $0 \leq m_x < M_x$, etc.

- Because FlowField's Fourier transform maps Real-valued gridpoint data into (roughly) half as many Complex-valued spectral coefficients, the number of gridpoints differs from the number of modes. In particular,

$$M_x = N_x \tag{5}$$
$$M_y = N_y \tag{6}$$
$$M_z = N_z/2 + 1 \tag{7}$$

- The mode numbers m_x and m_z are merely indices into the complex array of spectral coefficients; they are *not* equal to the Fourier **wave numbers** k_x and k_z that appear in eqn. 4. However, the two are related by

$$m_x = \begin{cases} k_x + M_x & -M_x/2 + 1 \le k_x < 0 \\ k_x & 0 \le k_x \le M_x/2 \end{cases} \tag{8}$$

$$m_z = \begin{cases} k_z & 0 \le k_z < M_z \\ \text{undefined} & k_z < 0 \end{cases} \tag{9}$$

and

$$k_x = \begin{cases} m_x & 0 \le m_x \le M_x/2 \\ m_x - M_x & M_x/2 < m_x < M_x \end{cases} \tag{10}$$

$$k_z = m_z \quad 0 \le m_z < M_z \tag{11}$$

- Gridpoint indices are related to the coordinates of gridpoints by

$$x_{n_x} = \frac{n_x L_x}{N_x} \qquad 0 \le n_x < N_x \tag{12}$$

$$y_{n_y} = \frac{b+a}{2} + \frac{b-a}{2} \cos\left(\frac{n_y \pi}{N_y - 1}\right) \quad 0 \le n_y < N_y \tag{13}$$

$$z_{n_z} = \frac{n_z L_z}{N_z} \qquad 0 \le n_z < N_z \tag{14}$$

3.4.3 FlowField states, access methods, and transforms

The primary functions of FlowField are to store data, either as spectral coefficients $\widehat{\widetilde{\mathbf{u}}}_{m_x,m_y,m_z}$ or as physical gridpoint values $\mathbf{u}(x_{n_x}, y_{n_y}, z_{n_z})$, and to transform between spectral and physical representations as needed. FlowField performs its x, z Fourier transforms together and its y Chebyshev transform separately, so that a FlowField can be in any one of four states:
Tildes ($\widetilde{u}$) denote Fourier coefficients; hats ($\widehat{u}$) Chebyshev coefficients. In what follows we use the abbreviations PP, PS, SP, and SS for the four states of FlowFields, with the xz state listed first.

FlowFields can be initialized in any of the four states. For example,

```
FlowField u(Nx,Ny,Nz,Lx,Lz,a,b,Physical,Spectral);
```

constructs a FlowField in state PS. FlowField has two functions for checking state:

```
fieldstate xzstate = u.xzstate();
fieldstate ystate  = u.ystate();
```

A `fieldstate` has value `Physical` or `Spectral`.

Table 1: **FlowField states and access functions.**

xz, y state	access function	mathematical meaning	type	description
Physical, Physical	`u(nx,ny,nz,i)`	$u_i(x_{n_x}, y_{n_y}, z_{n_z})$	Real	gridpoint values
Physical, Spectral	`u(nx,my,nz,i)`	$\widehat{u}_{i,m_y}(x_{n_x}, z_{n_z})$	Real	mixed state
Spectral, Physical	`u.cmplx(mx,ny,mz,i)`	$\widetilde{u}_{i,m_x,m_z}(y_{n_y})$	Complex	mixed state
Spectral, Spectral	`u.cmplx(mx,my,mz,i)`	$\widehat{\widetilde{u}}_{i,m_x,m_y,m_z}$	Complex	spectral coeffs

3.4.4 FlowField access methods

As noted above, the xz Fourier transform applied to a P* FlowField puts the FlowField in S* state and switches its data from real-valued to complex-valued. FlowField has separate functions for accessing real and complex-valued data. For example, real-valued P* FlowField data is set with

```
u(nx,ny,nz,i) = 4.0;                // for u in state P*
```

Complex-valued S* FlowField data is set with

```
u.cmplx(mx,my,mz,i) = 4.0 + 3.0*I; // for u in state S*
```

You *must* use the data access method that is appropriate for the FlowField's xz state. Using the wrong access method will corrupt FlowField data and lead to meaningless results. To ensure correct use, Channelflow provides debugging libraries that check FlowField state during each data access call. If the access method doesn't match the state, an error message will be printed and execution will stop. If you're unsure of the correctness of your code, link to the debugging libraries and run. See Section **??**.

The meaning of the assignments in the above examples depends further on the FlowField's `ystate`. For example, the above complex assignment sets the value of $\widetilde{u}_{i,m_x,m_z}(y_{m_y})$ if `u` is in state SP and $\widehat{\widetilde{u}}_{i,m_x,m_y,m_z}$ if `u` is SS.

FlowField provides conversion functions for array indices, gridpoint positions, and wavenumbers:

```
Real x = u.x(nx);   // get the x coordinate of the nxth gridpoint
Real y = u.y(ny);
Real z = u.z(nz);

int kx = u.kx(mx);  // get the wavenumber kx of the mxth mode
int kz = u.kz(mz);

int mx = u.mx(kx);  // get the mode number mx of the kxth Fourier mode
int mz = u.mz(kz);
```

Other functions provide bounds for the spatial indices. For example, a PP FlowField can be set to zero by

```
for (int i=0; i<u.Nd(); ++i)
  for (int ny=0; ny<u.Ny(); ++ny)
    for (int nx=0; nx<u.Nx(); ++nx)
      for (int nz=0; nz<u.Nz(); ++nz)
         u(nx,ny,nz,i) = 0.0;
```

whereas an SS FlowField is can be zeroed with

```
Complex zero = 0.0 + 0.0*I;
for (int i=0; i<u.Nd(); ++i)
  for (int my=0; my<u.My(); ++my)
    for (int mx=0; mx<u.Mx(); ++mx)
      for (int mz=0; mz<u.Mz(); ++mz)
         u.cmplx(mx,my,mz,i) = zero;
```

3.4.5 FlowField transform functions

The FlowField transform functions are

```
u.realfft_xz();          // P* -> S*
u.irealfft_xz();         // S* -> P*
u.chebyfft_y();          // *P -> *S
u.ichebyfft_y();         // *S -> *P

u.makeSpectral_xz();     // ** -> S*
u.makePhysical_xz();     // ** -> P*
u.makeSpectral_y();      // ** -> *S
u.makePhysical_y();      // ** -> *P

u.makeSpectral();        // ** -> SS
u.makePhysical();        // ** -> PP

u.makeState(Spectral, Physical); // ** -> SP
```

Let us restrict attention for the moment to scalar functions of two variables. For example, hold y fixed and let $f(x,z) = u_0(x,y,z)$. FlowField's discrete xz-Fourier transform and inverse are defined as

$$\widetilde{f}_{k_x,k_z} = \frac{1}{L_x L_z} \sum_{n_x=0}^{N_x-1} \sum_{n_z=0}^{N_z-1} f(x_{n_x}, z_{n_z})\, e^{-2\pi i (k_x x_{n_x}/L_x + k_z z_{n_z}/L_z)}\, \Delta x\, \Delta z, \tag{15}$$

$$f(x_{n_x}, z_{n_z}) = \sum_{k_x=-N_x/2+1}^{N_x/2} \sum_{k_z=-N_z/2+1}^{N_z/2} \widetilde{f}_{k_x,k_z} e^{2\pi i (k_x x_{n_x}/L_x + k_z z_{n_z}/L_z)} \tag{16}$$

for $f(x,z)$ on the domain $x \in L_x\mathbb{T}$ and $z \in L_z\mathbb{T}$, where $\mathbb{T}$ is the periodic unit interval. The gridpoints and stepsizes are defined by $x_{n_x} = n_x \Delta_x$, $z_{n_z} = n_z \Delta z$, $\Delta x = L_x/N_x$, and $\Delta z = L_z/N_z$.

Compare these to the continuous Fourier transform and its inverse,

$$\tilde{f}_{k_x,k_z} = \frac{1}{L_x L_z} \int_0^{L_x} \int_0^{L_z} f(x,z)\, e^{-2\pi i (k_x x/L_x + k_z z/L_z)}\, dx\, dz, \tag{17}$$

$$f(x,z) = \sum_{k_x=-\infty}^{\infty} \sum_{k_z=-\infty}^{\infty} \tilde{f}_{k_x,k_z} e^{2\pi i (k_x x/L_x + k_z z/L_z)}. \tag{18}$$

Note the notational distinction of wide versus narrow tildes between the discrete and continuous transforms. Eqn. 15 is a trapezoidal approximation to eqn. 17, so as $N_x, N_z \to \infty$, $\widetilde{f}_{k_x k_z} \to \tilde{f}_{k_x k_z}$. Thus the discrete Fourier transform and inverse can be viewed as a finite-sum approximation to the continous transform and inverse.

For sufficiently smooth f, the discrete spectral coefficients can be used to form an uniformly convergent approximation of f. Define

$$f^{N_x,N_z}(x,z) = \sum_{k_x=-N_x/2+1}^{N_x/2} \sum_{k_z=-N_z/2+1}^{N_z/2} \widetilde{f}_{k_x,k_z} e^{2\pi i(k_x x/L_x + k_z z/L_z)} \tag{19}$$

If f is sufficiently smooth, f^{N_x,N_z} converges to f uniformly (see [**?**] for details). Because analytic functions are available only for setting initial conditions (and then only to finite precision), Channelflow documentation generally drops the superscripts and treats f^{N_x,N_z} as if it were f exactly.

Note that by eqn. 17, $\widetilde{f}_{k_x+N_x,k_z} = \widetilde{f}_{k_x,k_z}$, and likewise for k_z. This allows some flexibility in the range over which the wavenumbers are chosen to vary. Channelflow uses zero-centered wavenumber ranges to reflect symmetry in the power-spectra of physical data and to assure that spectral differentiation is well-behaved. For example, refer to range of the k_x and k_z indices in eqns. 16 and 19.

FlowField's Chebyshev transform works by looping over n_x, n_z, in each case considering $f(y) = u_i(x,y,z)$ with x, z fixed, and applying the transform described in Section 3 to convert the function values $\{f_0, f_1, \ldots, f_{N_y-1}\}$ to Chebyshev coefficients $\{\widehat{f}_0, \widehat{f}_1, \ldots, \widehat{f}_{N_y-1}\}$.

3.4.6 FlowField differential operators and norms

`FlowField f,g` Note: If `f` is a 3d vector-valued FlowField, `FlowField g = grad(f);` produces a tensor-valued field g, with $g_{ij} = \partial f_i/\partial x_j$. However, FlowFields were written with only scalar and vector fields in mind, so g is stored as a 9d vector-valued field. A helper function $i3j(i,j) = i*3+j$ is provided to simplify access to the tensor field components. For example,

```
FlowField g = grad(f);
Complex g01 = g(mx,my,mz,i3j(0,1)); // i.e. g01 = df_0/dx_1 = df_0/dy
```

This is incomplete (since it only works for 2-tensors of 3d fields) and a bit of a kludge. At some point I hope to generalize FlowFields to more general tensors, at which point the syntax will have to change.

3.4.7 FlowField's layout in memory

Given the above relations, the expansion eqn. 4 can be rewritten as

$$\mathbf{u}(\mathbf{x}) = \sum_{m_x=0}^{M_x-1} \sum_{m_y=0}^{M_y-1} \sum_{m_z=0}^{M_z-1} \widehat{\widetilde{\mathbf{u}}}_{m_x,m_y,m_z} \bar{T}_{m_y}(y)\, e^{2\pi i(k_x x/L_x + k_z z/L_z)} + \textit{complex conjugate} \tag{20}$$

which more closely reflects how spectral FlowField data is laid out in computer memory and how it is accessed in Channelflow programs. Note that the negative-k_z modes are not included in the sum (they are accounted for by the implicit addition of the sum's complex conjugate), and that the the meaning and the ranges of the indices on $\widehat{\widetilde{\mathbf{u}}}$ have changed between eqn. 4 and 20.

FlowFields use four-dimensional data storage arrays. Three dimensions are for the x, y, z spatial dimensions; another dimension allows for the components i of the vector-valued field. In memory, the data is laid out as a long, one-dimensional array. Stepping through sequential memory locations, z varies most quickly, followed by x (to form the xz planes described below), then y, then i. The order was chosen to optimize the xz Fourier transforms. Hence the best looping order for accessing FlowField data is `i,ny,nx,nz`, with `i` outermost and `z` innermost.

FlowField's xz memory layout is taken directly from FFTW. Figure 2 illustrates xz memory layout for the case $N_x = 8$ and $N_z = 10$. The upper picture represents an $N_z \times N_x$ array of double-precision real numbers,

Table 2: FlowField differential operators

Convenience form	Preferred form	Meaning
`FlowField g = xdiff(f);`	`xdiff(f,g);`	$\mathbf{g} = \partial\mathbf{f}/\partial x$
`FlowField g = xdiff(f,n);`	`xdiff(f,g,n);`	$\mathbf{g} = \partial^n\mathbf{f}/\partial x^n$
`FlowField g = ydiff(f);`	`ydiff(f,g);`	$\mathbf{g} = \partial\mathbf{f}/\partial y$
`FlowField g = ydiff(f,n);`	`ydiff(f,g,n);`	$\mathbf{g} = \partial^n\mathbf{f}/\partial y^n$
`FlowField g = zdiff(f);`	`zdiff(f,g);`	$\mathbf{g} = \partial\mathbf{f}/\partial z$
`FlowField g = zdiff(f,n);`	`zdiff(f,g,n);`	$\mathbf{g} = \partial^n\mathbf{f}/\partial z^n$
`FlowField g = xdiff(f,m,n,p);`	`diff(f,g,m,n,p);`	$\mathbf{g} = \partial^{m+n+p}\mathbf{f}/\partial x^m\partial y^n\partial z^p$
`FlowField g = grad(f);`	`grad(f,g);`	$g = \nabla f, \quad g_i = \partial f/\partial x_i$ for 1d f
		$\mathbf{g} = \nabla\mathbf{f}, \quad g_{ij} = \partial f_i/\partial x_j$ for 3d f
`FlowField g = lapl(f);`	`lapl(f,g);`	$\mathbf{g} = \nabla^2\mathbf{f}$
`FlowField g = div(f);`	`div(f,g);`	$g = \nabla\cdot\mathbf{f}$
`FlowField g = curl(f);`	`curl(f,g);`	$\mathbf{g} = \nabla\times\mathbf{f}$
`FlowField g = norm(f);`	`norm(f,g);`	$\mathbf{g} = \|\mathbf{f}\|$
`FlowField g = norm2(f);`	`norm2(f,g);`	$\mathbf{g} = \|\mathbf{f}\|^2$
`FlowField g = energy(f);`	`energy(f,g);`	$\mathbf{g} = \frac{1}{2}\|\mathbf{f}\|^2$
`FlowField g = cross(f);`	`cross(f,h,g);`	$\mathbf{g} = \mathbf{f}\times\mathbf{h}$
`FlowField g = dot(f);`	`dot(f,h,g);`	$g = \mathbf{f}\cdot\mathbf{h}$
`FlowField g = outer(f);`	`outer(f,h,g);`	$g_{ij} = f_i h_j$

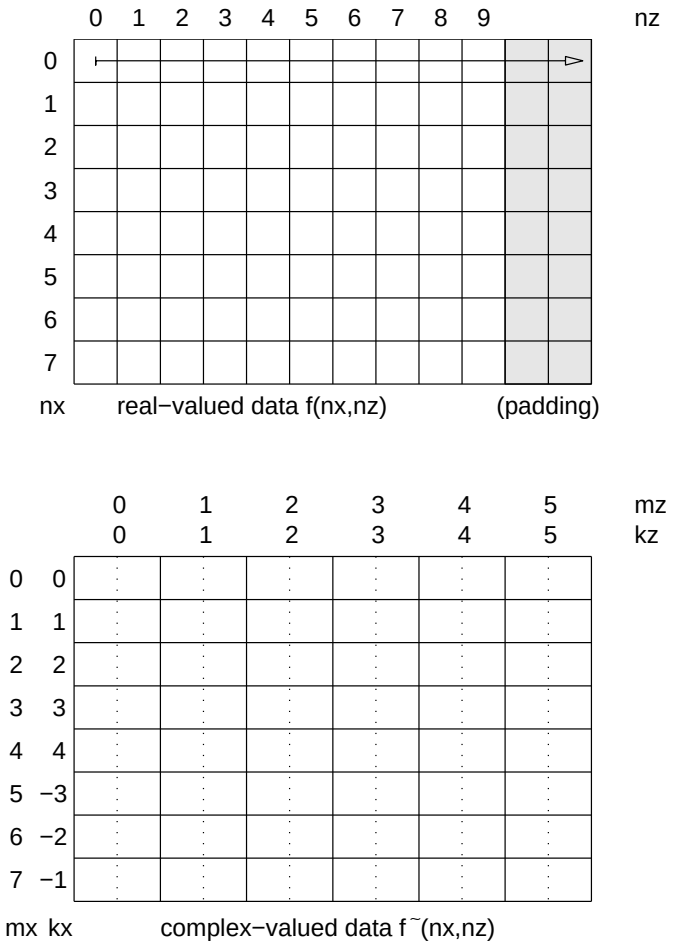

Figure 2: **Layout of data in memory for real-to-complex xz-Fourier transforms** for the case $N_x = 8$, $N_z = 10$. The Fourier transform converts real-valued data, above, to complex-valued data, below. Each solid box in the upper picture is a double-precision real number. Each solid box in the picture below is a double-precision complex number, with real and imaginary parts separated by a dashed line. The arrow indicates row-major storage order: successive memory locations store data with successive `nz`.

`Real r = L2Norm2(f);`	$r = \frac{1}{L_x L_y L_z} \int_0^{L_x} \int_a^b \int_0^{L_z} \mathbf{f} \cdot \mathbf{f} \, dx \, dy \, dz$
`Real r = L2Norm(f);`	$r = \sqrt{\texttt{L2Norm2(f)}}$
`Real r = L2Dist2(f,g);`	$r = \texttt{L2Norm2}(\mathbf{f} - \mathbf{g})$
`Real r = L2Dist(f,g);`	$r = \texttt{L2Norm}(\mathbf{f} - \mathbf{g})$
`Real r = bcNorm2(f);`	$r = \frac{1}{L_x L_z} \int_0^{L_x} \int_0^{L_z} (f \cdot f\|_{y=a} + f \cdot f\|_{y=b}) \, dx \, dz$
`Real r = bcNorm(f);`	$r = \sqrt{\texttt{bcNorm2(f)}}$
`Real r = bcDist2(f,g);`	$r = \texttt{bcNorm2}(\mathbf{f} - \mathbf{g})$
`Real r = bcDist(f,g);`	$r = \texttt{bcNorm}(\mathbf{f} - \mathbf{g})$
`Real r = divNorm2(f);`	$r = \texttt{L2Norm2}(\nabla \cdot \mathbf{f})$
`Real r = divNorm(f);`	$r = \sqrt{\texttt{divNorm2(f)}}$
`Real r = divL2Dist2(f,g);`	$r = \texttt{divNorm2}(\mathbf{f} - \mathbf{g})$
`Real r = divL2Dist(f);`	$r = \texttt{divNorm}(\mathbf{f} - \mathbf{g})$

Table 3: FlowField differential operators

with two columns for padding. The bottom picture shows the same memory after the Fourier transform, now interpreted as an $M_x \times M_z = N_x \times (N_z/2 + 1)$ array of complex numbers. In the bottom picture note the correspondence between the mode-number array index m_x and the wavenumber k_x, and the reduced range of the m_z array index. The Fourier coefficients with negative k_z are defined implicitly by $\widetilde{f}_{k_x,-k_z} = \widetilde{f}^*_{-k_x,k_z}$.

3.5 DNS and related classes

A DNS object advances a pair of velocity and pressure FlowFields forward in time, according to the Navier-Stokes. This section describes how to use the DNS class. For its mathematical details, see Section 4.

DNS objects are constructed by

```
DNS dns(u, U, nu, dt, flags);
```

or

```
DNS dns(u, U, nu, dt, flags, T0);
```

Of the arguments, `u` is a FlowField representing the initial condition of the fluctuating velocity, `U` is a ChebyCoeff representing the base flow, `dt` is either a Real number or a TimeStep object representing the finite-difference time step, and `flags` is a DNSFlags object. The optional `T0` argument is a real number that specifies the starting time or the integration.

3.5.1 Configuring DNS with DNSFlags

The DNSFlags class is used to configure some optional generalizations of CHQZ's algorithm. DNSFlags contains several flag variables which can be set at construction or assigned afterwards. For example,

```
DNSFlags flags(BulkVelocity, CNAB2, Rotational, DealiasXZ, PrintTime);
```

or

```
DNSFlags flags; // set to default values
flags.constraint   = PressureGradient;
flags.timestepping = CNAB2;
```

The complete set of DNSFlags variables and their allowed values are

DNSFlags variable	allowed values (default first)
`flags.constraint`	`BulkVelocity, PressureGradient`
`flags.timestepping`	`CNFE1, CNAB2, CNRK2, SMRK2, SBDF1, ..., SBDF4`
`flags.initstepping`	`CNFE1, CNRK2, SMRK2`
`flags.nonlinearity`	`Rotational, SkewSymmetric, Alternating, Linearized`
`flags.dealiasing`	`DealiasXZ, NoDealiasing, DealiasY, DealiasXYZ`
`flags.verbosity`	`PrintTime, Silent, VerifyTauSolve, PrintAll`

The basic meanings of the DNSFlags variables are

- `flags.constraint` : Periodic channel flows satisfy the Navier-Stokes equations with either the bulk velocity or the spatial-mean pressure gradient set as an external constraint. This flag sets which constraint is to be enforced. DNS's default behavior determines the spatial-mean pressure gradient or bulk velocity from the fluctuation's initial condition `u` and matches this as a fixed constraint at each time step. DNS can match time-varying constraints as well. See Section 3.5.3 for further details.
- `flags.timestepping`: The DNS class implements seven different time-stepping algorithms. (The default is SBDF3.)
  - `CNFE1` or `SBDF1`: 1st-order Crank-Nicolson, Foward-Euler or 1st-order Semi-implicit Backward Differentiation Formula –two names for the same algorithm. This algorithms is extremely simple and needs no initialization need, but its 1st-order error scaling makes it practically worthless, except for initializing other algorithms.
  - `CNAB2` 2nd-order Crank-Nicolson, Adams-Bashforth. A popular algorithm, but higher-frequency modes are poorly damped (see [**?**]) Requires one initialization step. Zang warns against using `CNAB2` in combination with Rotational nonlinearity unless the high-frequency modes are dealiased [**?**]. `CNAB2` enforces zero-divergence at successive timesteps and momentum equations halfway between successive time steps, which can lead to slowly decaying period-$2dt$ oscillation in the pressure field, unless pressure and velocity are initialized accurately.
  - `CNRK2`: a three-substep, 2nd-order semi-implicit Crank-Nicolson, Runge-Kutta algorithm, developed by Zang and Hussaini [**?**] and but implemented in Channelflow from the Peyret's exposition [**?**]. According to Peyret, Zang and Hussaini observed 3rd-order scaling for this algorithm applied to low-viscosity flows, even though it is theoretically 2nd-order. Numerical tests in Channelflow show 2nd-order scaling for velocity fields at $Re = 10^3 - 10^4$, and *1st-order* scaling for pressure, due to a phase error in the pressure field. `CNRK2` requires no initialization.
  - `SMRK2`: a three-substep, 2nd-order semi-implicit Runge-Kutta developed by Spalart, Moser, and Rogers [**?**]. Identical characteristics as `CNRK2`, including observed 2nd-order scaling consistent with theory, contrary to authors' claim of 3rd-order scaling, and 1st-order phase error in pressure. Requires no initialization.

- `SBDF2, SBDF3, SBDF4`: 2nd, 3rd, and 4th-order Semi-implicit Backward Differentiation Formulae, requiring 1,2, and 3 initialization steps. I have found the `SBDF` schemes to be the best-behaved of the lot. When solving u^{n+1} and p^{n+1}, `SBDF` schemes enforce divergence and momentum equations at t_{n+1}. This strongly implicit formulation poduces strong damping for high-frequency modes and results in pressure field as accurate as the velocity field. `SBDF3` is particularly good: it has the strongest asympotitc decay of all 3rd-order implicit-explicit linear multistep schemes. For these reasons, `SBDF3` is the default value of `flags.timestepping`. Peyret terms these algorithms AB/BDEk (kth-order Adams-Bashforth Backward-Differentiation).

- `flags.initstepping`: Some of the time-stepping algorithms listed above (`SBDF` in particular) require data from N previous time steps. Supplying these past values to the DNS constructor would entail a number of tedious practical problems, so the DNS class instead takes its first N steps with an *initialization timestepping* algorithm that requires no previous data. This initialization algorithm is specified by `flags.initstepping`. Valid values are `CNFE`, `CNRK2`, and `SMRK2`. The default is `CNRK2`.

- `flags.nonlinearity`: The nonlinear term in the Navier-Stokes calculation can be computed in a number of forms that are equivalent in continuous mathematics but slightly different when computed with spectral expansions and collocation. The default is `SkewSymmetric`.
  - `Rotational`: Fast but generates high-frequency errors unless dealiased (see [**?**]).
  - `SkewSymmetric`: Comparatively expensive to compute (factor of two (?) compared to `Rotational` but
  - `Convective`.
  - `Divergence`.
  - `Alternating` convection/divergence an alternating time steps. A cheap approximation to `SkewSymmetric`, which is an average of the convective and divergence forms. I have not yet analyzed how the alternating nonlinearity method interacts with multistepping algorithms.
  - `Linearized` about the base flow.

- `flags.dealiasing`: Nonlinear terms are calculated with collocation methods. FlowFields can be padded with zeros to eliminate aliasing errors. `DealiasXZ` causes 2/3-style padding in xz: at each time-step the upper 1/3 of x and z of the velocity field's Fourier coefficients are set to zero. Eventually, `DealiasY` will cause 3/2-style padding in y, with collocation calculations performed in temporary arrays of length $3N_y/2$ –but that is not yet implemented! See [**?**] and Section 4.4 for more details.

- `flags.verbosity`: This flag governs what the DNS prints at each timestep. `PrintTime` prints the integration time at each timestep, which is helpful when running Channelflow programs interactively. `VerifyTauSolve` prints a verbose and expensive verification of the tau-equation solutions for each Fourier mode. Other values are self-explanatory.

For precise specification of how the DNSFlags configuration variables affect the integration, please refer to Section 4.

3.5.2 Base-fluctuation decomposition

DNS decomposes the velocity and pressure fields into *base* and *fluctuating* parts in the form

$$\mathbf{u}_{\text{tot}}(\mathbf{x},t) = U(y)\mathbf{e}_x + \mathbf{u}(\mathbf{x},t) \tag{21}$$

$$p_{\text{tot}}(\mathbf{x},t) = x\frac{dP}{dx}(t) + p(\mathbf{x},t) \tag{22}$$

Channelflow represents the fluctuating velocity and pressure fields $\mathbf{u}$ and p with xz-periodic FlowFields. Hence in the decomposition of the pressure gradient,

$$\nabla p_{\text{tot}}(\mathbf{x}, t) = \frac{dP}{dx}(t)\mathbf{e}_x + \nabla p(\mathbf{x}, t), \tag{23}$$

the fluctuating pressure gradient ∇p has a zero spatial mean, and all of the spatial-mean pressure gradient is carried by the base pressure gradient. For simplicity, dP/dx is referred to as the mean pressure gradient in subsequent material, with spatial-mean implied. DNS imposes no further restriction on the base flow $U(y)$ or the base pressure gradient dP/dx: they do not have to solve the Navier-Stokes equations as a pair, nor is $\mathbf{u}$ required to have zero spatial mean. The base flow $U(y)$ for a simulation is set through the ChebyCoeff `U` argument to the DNS constructor.

3.5.3 Enforcing bulk velocity or mean pressure constraints

A channel flow can satisfy either an externally imposed bulk velocity, or an externally imposed mean pressure gradient. When one of is enforced as a constraint, the other is a dependent variable whose value is determined from the momentum equation. The DNS class allows either type of constraint, as specified by its DNSFlags argument. By default, the DNS determines the value of the constraint from the initial data and matches that value at all future times. For bulk velocity, the initial value is determined by

$$U_{\text{bulk}} = \frac{1}{L_x L_z (b-a)} \int_0^{L_x} \int_a^b \int_0^{L_z} U(y) + u(\mathbf{x}, 0)\, dx\, dy\, dz \tag{24}$$

The initial mean pressure gradient is set from the initial wall-shear, according to

$$\frac{dP}{dx} = \frac{\nu}{b-a}\left(\frac{du_{\text{mean}}}{dy}\bigg|_b - \frac{du_{\text{mean}}}{dy}\bigg|_a \right) \tag{25}$$

Check correctness of eqn, use of ν vs μ, and ρ. Note that this choice is somewhat arbitrary –it assumes the net acceleration of the fluid is zero.

The DNS class allows the initial constraint values to be reset. For example,

```
DNS dns(u,U,nu,dt,flags);
dns.reset_dPdx(0.0);
```

resets the mean pressure constraint to zero and sets the constraint type to `PressureGradient`.

```
DNS dns(u,U,nu,dt,flags);
dns.reset_Ubulk(0.0);
```

resets the bulk velocity constraint to zero and sets the constraint type to `BulkVelocity`.

3.5.4 Fixed and variable time-stepping

Fixed time-stepping

In the simplest case, a DNS performs fixed time-stepping and enforces a constant bulk velocity or mean pressure gradient

```
Real dt=0.10;
DNSFlags flags(BulkVelocity, RK3, Rotational, DealiasXZ, PrintTime);
DNS dns(u, U, nu, dt, flags);
```

```
for (int n=0; n<N; ++n)
   dns.advance(u, q);
```

The loop advances the fluctuating velocity `u` and modified pressure `q` `N` steps of length `dt`. The `advance()` function can also take multiple steps internally, for example,

```
int m = 10;
for (int n=0; n<N; ++n)
   dns.advance(u, q, m);
```

advances `u` and `q` a total of `N*m` steps of length `dt`. The integration time can be determined at any point by calling `Real t = dns.advance(u, q, m);`.

Variable time-stepping

Variable time-stepping minimizes the computational cost of an integration by maximizing the timestep while keeping the CFL number near a threshold. The optional TimeStep class automates some of the issues associated with variable timesteps. TimeStep tries to maximize the CFL number subject to the constraints that (1) the timestep stays in a given range, (2) the CFL number stays in a given range, and (3) the timestep is a whole-number fraction of a fixed time-interval. The last constraint allows one to stop and examine integrations at fixed time-intervals. For example,

```
TimeStep dt(dtstart, dtmin, dtmax, dT, CFLmin, CFLmax);
DNS dns(u, U, nu, dt, flags);

for (Real t=0; t<T0; t += dT) {
  dns.advance(u, q, dt.n());

  if (dt.adjust(dns.CFL()))
    dns.reset(nu, dt);
}
```

In this example, the TimeStep object adjusts itself to keep the CFL number between `CFLmax` and over `CFLmin`, `dt` between `dtmin` and `dtmax`, and `dt` a whole-number fraction of `dT`, so that `dt*dt.n() = dT` and each pass through the for-loop then covers the same time-interval. If the CFL number goes out of range, `dt.adjust` changes the value of the time step and returns `true`, and the `dns` object is reset to compute with the new integration timestep. Resetting the DNS's timestep is a moderately expensive operation (about the same as advancing one timestep), so it should be done infrequently.

Note that, when using `CNAB2` or `SBDF` (multistep) time-stepping algorithms, resetting the time step requires reinitializing the multistep algorithm. In this case, the DNS object reinitialize by taking several steps with the `flags.initstepping` algorithm.

CFL number. Measure expense of dns.reset().

Time-varying constraints

The following code enforces a time-varying bulk velocity.

```
DNSFlags flags;
flags.constraint = BulkVelocity;
DNS dns(u, U, nu, dt, flags);
for (Real t=0; t<T0; t += dt) {
  Real ubulk = sin(k*t);
  dns.advance(u, q, ubulk);
}
```

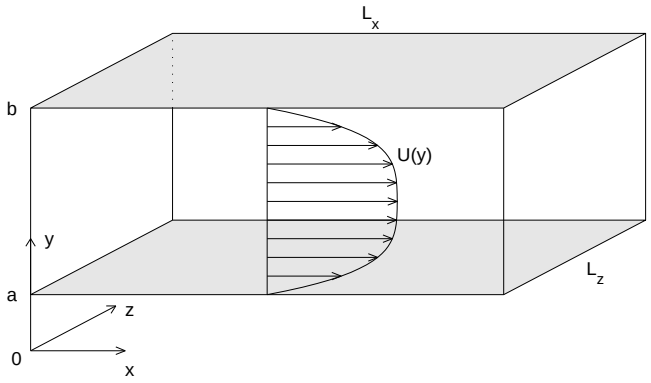

Figure 3: **Schematic of channel flow.** Fluid flows between two rigid walls at $y = a$ and $y = b$. The boundary conditions are periodic in x and z and no-slip at the walls. The mean flow $U(y)$ is driven in the x-direction by a mean pressure gradient.

To enforce a time-varying constraint on the pressure gradient, change the first DNSFlags argument to `PressureGradient` and rename `ubulk` to `dPdx` for clarity.

Note that time-varying constraints require CNAB2 time-stepping. I haven't yet figured out how to enforce the constraints properly in RK3 substeps. Note also that the `advance` function distinguishes variable-constraint timestepping from multistep time-stepping (Section 3.5.4 by the type of the third argument. If you write

```
Real m = 10;              // note the Real type
for (int n=0; n<N; ++n)
   dns.advance(u, q, m); // enforce Ubulk or dPdx to 10!
```

`advance()` will interpret the `m` argument as a time-varying constraint to be enforced!

4 Mathematical details

This section discusses in some detail the mathematics of the spectral Channelflow algorithm, in order to specify the consequences of configuration choices and to provide a point of reference for comments in the source code.

4.1 The Navier-Stokes equations

Consider an incompressible wall-bounded fluid flow in a rectangular domain $\Omega \triangleq L_x\mathbb{T} \times [a, b] \times L_z\mathbb{T}$, where $\mathbb{T}$ is the periodic unit interval. The fluid flow in Ω is governed by the incompressible Navier-Stokes equations,

$$\frac{\partial \mathbf{u}_{\text{tot}}}{\partial t} + \mathbf{u}_{\text{tot}} \cdot \nabla \mathbf{u}_{\text{tot}} = -\nabla p_{\text{tot}} + \nu \nabla^2 \mathbf{u}_{\text{tot}}, \tag{26}$$

$$\nabla \cdot \mathbf{u}_{\text{tot}} = 0, \tag{27}$$

a where $\mathbf{u}_{\text{tot}}(\mathbf{x}, t)$ is the total fluid velocity field and $p_{\text{tot}}(\mathbf{x}, t)$ is the total pressure field. The upper and lower surfaces of Ω are rigid walls, giving rise to no-slip boundary conditions: $\mathbf{u} = 0$ at $y = a$ and $y = b$.

The boundary conditions in the x and z directions are periodic: $\mathbf{u}_{\text{tot}}(x + L_x, y, z, t) = \mathbf{u}_{\text{tot}}(x, y, z, t)$ and $\mathbf{u}_{\text{tot}}(x, y, z + L_z, t) = \mathbf{u}_{\text{tot}}(x, y, z, t)$.[1]

4.2 Base-fluctuation decomposition

Channelflow allows the total velocity and pressure fields to be broken into constant and fluctuating parts. The velocity field is the sum of the *base velocity* or *base flow* $U(y)\mathbf{e}_x$, and the *fluctuating velocity* $\mathbf{u}(\mathbf{x}, t)$.

$$\mathbf{u}_{\text{tot}}(\mathbf{x}, t) = U(y)\, \mathbf{e}_x + \mathbf{u}(\mathbf{x}, t). \tag{28}$$

The total pressure field is the sum of a linear-in-x term $\Pi_x(t)\, x$ and a periodic fluctuating pressure $p(\mathbf{x}, t)$. The gradient of this decomposition relates the total pressure gradient to a spatially-constant *base pressure gradient* $\Pi_x \mathbf{e}_x$ and a *fluctuating pressure gradient* $\nabla p(\mathbf{x}, t)$.

$$p_{\text{tot}}(\mathbf{x}, t) = \Pi_x(t)\, x + p(\mathbf{x}, t) \tag{29}$$

$$\nabla p_{\text{tot}}(\mathbf{x}, t) = \Pi_x(t)\, \mathbf{e}_x + \nabla p(\mathbf{x}, t) \tag{30}$$

These forms for the base flow and pressure gradient are general enough to represent cases like Poisseuille, Couette, and turbulent mean profiles. Note that Channelflow does not require the base velocity and base pressure gradient to satisfy the Navier-Stokes equations themselves. Substituting eqns. 28 and 30 into eqn. 26 gives

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla p = \nu \nabla^2 \mathbf{u} - \mathbf{u}_{\text{tot}} \cdot \nabla \mathbf{u}_{\text{tot}} + \left[\nu \frac{\partial^2 U}{\partial y^2} - \Pi_x \right] \mathbf{e}_x \tag{31}$$

4.3 Forms for the nonlinear term

There are several different forms for the term of form $\mathbf{u}_{\text{tot}} \cdot \nabla \mathbf{u}_{\text{tot}}$ in eqn. 31 that are identical in continuous mathematics but have different properties when discretized. These are

$$\textit{the convection form} \qquad \mathbf{u}_{\text{tot}} \cdot \nabla \mathbf{u}_{\text{tot}} \tag{32}$$

$$\textit{the divergence form} \qquad \nabla \cdot (\mathbf{u}_{\text{tot}} \mathbf{u}_{\text{tot}}) \tag{33}$$

$$\textit{the skew-symmetric form} \qquad \frac{1}{2} \mathbf{u}_{\text{tot}} \cdot \nabla \mathbf{u}_{\text{tot}} + \frac{1}{2} \nabla \cdot (\mathbf{u}_{\text{tot}} \mathbf{u}_{\text{tot}}) \tag{34}$$

$$\textit{the rotational form} \qquad (\nabla \times \mathbf{u}_{\text{tot}}) \times \mathbf{u}_{\text{tot}} + \frac{1}{2} \nabla (\mathbf{u}_{\text{tot}} \cdot \mathbf{u}_{\text{tot}}) \tag{35}$$

These expressions are identically equal, assuming $\nabla \cdot \mathbf{u}_{\text{tot}} = 0$. When discretized, the rotational form is the least expensive to compute, but it introduces errors in the high spatial frequencies unless dealiased transforms are used. The skew-symmetric form produces no such errors but is roughly twice as expensive to compute. Note that the skew-symmetric form is the average of the convection and divergence forms. One can simulate this averaging by alternating between the convection and divergence forms on successive timesteps. In practice the alternating method is as well-behaved as the skewsymmetric and almost as fast as the rotational. Zang recommends using the skew-symmetric or alternating forms with aliased transforms or the rotational form with dealiased transforms. See Zang ([**?**]) for further details. Channelflow implements the rotational, convection, divergence, skew-symmetric, and alternating forms. The form is chosen by setting the DNSFlags `nonlinearity` variable –see Section 3.5.1.

[1] Components of vector variables are written several ways: $\mathbf{x} = (x, y, z)$ or $\mathbf{x} = (x_0, x_1, x_2)$, and $\mathbf{u} = (u, v, w)$ or (u_0, u_1, u_2). A unit vector in the x (or x_0) direction is $\mathbf{e}_x$ (or $\mathbf{e}_0$).

For historical reasons, the Channelflow does not compute the rotational form exactly as shown above; rather, the nonlinear term is first expanded with the base-fluctuation decomposition and then the rotational form is applied to $\mathbf{u} \cdot \nabla \mathbf{u}$:

$$\mathbf{u}_{\text{tot}} \cdot \nabla \mathbf{u}_{\text{tot}} = U \frac{\partial \mathbf{u}}{\partial x} + v \frac{\partial U}{\partial y} \mathbf{e}_x + \mathbf{u} \cdot \nabla \mathbf{u} \tag{36}$$

$$= U \frac{\partial \mathbf{u}}{\partial x} + v \frac{\partial U}{\partial y} \mathbf{e}_x + (\nabla \times \mathbf{u}) \times \mathbf{u} + \frac{1}{2} \nabla (\mathbf{u} \cdot \mathbf{u}) \tag{37}$$

DNS computes the nonlinear term according to the value of `flags.nonlinearity`. Here we list the form of the Navier-Stokes equation solved by DNS with various values of the flags (with $\mathbf{u}_{\text{tot}} = \mathbf{u} + U\mathbf{e}_x$ and U fixed).

Rotational:

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \left[p + \frac{1}{2} \mathbf{u} \cdot \mathbf{u} \right] = \nu \nabla^2 \mathbf{u} - \left[(\nabla \times \mathbf{u}) \times \mathbf{u} + U \frac{\partial \mathbf{u}}{\partial x} + v \frac{\partial U}{\partial y} \mathbf{e}_x \right] + \left[\nu \frac{\partial^2 U}{\partial y^2} - \Pi_x \right] \mathbf{e}_x \tag{38}$$

Convection:

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla p = \nu \nabla^2 \mathbf{u} - \mathbf{u}_{\text{tot}} \cdot \nabla \mathbf{u}_{\text{tot}} + \left[\nu \frac{\partial^2 U}{\partial y^2} - \Pi_x \right] \mathbf{e}_x \tag{39}$$

Divergence:

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla p = \nu \nabla^2 \mathbf{u} - \nabla (\mathbf{u}_{\text{tot}} \cdot \mathbf{u}_{\text{tot}}) + \left[\nu \frac{\partial^2 U}{\partial y^2} - \Pi_x \right] \mathbf{e}_x \tag{40}$$

Skew-symmetric:

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla p = \nu \nabla^2 \mathbf{u} - \left[\frac{1}{2} \mathbf{u}_{\text{tot}} \cdot \nabla \mathbf{u}_{\text{tot}} + \frac{1}{2} \nabla (\mathbf{u}_{\text{tot}} \cdot \mathbf{u}_{\text{tot}}) \right] + \left[\nu \frac{\partial^2 U}{\partial y^2} - \Pi_x \right] \mathbf{e}_x \tag{41}$$

Linearized:

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla p = \nu \nabla^2 \mathbf{u} - \left[U \frac{\partial \mathbf{u}}{\partial x} + v \frac{\partial U}{\partial y} \mathbf{e}_x \right] + \left[\nu \frac{\partial^2 U}{\partial y^2} - \Pi_x \right] \mathbf{e}_x \tag{42}$$

Alternating: eqns. 39 and 40 on alternating time steps.

Eqns. 41-42 can be reunited with notation. With U fixed and $\mathbf{u}_{\text{tot}}$ defined as $\mathbf{u} + U\mathbf{e}_x$, define the *nonlinear term* $\mathbf{N}(\mathbf{u})$ by

$$\mathbf{N}(\mathbf{u}) \triangleq \begin{cases} (\nabla \times \mathbf{u}) \times \mathbf{u} + U \frac{\partial \mathbf{u}}{\partial x} + v \frac{\partial U}{\partial y} \mathbf{e}_x & \text{Rotational} \\ \mathbf{u}_{\text{tot}} \cdot \nabla \mathbf{u}_{\text{tot}} & \text{Convection} \\ \frac{1}{2} \nabla (\mathbf{u}_{\text{tot}} \cdot \mathbf{u}_{\text{tot}}) & \text{Divergence} \\ \frac{1}{2} \mathbf{u}_{\text{tot}} \cdot \nabla \mathbf{u}_{\text{tot}} + \frac{1}{2} \nabla (\mathbf{u}_{\text{tot}} \cdot \mathbf{u}_{\text{tot}}) & \text{Skew-symmetric} \\ U \frac{\partial \mathbf{u}}{\partial x} + v \frac{\partial U}{\partial y} \mathbf{e}_x & \text{Linearized} \end{cases} \tag{43}$$

and the *modified pressure* q by

$$q \triangleq \begin{cases} p + \frac{1}{2} \mathbf{u} \cdot \mathbf{u} & \text{Rotational} \\ p & \text{else} \end{cases} \tag{44}$$

Define also the *linear term* $\mathbf{L}(\mathbf{u})$ and the *constant term* $\mathbf{C}$ by

$$\mathbf{L}\mathbf{u} \triangleq \nu\nabla^2\mathbf{u} \tag{45}$$

$$\mathbf{C} \triangleq \left[\nu\frac{\partial^2 U}{\partial y^2} - \Pi_x\right]\mathbf{e}_x \tag{46}$$

Note that the constant term is constant in $\mathbf{u}$, but it may vary in time, since it contains the mean pressure gradient, which is a potentially time-varying external forcing parameter. With these definitions eqns. 41 and 38 can be written

$$\frac{\partial\mathbf{u}}{\partial t} + \nabla q = \mathbf{L}\mathbf{u} - \mathbf{N}(\mathbf{u}) + \mathbf{C} \tag{47}$$

The DNS `advance(u,q)` function advances the FlowFields `u` and `q` to their (approximate) values at the next time step, according to eqn. 47 and the constraint $\nabla\cdot\mathbf{u} = 0$. Note that the meaning of the returned value of `q` depends on the choice of nonlinearity, according to eqn. 44.

The next step in the derivation is to Fourier-transform eqn. 31. We apply the continuous Fourier transform (eqn. 17) since eqn. 31 is continuous and introduce truncation later. The Fourier-transformed operators for the gradient, the Laplacian, and the linear operator $\mathbf{L}$ are

$$\tilde{\nabla}_{k_x k_z} \triangleq 2\pi i\frac{k_x}{L_x}\mathbf{e}_x + \frac{\partial}{\partial y}\mathbf{e}_y + 2\pi i\frac{k_z}{L_z}\mathbf{e}_z, \tag{48}$$

$$\tilde{\nabla}^2_{k_x k_z} \triangleq \frac{\partial^2}{\partial y^2} - 4\pi^2\left(\frac{k_x^2}{L_x^2} + \frac{k_z^2}{L_z^2}\right), \tag{49}$$

$$\tilde{\mathbf{L}}_{k_x k_z} \triangleq \nu\tilde{\nabla}^2_{k_x k_z} \tag{50}$$

With these definitions, $\widetilde{\nabla q} = \tilde{\nabla}\tilde{q}$ and $\widetilde{\mathbf{L}\mathbf{u}} = \tilde{\mathbf{L}}\tilde{\mathbf{u}}$. Here and onwards $k_x k_z$ subscripts will often be suppressed, to reduce clutter. The Fourier transform of eqn. 31 can then be written

$$\frac{\partial\tilde{\mathbf{u}}}{\partial t} + \tilde{\nabla}\tilde{q} = \tilde{\mathbf{L}}\tilde{\mathbf{u}} - \widetilde{\mathbf{N}(\mathbf{u})} + \tilde{\mathbf{C}} \tag{51}$$

Note that since $\mathbf{C}$ is spatially constant, so $\tilde{\mathbf{C}} = \mathbf{C}\,\delta_{k_x 0}\,\delta_{k_z 0}$.

4.4 Time-stepping algorithms

DNS currently offers two time-integration schemes: CNAB2, a mixed Crank-Nicolson/Adams-Bashforth scheme, and RK3, a mixed 3rd-order Runge-Kutta scheme. Both schemes treat the linear term implicitly and the nonlinear term explicitly. CNAB is simpler so let's begin there. Let $\tilde{\mathbf{u}}^n$ be the approximation of $\tilde{\mathbf{u}}$ at time $t = n\Delta t$, and let $\tilde{\mathbf{N}}^n \triangleq \widetilde{\mathbf{N}(\mathbf{u}^n)}$. Then we approximate terms in eqn. 51 at $t = (n-1/2)\Delta t$ with

$$\frac{\partial}{\partial t}\tilde{\mathbf{u}}^{n+1/2} = \frac{\tilde{\mathbf{u}}^{n+1} - \tilde{\mathbf{u}}^n}{\Delta t} + O(\Delta t^2) \tag{52}$$

$$\tilde{\mathbf{L}}\tilde{\mathbf{u}}^{n+1/2} = \frac{1}{2}\tilde{\mathbf{L}}\tilde{\mathbf{u}}^{n+1} + \frac{1}{2}\tilde{\mathbf{L}}\tilde{\mathbf{u}}^n + O(\Delta t^2) \tag{53}$$

$$\tilde{\nabla}\tilde{q}^{n+1/2} = \frac{1}{2}\tilde{\nabla}\tilde{q}^{n+1} + \frac{1}{2}\tilde{\nabla}\tilde{q}^n + O(\Delta t^2) \tag{54}$$

$$\tilde{\mathbf{N}}^{n+1/2} = \frac{3}{2}\tilde{\mathbf{N}}^n \quad - \frac{1}{2}\tilde{\mathbf{N}}^{n-1} + O(\Delta t^2) \tag{55}$$

$$\tilde{\mathbf{C}}^{n+1/2} = \frac{1}{2}\tilde{\mathbf{C}}^{n+1} \; + \frac{1}{2}\tilde{\mathbf{C}}^n + O(\Delta t^2) \tag{56}$$

Table 4: **Time-stepping coefficients**

	i	α_i	β_i	γ_i	ζ_i
CNAB	0	1/2	1/2	3/2	-1/2
RK3	0	29/96	37/160	8/15	0
	1	-3/40	5/24	5/12	-17/60
	2	1/6	1/6	3/4	-5/12

The time-derivative approximation is obvious, the approximation for the linear term is called Crank-Nicolson, and that of the nonlinear term is Adams-Bashforth (see CHQZ section 4.3). Plugging those into eqn. 51 and rearranging gives

$$\left[\frac{1}{\Delta t} - \frac{1}{2}\tilde{\mathbf{L}}\right]\tilde{\mathbf{u}}^{n+1} + \frac{1}{2}\tilde{\nabla}q^{n+1} = \left[\frac{1}{\Delta t} + \frac{1}{2}\tilde{\mathbf{L}}\right]\tilde{\mathbf{u}}^{n} - \frac{1}{2}\tilde{\nabla}q^{n} + \frac{3}{2}\tilde{\mathbf{N}}^{n} - \frac{1}{2}\tilde{\mathbf{N}}^{n-1} + \frac{1}{2}\tilde{\mathbf{C}}^{n+1} + \frac{1}{2}\tilde{\mathbf{C}}^{n} \quad (57)$$

At this point we drop the $O(\Delta t^2)$ notation and take eqn. 57 as an update rule for an approximate solution $\tilde{\mathbf{u}}^{n+1}$. Eqn. 57 has several notable properties: (1) it is linear in the unknowns $\tilde{\mathbf{u}}^{n+1}$ and $\tilde{q}^{n+1}$, (2) its right-hand side can be computed directly from velocity and pressure fields at previous time-steps and the external mean-pressure parameter, and (3) the linear equations for each Fourier mode $k_x k_z$ are independent.

Channelflow's 3rd-order Runge-Kutta scheme, based on [**?**], is similar in principle but involves three substeps for each timestep of length Δt, with different coefficients α_i, β_i, γ_i, and ζ_i for each substep.

$$\left[\frac{1}{\Delta t} - \beta_i\tilde{\mathbf{L}}\right]\tilde{\mathbf{u}}^{n,i+1} + \beta_i\tilde{\nabla}\tilde{q}^{n,i+1}$$
$$= \left[\frac{1}{\Delta t} + \alpha_i\tilde{\mathbf{L}}\right]\tilde{\mathbf{u}}^{n,i} - \alpha_i\tilde{\nabla}\tilde{q}^{n} + \gamma_i\tilde{\mathbf{N}}^{n,i} + \zeta_i\tilde{\mathbf{N}}^{n,i-1} + \beta_i\tilde{\mathbf{C}}^{n+1} + \alpha_i\tilde{\mathbf{C}}^{n} \quad (58)$$

The second superscript indicates the Runge-Kutta substeps. For example, a three-substep follows the sequence $\tilde{\mathbf{u}}^{n,0}, \tilde{\mathbf{u}}^{n,1}, \tilde{\mathbf{u}}^{n,2}, \tilde{\mathbf{u}}^{n+1,0}$. RK3 is a particularly convenient time-stepping scheme because $\zeta_0 = 0$ eliminates the previous-step nonlinear term $\tilde{\mathbf{N}}^{n,i-1}$ when $i = 0$. Consequently the time-stepping can be started from a single instantaneous velocity field. For CNAB, both $\tilde{\mathbf{N}}^{n}$ and $\tilde{\mathbf{N}}^{n-1}$ are always required, so two consecutive velocity fields are needed for starting the time-stepping. The CNAB time-stepping algorithm can also be expressed in a form like eqn. 58, so we'll proceed using this as the general form.

Expanding $\tilde{\mathbf{L}}$ on the left-hand side of eqn. 58 results in an equation of the form

$$\nu\tilde{\mathbf{u}}''^{\,n,i+1} - \lambda\tilde{\mathbf{u}}^{\,n,i+1} - \tilde{\nabla}\tilde{q}^{n,i+1} = -\tilde{\mathbf{R}}^{n,i} \quad (59)$$

where

$$\lambda \triangleq \frac{1}{\beta_i\Delta t} + 4\pi^2\nu\left(\frac{k_x^2}{L_x^2} + \frac{k_z^2}{L_z^2}\right) \quad (60)$$

$$\tilde{\mathbf{R}}^{n,i} \triangleq \left[\frac{1}{\beta_i\Delta t} + \frac{\alpha_i}{\beta_i}\tilde{\mathbf{L}}\right]\tilde{\mathbf{u}}^{n,i} + \frac{\alpha_i}{\beta_i}\tilde{\nabla}\tilde{q}^{n,i} + \frac{\gamma_i}{\beta_i}\tilde{\mathbf{N}}^{n,i} + \frac{\zeta_i}{\beta_i}\tilde{\mathbf{N}}^{n,i-1} + \tilde{\mathbf{C}}^{n,i+1} + \frac{\alpha_i}{\beta_i}\tilde{\mathbf{C}}^{n,i} \quad (61)$$

$$\tilde{\mathbf{u}}'' \triangleq \frac{d^2}{dy^2}\tilde{\mathbf{u}} \quad (62)$$

Thus, at each timestep or substep, we need to solve eqn. 59 for each Fourier mode. The complete system of equations to be solved is

$$\nu\tilde{\mathbf{u}}'' - \lambda\tilde{\mathbf{u}} - \tilde{\nabla}\tilde{q} = -\tilde{\mathbf{R}} \tag{63}$$

$$\tilde{\nabla}\cdot\tilde{\mathbf{u}} = 0 \tag{64}$$

$$\tilde{\mathbf{u}}(a) = \tilde{\mathbf{u}} = 0 \tag{65}$$

From here on the time superscripts are suppressed. For lack of a better term, we call eqns. 63–65 the *tau equations*. The name derives from the need to add a *tau correction* to the solution of the equations in their discretized form. See CHQZ Section 7.3.1 and Section 4.5.2.

The bulk of the DNS `advance()` method is concerned with looping over the Fourier modes and calculating $\tilde{\mathbf{R}}$ in preparation for solving the tau equations. The actual solution is computed by TauSolver and related classes, discussed in Section 4.5. If xz-dealiasing is set in the DNSFlags, this loop excludes the highest one-third of Fourier modes and sets those modes to zero.

4.5 TauSolver

The TauSolver class solves equations of the form of eqns. 63–65. An DNS object contains an $N_x \times (N_z/2+1)$ array of TauSolvers, each one configured solving eqns. 63–65 for a given k_x, k_z pair. The TauSolver's solution method is as follows.

4.5.1 The influence-matrix method.

Eqns. 63–65 constitute three coupled differential equations in four unknowns $(\tilde{u}, \tilde{v}, \tilde{w}, \tilde{q})$, with one constraint equation and three boundary conditions. CHQZ, following Klieser and Schumann ([**?**]), show how to decompose these coupled equations into independent one-dimensional Helmholtz equations. For simplicity of presentation in this section we assume the walls are at $y = \pm 1$. We can isolate a system of equations in $\tilde{q}$ and $\tilde{v}$ by taking the divergence of eqn. 63, the $\tilde{v}$-component of the same, and evaluating $\tilde{\nabla}\cdot\tilde{\mathbf{u}} = 0$ at the two walls. This gives

$$\tilde{q}'' - \kappa^2\tilde{q} = -\tilde{\nabla}\cdot\tilde{\mathbf{R}} \qquad \tilde{v}'(\pm 1) = 0 \tag{66}$$

$$\nu\tilde{v}'' - \lambda\tilde{v} - \tilde{q}\,' = -\hat{R}_y \qquad \tilde{v}(\pm 1) = 0 \tag{67}$$

Eqns. 66 and 67 form a complete system for $\tilde{q}$ and $\tilde{v}$. Call this the *A-problem*. The A-problem is tricky to solve because $\tilde{q}$ appears in the $\tilde{v}$ differential equation while $\tilde{v}$ appears in the boundary condition.

To solve the A-problem, consider the inhomogeneous *B-problem*:

$$\tilde{q}'' - \kappa^2\tilde{q} = -\tilde{\nabla}\cdot\tilde{\mathbf{R}} \qquad \tilde{q}(\pm 1) = Q_\pm \tag{68}$$

$$\nu\tilde{v}'' - \lambda\tilde{v} - \tilde{q}\,' = -\hat{R}_y \qquad \tilde{v}(\pm 1) = 0 \tag{69}$$

The proper values $Q_\pm$ for the modified-pressure boundary conditions are unknown but will be determined from the requirement that $\tilde{v}'(a) = \tilde{v}'(b) = 0$. First let $(\tilde{q}_p, \tilde{v}_p)$ be the particular solution to the A-problem with homogeneous Dirichlet boundary conditions, i.e.

$$\tilde{q}_p'' - \kappa^2\tilde{q}_p = -\tilde{\nabla}\cdot\tilde{\mathbf{R}} \qquad \tilde{q}_p(\pm 1) = 0 \tag{70}$$

$$\nu\tilde{v}_p'' - \lambda\tilde{v}_p - \tilde{q}_p' = -\hat{R}_y \qquad \tilde{v}_p(\pm 1) = 0 \tag{71}$$

Next solve the B_+*-problem*,

$$\tilde{q}_+'' - \kappa^2 \tilde{q}_+ = 0 \qquad \tilde{q}_+(-1) = 0, \quad \tilde{q}_+(1) = 1 \tag{72}$$

$$\nu\tilde{v}_+'' - \lambda\tilde{v}_+ - \tilde{q}_+' = 0 \qquad \tilde{v}_+(\pm 1) = 0 \tag{73}$$

and the B_-*-problem*,

$$\tilde{q}_-'' - \kappa^2 \tilde{q}_- = 0 \qquad \tilde{q}_-(-1) = 1, \quad \tilde{q}_-(1) = 0 \tag{74}$$

$$\nu\tilde{v}_-'' - \lambda\tilde{v}_- - \tilde{q}_-' = 0 \qquad \tilde{v}_-(\pm 1) = 0 \tag{75}$$

Then the solution to the A-problem can be formed from the solutions to the particular A-problem and the homogeneous $B_\pm$-problems, by

$$\begin{pmatrix} \tilde{q} \\ \tilde{v} \end{pmatrix} = \begin{pmatrix} \tilde{q}_p \\ \tilde{v}_p \end{pmatrix} + \delta_+ \begin{pmatrix} \tilde{q}_+ \\ \tilde{v}_+ \end{pmatrix} + \delta_- \begin{pmatrix} \tilde{q}_- \\ \tilde{v}_- \end{pmatrix}, \tag{76}$$

The boundary conditions on $(\tilde{q}, \tilde{v})$ for the A-problem are satisfied if

$$\begin{bmatrix} \tilde{v}_+'(+1) & \tilde{v}_-'(+1) \\ \tilde{v}_+(-1) & \tilde{v}_-(-1) \end{bmatrix} \begin{pmatrix} \delta_+ \\ \delta_- \end{pmatrix} = - \begin{pmatrix} \tilde{v}_p(+1) \\ \tilde{v}_p(-1) \end{pmatrix} \tag{77}$$

Eqn. 77 is known as the *influence-matrix* equation. Solving it for $\delta_\pm$ produces the proper boundary conditions for the B-problem, and the consequent solution to the B-problem then satisfies the original A-problem. Alternatively, one can construct the solution to the A-problem directly from 76. Note that the $B_\pm$-problems are independent of the velocity and pressure fields, so their solutions can be precomputed and stored. This saves two complex Helmholtz computations per timestep for each Fourier mode. Channelflow takes this approach. An alternative is to determine boundary conditions $\tilde{q}(\pm 1) = Q_\pm$ from $\delta_\pm$ and eqn. 76, and use this to solve eqns. 68 and 69. This would save memory at the expense two complex Helmholtz solutions per timestep. In the future Channelflow might allow this as an option.

4.5.2 The tau correction

To be written.

4.6 HelmholtzSolver

The differential equations to be solved in Section 4.5 are all Helmholtz equations of the form

$$\nu u'' - \lambda u = f \qquad u(\pm 1) = u_\pm \tag{78}$$

where u is unknown, ν and λ are given parameters, and the right-hand-sides f and $u_\pm$ are given. The Chebyshev tau approximation of eqn. 78 is

$$\nu\hat{u}_n^{(2)} - \lambda\hat{u}_n = \widehat{f}_n \quad 0 \le n \le N-3 \tag{79}$$

$$\sum_{n=0}^{N-1} \hat{u}_n = u_+ \tag{80}$$

$$\sum_{n=0}^{N-1} (-1)^n \hat{u}_n = u_- \tag{81}$$

where $\hat{u}_n^{(2)}$, $\hat{u}_n$, and $\hat{f}_n$, are the Chebyshev expansion coefficients of u'', u, and f. CHQZ show how to express eqns. 79–81 as two independent banded tridiagonal matrix equations.

4.7 BandedTridiag

To be written.

5 Incidental classes

5.1 Real and Complex

Channelflow uses a tricks in `mathdefs.h` to simplify the declaration of double-precision floating point and complex floating-point numbers. These are

```
typedef double Real;
typedef std::complex<double> Complex;
const Complex I (0.0, 1.0);
```

These definitions allows declarations like

```
Real x = 4.3;
Complex z = 0.6 + 3.2*I;
```

Like all software tricks, these are probably bad ideas that will cause other people no end of headaches. Please let me know if you experience problems. Problems can probably be mitigated by use of namespaces.

5.2 BasisFunc

`BasisFunc` was originally written to represent unit-normalized Complex, vector-valued functions of the form

$$\mathbf{u}(\mathbf{x}) = \sum_{n=0}^{N-1} \hat{\mathbf{u}}_n \bar{T}_n(y) e^{2\pi i(k_x x/L_x + k_z z/L_z)} \tag{82}$$

with the vector dimension fixed at 3, for a particular purpose in my numerical research. Once written, however, `BasisFunc` objects became handy for representing single Fourier components of three-dimensional `FlowFields`.

Typical usage is like this

```
BasisFunc f = u.profile(mx,mz);      // u is a FlowField
f.makePhysical();
f.save("f");                         // save to ASCII file
ComplexChebyCoeff f0 = f[0];         // extract u-component
ComplexChebyCoeff fu = f.u();        // extract u-component
Complex fu_b = f[0].eval_b();        // extract value of u-comp at b
```

5.3 TurbStats

5.4 PoissonSolver

To be written. See example of use in in tests/poissonTest.cpp.

5.5 PressureSolver

To be written. See example of use in in tests/pressureTest.cpp.

6 Design

6.1 Channelflow class heirarchy

To be written.

6.2 Extending Channelflow

To be written.

7 Validation

7.1 Integration of Orr-Sommerfeld eigenfunctions

7.2 Law of the wall in a channel flow

Figure **??** compares the mean velocity profile of a turbulent channel flow computed with Channelflow to theoretical and experimental scaling laws. The figure is plotted in "wall units". Define the *friction velocity* u_* by

$$u^* \triangleq \sqrt{\nu \left.\frac{dU}{dy}\right|_{y=0}} \tag{83}$$

where $U(y)$ is the mean velocity profile. The wall-units for lengths and velocities are then defined by

$$y_+ \triangleq y u^*/\nu \tag{84}$$

$$U_+ \triangleq U/u^* \tag{85}$$

In these units the mean flow of a turbulent channel flow obeys

$$U_+ = y_+ \qquad \text{in the viscous sublayer } y_+ < 10 \tag{86}$$

$$U_+ = 2.5 \ln y_+ + 5 \qquad \text{in the inertial layer} \qquad y_+ > 30 \tag{87}$$

For a derivation of the scaling laws, see Tennekes and Lumley ([**?**]) Section 5.2. The mean velocity profile shown in Figure **??** was computed by `walllaw.cpp` in the `examples/source` directory. The computation uses a $48 \times 97 \times 24$ grid on a domain of size $(L_x, L_y, L_z) = (4\pi/3, 2, 2\pi/3)$, with viscosity $\nu = 1/4000$, resulting in a centerline-velocity Reynolds number of $Re = 3110$ and $u^* = 0.0450$. Integration was RK3 with rotational nonlinearity calculation and dealiasing in xz, with constant mass flux enforced.

8 Benchmarks

8.1 Speed

Channelflow owes its speed to Matteo Frigo and Steven G. Johnson's powerful and elegant FFTW, the Fastest Fourier Transform in the West ([**?**]). Comparisons to Fortran to be redone and written.

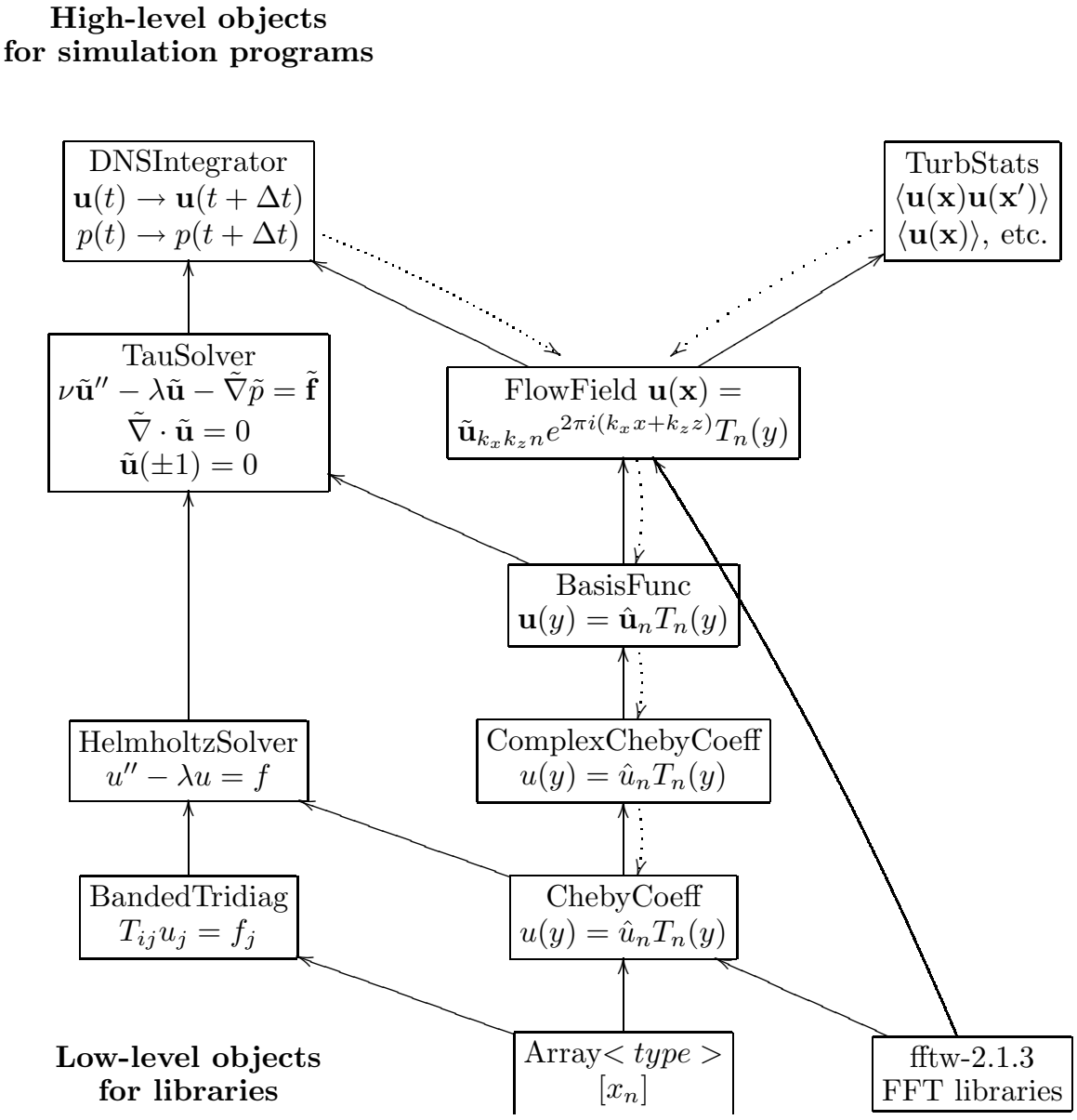

Figure 4: **Structure of Channelflow software.**

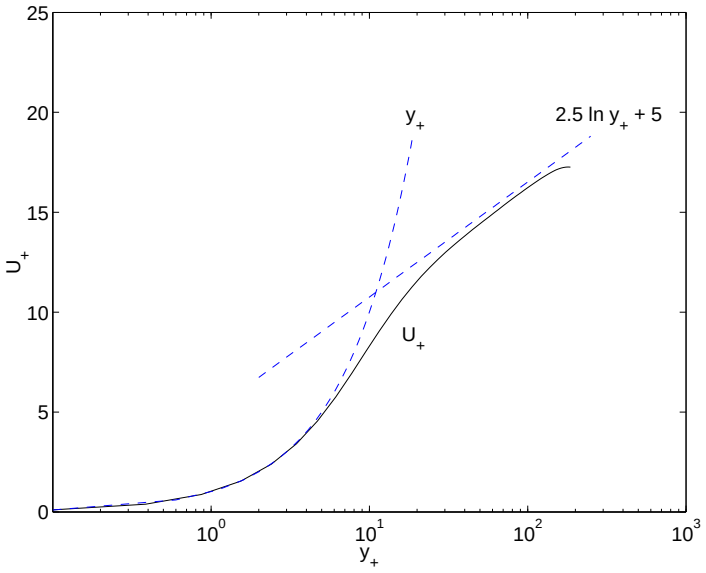

Figure 5: **The law of the wall.** Mean velocity of a turbulent channelflow compared to scaling laws for the viscous sublayer ($U/u^* \approx y_+$) and the inertial layer ($U/u^* \approx 2.5 \ln y_+ + 5$).

8.2 Memory

There is some memory overhead associated with programming in C++. First, C++ programming style encourages use of class member data in order to make objects as independent and self-contained as possible. Second, C++ compilers introduce extra data fields into objects for things like virtual function pointers. These effects can compound quickly when objects contain other objects, or worse, arrays of other objects.

My policy in writing Channelflow was to pay close attention to these effects on large or heavily used objects, where there might be significant cost, but to incur small memory overhead when it resulted in better modularization. The FlowField class, for example, has a array of Reals of length dN^3 for storing d-dimensional real-valued data on an $N \times N \times N$ grid. (For simplicity, let $N_x = N_y = N_z$ in this section.) But FlowField also has several small constant-length data members, which facilitate independence between multiple FlowField objects, and a few N-length arrays, which cache precomputed trigonometric functions. As as result, the size of a FlowField is roughly $dN^3 + aN + b$ reals, where a and b are $O(10)$. For typical N, the $aN + b$ overhead is negligible. Lastly, Channelflow makes minimal use of inheritance and virtual functions, because in my experience, those features make C++ code hard to understand.

To check that C++ memory overhead was in fact negligible, I compared the actual memory consumption of running programs to scaling-law estimates. The scaling laws are derived from the formula

$$\text{\# megabytes} = (\text{\# scalar fields}) \times \frac{N^3 \text{ Reals}}{\text{scalar field}} \times \frac{8 \text{ bytes}}{\text{Real}} \times \frac{1 \text{ megabyte}}{2^{20}\text{bytes}} \tag{88}$$

The minimal set of data for second-order time-stepping has 14 scalar fields, from the four three-dimensional fields $\mathbf{u}^{n+1}, \mathbf{u}^n, \mathbf{f}^n, \mathbf{f}^{n-1}$, and the two scalar fields q^{n+1} and q^n, giving a minimal estimate of $14 \times 2^{-17} N^3$ MB.[2] The CNAB TauSolver caches 6 additional precomputed scalar fields $q_0, v_0, q_+, v_+, q_-, v_-$ for influence-matrix calculations (as noted in Section 4.5.1), resulting in an CNAB estimate of $20 \times 2^{-17} N^3$. The RK3

[2]Clever use of memory during timestepping calculations could probably reduce the number of fields stored below fourteen, but memory is cheap enough that the savings isn't worth the cost to intelligibility of the code.

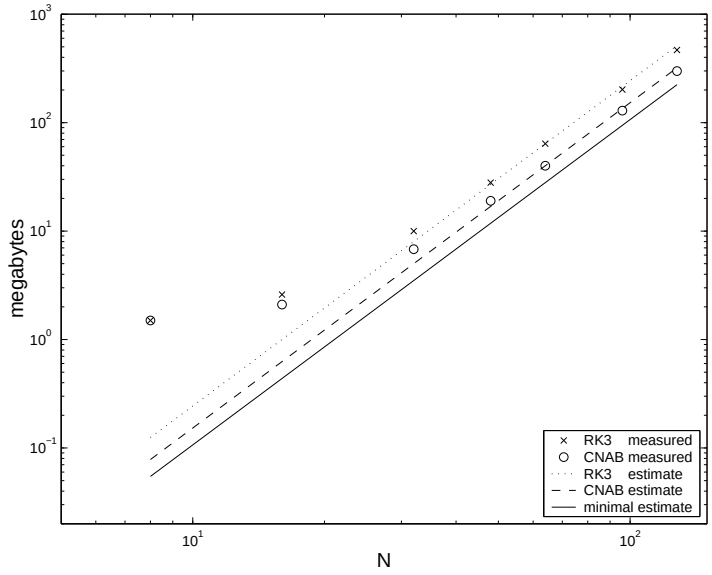

Figure 6: **Memory consumption as function of gridsize.** Comparison of actual memory consumption to scaling-law estimates.

TauSolver caches these 6 additional precomputed scalar fields for each of 3 Runge-Kutta substeps, resulting in an RK3 estimate of $32 \times 2^{-17} N^3$ MB.

Figure **??** compares the actual memory consumption measured by the GNU "top" utility to the scaling-law estimates. The memory overhead is small for $N = 32$ and negligible beyond that. Note that the overhead includes the C libraries, I/O facilities, etc., which accounts for the departure from estimates for small N.[3] Note also, from the scaling-law formula and the figure, that the memory overhead for caching precomputed (q, v) fields for the influence-matrix method is roughly a factor of $3/2$ for CNAB and 2 for RK3. Future releases will probably have an option to eliminate caching at the cost of speed.

9 Software issues

9.1 Installation

9.2 Debugging

The Channelflow library code contains hundreds of safety checks on things like array bounds and Physical/Spectral states of ChebyCoeff and FlowFields. These safety checks are turned off in the optimized libraries. If your Channelflow program produces a segmentation violation or bizarre numerical results, you should recompile your code with debugging flags on and relink to the debugging libraries. For example, for the program foo.cpp, run "make foo.dx" and then either run foo.dx on the console or in a debugger such as gdb. I often set "break exit" in gdb so that I can examine the stack at the moment of exit. For more information on debugging, consult the gdb manual.

[3]For the truly pedantic, note that the estimates for CNAB and RK3 very slightly *overestimate* memory consumption for large N. This is due to the lack of caching of precomputed (q, v) fields for aliased Fourier modes.

References

[1] U. Ascher, S. Ruuth, and B. Wetton. Implicit-explicit methods for time-dependent partial differential equations. *SIAM J. Numer. Anal.*, 32(3):797–823, June 1995.

[2] C. Canuto, M.Y. Hussaini, A. Quateroni, and T.A. Zhang. *Spectral Methods in Fluid Dynamics*. Springer-Verlag, 1988.

[3] M. Frigo and S.G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[4] L. Kleiser and U. Schuman. Treatment of incompressibility and boundary conditions in 3-d numerical spectral simulations of plane channel flows. In E. Hirschel, editor, *Proc. 3rd GAMM Conf. Numerical Methods in Fluid Mechanics*, pages 165–173, Viewweg, Braunschweig, 1980. GAMM.

[5] R. Peyret. *Spectral Methods for Incompressible Flows*. Springer-Verlag, 2002.

[6] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1988.

[7] P.R. Spalart, R.D. Moser, and M.M. Rogers. Spectral methods for the Navier-Stokes equations with one infinite and two periodic directions. *J. Comp. Physics*, 96:297–324, 1990.

[8] H. Tennekes and J.L. Lumley. *A First Course in Turbulence*. MIT Press, 1972.

[9] T.A. Zang. On the rotation and skew-symmetric forms for incompressible flow simulations. *Appl. Numer. Math.*, 7:27–40, 1991.

[10] T.A. Zang and M.Y. Hussaini. Numerical experiments on subcritical transition mechanisims. *AIAA paper*, 85-0296, 1985.