Building DNS

A sample code might have form like

```
#include <iomanip>
#include <iostream>
#include "channelflow/dns.h"
#include "channelflow/flowfield.h"
#include "channelflow/utilfuncs.h"

using namespace std;
using namespace chflow;

int main(int argc, char* argv[]) {
    cfMPI_Init(&argc, &argv);
    {
        /* main code */
    }
    cfMPI_Finalize();
}
```

Define parameters

Domain configuration

As a normal simulation's configuration, we need to define boundaries of domain, including lengths and numbers of points for each direction. For example, for 3d configuration, lengths and numbers of points of axises are (L_x, L_y, L_z) and coressponding (N_x, N_y, N_z). Note that, in Channelflow framework, y-axis is determined as vertical direction of computational domain. As a result, horizontal directions contain x- and z-axis. However, it's a litte of difficulty in 2d configuration for defining $L_y = 0$ (or $L_y \approx 0$) in Channelflow. In this case, we can set an L_z value small enough to suppress instability in z variation in the velocity field (but not so small as to cause CFL problems)(read this), coressponding to a thin point layer, for example, $L_z \approx 0.05$ and $N_z \approx 6$ to avoid 3d behaviours generated.

```
// Define gridsize
const int Nx = 16; // Nx (horizontal)
const int Ny = 15; // Ny (vertical)
const int Nz = 16; // Nz (horizontal)

// Define box size
const Real Lx = pi; // Lx
const Real Ly_min = 0.0; // Ly_min = a
const Real Ly_max = 1.0; // Ly_max = b
const Real Lz = pi; // Lz
```

In offical documents of Channelflow, Ly_min and Ly_max can be called a and b, respectively.

Flow parameters

Some potential flow properties are Reynolds number, kinematic viscosity, ... or same kinds of parameter.

```
const Real Reynolds = 400.0; // Reynolds number
const Real nu = 1.0 / Reynolds; // kinematic viscosity
const Real dPdx = 0.0; // mean pressure gradient
```

Solver parameters

For solver, we also need to set parameters regarding core system of solver, such as integration parameters.

```
// Define integration parameters
const int n = 32;          // take n steps between printouts
const Real dt = 1.0 / n;   // integration timestep
const Real T = 30.0;       // integrate from t=0 to t=T
```

Build governing equations for DNS solver

Flow fields

To build fluid dynamic problem, flow fields are required to take governing equations together.

```
// Construct data fields: 3d velocity and 1d pressure
cout << "building velocity and pressure fields..." << flush;
vector<FlowField> fields = {
    FlowField(Nx, Ny, Nz, 3, Lx, Lz, a, b) // define 3d-vector velocity field
    ,FlowField(Nx, Ny, Nz, 1, Lx, Lz, a, b)  // define scalar pressure field
    // ,FlowField(Nx, Ny, Nz, 1, Lx, Lz, a, b)  // define scalar temperature field
    // ,FlowField(Nx, Ny, Nz, 1, Lx, Lz, a, b)  // define scalar sanility field
};
cout << "done" << endl;
```

For convernience, we can define some macros for variables before the main function or in external libraries, might like this

```
#define U fields[0] // for velocity field
#define p fields[1] // for pressure field
// #define T fields[2] // maybe for temperature field
// #define S fields[3] // maybe for sanility field
```

then we can get fields easily as `U` or `p` instead of calling `fields[0]` for velocity or `fields[1]` for pressure.

Differential operators and norms

Convenience form	**Preferred form**	**Meaning**
FlowField g = xdiff(f)	xdiff(f,g)	$\boldsymbol{g} = \partial \boldsymbol{f} / \partial x$
FlowField g = xdiff(f,n)	xdiff(f,g,n)	$\boldsymbol{g} = \partial^n \boldsymbol{f} / \partial x^n$

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

Convenience form	Preferred form	Meaning
FlowField g = dot(f)	dot(f,h,g)	$g = \boldsymbol{f} \cdot \boldsymbol{h}$
FlowField g = outer(f)	outer(f,h,g)	$g_{ij} = f_i h_j$

Convenience form	Meaning
Real r = L2Norm2(f)	$r = \frac{1}{L_x L_y L_z} \int_0^{L_x} \int_0^{L_y} \int_0^{L_z} f \cdot f \, dx \, dy \, dz$
Real r = L2Norm(f)	$r = \sqrt{\mathrm{L2Norm2}(f)}$
Real r = L2Dist2(f, g)	$r = \mathrm{L2Norm2}(f - g)$
Real r = L2Dist(f, g)	$r = \mathrm{L2Norm}(f - g)$
Real r = bcNorm2(f)	$r = \frac{1}{L_x L_z} \int_0^{L_x} \int_0^{L_z} \left(f \cdot f\vert_{y=0} + f \cdot f\vert_{y=L_y} \right) dx dz$
Real r = bcNorm(f)	$r = \sqrt{\mathrm{bcNorm2}(f)}$
Real r = bcDist2(f, g)	$r = \mathrm{bcNorm2}(f - g)$
Real r = bcDist(f, g)	$r = \mathrm{bcNorm}(f - g)$
Real r = divNorm2(f)	$r = \mathrm{L2Norm2}(\nabla \cdot f)$
Real r = divNorm(f)	$r = \sqrt{\mathrm{divNorm2}(f)}$
Real r = divL2Dist2(f, g)	$r = \mathrm{divNorm2}(f - g)$
Real r = divL2Dist(f, g)	$r = \mathrm{divNorm}(f - g)$

Initial conditions

```
// Define size and smoothness of initial disturbance
Real spectralDecay = 0.5;
Real magnitude = 0.1;
int kxmax = 3;
int kzmax = 3;
// Perturb velocity field
fields[0].addPerturbations(kxmax, kzmax, 1.0, spectralDecay);
fields[0] *= magnitude / L2Norm(fields[0]);
```

Flags of DNS solver

```
// Define DNS parameters
DNSFlags flags;
flags.baseflow = LaminarBase;

// set time-stepping algorithm
// CNFE1 or SBDF1: 1st-order Crank-Nicolson, Foward-Euler or 1st-order Semi-implicit Backward Diff
// CNAB2: 2nd-order Crank-Nicolson, Adams-Bashforth
// CNRK2: 2nd-order semi-implicit Crank-Nicolson, Runge-Kutta algorithm
// SMRK2: 2nd-order semi-implicit Runge-Kutta
// SBDF2, SBDF3, SBDF4: 2nd, 3rd, and 4th-order Semi-implicit Backward Differentiation Formulae
flags.timestepping = SBDF3; // CNFE1, CNAB2, CNRK2, SMRK2, SBDF1,SBDF2,SBDF3[default], SBDF4

// set initialization timestepping algorithm
// Some of the time-stepping algorithms listed above (SBDF in particular) require data from N prev
// This indicates that DNS class instead takes its first N steps with an initialization timesteppi
flags.initstepping = CNRK2; // CNFE1, CNRK2[default], SMRK2

// set form of nonlinear term of Navier-Stokes calculation
flags.nonlinearity = Rotational; // Rotational, SkewSymmetric[default], Alternating, Linearized
// Nonlinear terms are calculated with collocation methods
flags.dealiasing = DealiasXZ; // DealiasXZ=2/3, NoDealiasing, DealiasY=3/2, DealiasXYZ
// flags.nonlinearity = SkewSymmetric;
// flags.dealiasing   = NoDealiasing;

// boundary conditions
flags.ulowerwall = -1.0; // boundary condition U(z=minLz=a)=-1.0
flags.uupperwall = 1.0; // boundary condition U(z=maxLz=b)=1.0

flags.taucorrection = true;
flags.constraint = PressureGradient;  // enforce constant pressure gradient
flags.dPdx = dPdx;
flags.dt = dt;
flags.nu = nu;
```

Build Navier-Stoke integrator

```
// Construct Navier-Stoke integrator, set integration method
cout << "building DNS..." << flush;
DNS dns(fields, flags);
cout << "done" << endl;
```

This code is for Couette problem as sample code.

Building a DNS solver of custom governing equations will be written later

Main time loop

```
mkdir("data"); // create 'data' folder for saving data of simulation
Real cfl = dns.CFL(fields[0]); // compute CFL factor from velocity
for (Real t = 0; t <= T; t += n * dt) {
    // print real-time information of running simulation
    cout << "         t == " << t << endl;
    cout << "       CFL == " << cfl << endl;
    cout << " L2Norm(u) == " << L2Norm(fields[0]) << endl;
    cout << "divNorm(u) == " << divNorm(fields[0]) << endl;
    cout << "      dPdx == " << dns.dPdx() << endl;
    cout << "     Ubulk == " << dns.Ubulk() << endl;

    // Write velocity and modified pressure fields to disk
    fields[0].save("data/u" + i2s(int(t))); // 3d velocity
    fields[1].save("data/q" + i2s(int(t))); // scalar pressure

    // Take n steps of length dt
    dns.advance(fields, n);
    cout << endl;
}
```