Spectral Simulation of Turbulent Flow

Mahendra K. Verma

September 17, 2007

Contents

Chapter 1

Introduction

In this report we describe a procedure called *pseudo-spectral method* to solve the equations for the incompressible fluid flows under various conditions. The problems attempted are

1.1 Incompressible fluid flow

The equations are

$$\begin{aligned} \frac{\partial \mathbf{U}}{\partial t} + (\mathbf{U}\cdot\nabla)\mathbf{U} &= -\nabla p + \nu\nabla^2\mathbf{U} + \mathbf{f}^{\mathbf{U}}, \\ \nabla\cdot\mathbf{U} &= 0, \end{aligned}$$

where $\mathbf{U}$, p, and $\mathbf{f}^{\mathbf{U}}$ are the velocity field, pressure field, and the forcing field respectively, and ν is the viscosity.

Nondimensionalize

Re no.. turbulence.. reqd size etc.

Boundary condition

1.2 Flow of a passive scalar in an incompressible flow

The equations are

$$\begin{aligned} \frac{\partial \mathbf{U}}{\partial t} + (\mathbf{U}\cdot\nabla)\mathbf{U} &= -\nabla p + \nu\nabla^2\mathbf{U} + \mathbf{f}^{\mathbf{U}} \\ \frac{\partial \zeta}{\partial t} + (\mathbf{U}\cdot\nabla)\zeta &= \kappa\nabla^2\zeta + f^{\zeta}, \\ \nabla\cdot\mathbf{U} &= 0, \end{aligned}$$

where ζ, and f^{ζ} are the scalar and forcing fields respectively, and κ is the diffusive coefficient.

Nondimensionalize.
Boundary condition

1.3 MHD

The equations are

$$\begin{aligned} \frac{\partial \mathbf{U}}{\partial t} + (\mathbf{U}\cdot\nabla)\mathbf{U} &= -\nabla p + \nu\nabla^2\mathbf{U} + \mathbf{f}^{\mathbf{U}} \\ \frac{\partial \mathbf{B}}{\partial t} + (\mathbf{U}\cdot\nabla)\mathbf{B} &= (\mathbf{B}\cdot\nabla)\mathbf{U} + \eta\nabla^2\mathbf{B} + \mathbf{f}^{\mathbf{B}}, \\ \nabla\cdot\mathbf{U} &= 0, \end{aligned}$$

where $\mathbf{B}$, and $\mathbf{f}^{\mathbf{B}}$ are the magnetic field and the magnetic-forcing field respectively, and η is the diffusive coefficient.

Nondimensionalize.
Boundary condition

1.4 Rayleigh Benard convection

eqn
Nondimensionalize
Boundary condition

1.5 Magnetoconvection

eqn
Nondimensionalize
Boundary condition

1.6 Programming strategy

We implement the pseudo-spectral method in an object-oriented language C++. We notice that most of the operations in the simulation are common for all the solvers, be it fluid turbulence or MHD turbulence. Therefore, we create generic library function. For example compute_nlin...

Library of basis functions
Lib Fields
Lib IncFlow
Lib IncFlow
then.. solvers

1.7 Notation

- Vector fields $\mathbf{V}, \mathbf{W}$
- Scalar field ζ
- Forward transform $\mathcal{F}$
- Inverse transform $\mathcal{F}^{-1}$
- Flux Π
- Shell-to-shell transfer T_{nm}^{uu}
- Derivative

Basis Functions

Chapter 2

Fourier Transform

A library named libfourier.a contains functions that performs Fourier transforms in 1D, 2D, and 3D. There are functions that compute total energy, energy spectrum, etc. for the function.

2.1 Definitions

Fourier transform is defined for a continuous differentiable function $f(\mathbf{x})$ (strictly speaking square Lebesgue integrable functions L^2) defined in a D-dimensional periodic box. If the size of the box along sth direction is L_s, then the Foward Fourier transform is defined as

$$\hat{f}_{\mathbf{K}} = \frac{1}{\prod L_s} \int d\mathbf{x} f(\mathbf{x}) \exp(-i\mathbf{K} \cdot \mathbf{x}),$$

where $\mathbf{K}$ is the wavenumber, and $\hat{f}_{\mathbf{K}}$ is the function in Fourier space. We can invert the above function, and the inverse Fourier transform is

$$f(\mathbf{x}) = \sum_{\mathbf{K}} \hat{f}_{\mathbf{K}} \exp(i\mathbf{K} \cdot \mathbf{x})$$

In numerical computation, we discretize the real space. Suppose the sth direction is discretized in N_s segments, then $x_s = j_s L_s / N_s$ with $j_s = (0, N_s - 1)$. The component of wave vector along s is

$$K_s = 2\pi k_s / L_s = k_s \text{kfactor}(s), \tag{2.1}$$

where k_s is an integer, and $\text{kfactor}(s) = 2\pi / L_s$. As a result of descretization, the above equations translate to

$$\hat{f}_{\mathbf{k}} = \frac{1}{\prod^{N_s}} \sum_{\mathbf{k}} f_{\mathbf{j}} \exp\left(-2\pi i \sum_s \frac{j_s k_s}{N_s} \right), \tag{2.2}$$

$$f_{\mathbf{j}} \quad = \quad \sum_{\mathbf{j}} \hat{f}_{\mathbf{k}} \exp\left(2\pi i \sum_{s} \frac{j_s k_s}{N_s}\right), \tag{2.3}$$

where $\mathbf{j}$ and $\mathbf{k}$ are a vector comprising of j_s and k_s respectively. These formulas are called *Discrete Fourier Transforms* (DFT).

The number of independent variables $f_{\mathbf{j}}$ or $\hat{f}_{\mathbf{k}}$ must be the same in Eqs. (2.2,2.3) or in both real and Fourier space for the transformation in both forward and inverse directions. In our turbulence simulation *we take* $f_{\mathbf{j}}$ *to be real.* Since the variables $\hat{f}_{\mathbf{k}}$ are complex, the number of wavenumbers in the Fourier space is roughly half of the number of grid points in the real space. As an example, take a periodic real function $f(x)$ defined in one dimension and descretized into N points ($0 \le j < N$). In the Fourier space, the range of wavenumbers are $0 \le k \le N/2$ with $\hat{f}_0$ and $\hat{f}_{N/2}$ as real and $\hat{f}_1, ..., \hat{f}_{k-1}$ as complex. Hence the number of independent variables in both real and Fourier space are N.

In the above two equations of DFT are independent of the box size L_s. It is important to understand how the size of the system enters in spectral simulations. The wavenumber K_s is connected to the integer index k_s by Eq. (2.1). If we choose kfactor$(s) = 1$ for all s, then $K_s = k_s$, and the box size along all directions are 2π. For this case the wavenumbers in the sth direction is $K_s = 1, 2, ..., N_s/2$, and the shortest wavelength in this direction is $2\pi/k_{max} = 4\pi/N_s$.

The above example is trivial one. Let us consider a simulation in a box of grids $(N, 5N)$. What is the system size? System size cannot be fixed by the definitions of DFT, but it is determined by kfactor. If kfactor(1) = kfactor(2) = 1, then the size in both the directions are 2π, and $K_s = k_s$ with $K_1 = 0, 1, .., N/2$ but $K_2 = 0, 1, 2, .., 5N/2$. Clearly the shortest wavelength in the x_2 direction is five times shorter because the box size is the same in both the directions. On the contrary, if we choose kfactor$(1) = 1$ and kfactor$(2) = 1/5$, then $L_1 = 2\pi$ and $L_2 = 10\pi$, and $K_1 = k_1$ and $K_2 = k_2/5$ with$K_1 = 0, 1, .., N/2$ and $K_2 = 0, 1/5, 2/5, .., N/2$. Hence, the minimum wavelength in both the directions is the same as expected, but the maximum nonzero wavelength (10π) is five times larger along x_2 compared to that along x_1, consistent with the ratio of the sizes along both the directions.

It is also important to note that the size of the system (or the difference between k_s and K_s) does not appear in convolution calculation, but it appears in the viscous term. Before we get into implementation issues, some of the useful properties of Fourier transforms are as follows.

For real $f(\mathbf{x})$

$$\hat{f}_{-\mathbf{k}} = \left(\hat{f}_{\mathbf{k}}\right)^*.$$

Another general and useful property of the Fourier transform is

$$\hat{f}_{...,k_s+N_s,..} = \hat{f}_{...,k_s,...}.$$

2.2 Energy computation

The energy density of the function $f(\mathbf{x})$, i.e., energy per unit volune, is defined as

$$E = \frac{1}{\prod L_i} \frac{1}{2} \int d\mathbf{x} |f(\mathbf{x})|^2.$$

Using the definition of Fourier transform we can show that

$$E = \frac{1}{2} \sum_{\mathbf{k}} |\hat{f}(\mathbf{k})|^2,$$

with $|\hat{f}(0)|^2/2$ as the energy of the mean field.

We define the isotropic energy spectrum $e(K)$ of the field as the energy contained in the wavenumber shell K. We compute this function using

$$e(K) = \frac{1}{2} \sum_{K \le K' < K+1} |\hat{f}(\mathbf{K}')|^2.$$

The function $e(K)$ contains the energy of the Fourier variables on the inner surface of the shell, but not that of the outer surface of the shell. Note that $\mathbf{K}$ is the wavenumber, not the array index k (recall $K_s = k_s \times \text{kfactor}_s$).

In the similar manner we define a function $S_n(K)$ that is defined as

$$S_n(K) = \frac{1}{2} \sum_{K \le K' < K+1} K^n |\hat{f}(K')|^2.$$

The function $S_n(K)$ is useful for computing dissipation spectrum etc.

If we have two functions f and g, then another function that appears similars to the energy is

$$E^{f,g} = \frac{1}{2} \sum_{\mathbf{k}} \Re[f(\mathbf{k}) \times \text{conj}(g(\mathbf{k}))].$$

We use this definition to compute cross helicity in magnetohydrodynamics. One can define appropriate isotropic energy spectrum for the above function.

2.3 Implementation

We implement Fourier transform and its inverse using FFTW (Fastest Fourier Transform in the West). Since velocity and magnetic field etc. are real, only $k_{lastindex} \ge 0$ are stored. We use FFTW library for the transforms that has stores variables f in a specific manner.

- For a 1D real array $f(N_1)$, FFTW allocates $(N_1/2 + 1)$ dimensional complex array. The real variables are stored up to N_1, but Fourier space variables stored up to $k = 0 : N_1/2$. Note that $\hat{f}(0)$ and $\hat{f}(N_1/2)$ are real.

- For a 2D real array $f(N_1, N_2)$, FFTW allocates $N_1 \times (N_2/2+1)$ dimensional complex array. The real variables are stored up to $(0 : N_1 - 1, 0 : N_2 - 1)$, but Fourier space variables are stored up to $\mathbf{k} = (-N_1/2 + 1 : N_1/2, 0 : N_2/2)$. Since $\hat{f}_{k_x+N_x,k_y} = \hat{f}_{k_x,k_y}$, fields with negative wavenumber arguments are stored with a shift of N_x. That is ,$\hat{f}_{-k_x,k_y}$ is stored in location $(-k_x + N_x, k_y)$. As a result, k_x are stored in the order of $(k_x = 0, 1, .., N_1/2, -N_1/2 + 1, -N_1/2 + 2, .., -1)$ along the first index. See Fig. ??? for illustration.

- For a 3D real array $f(N_1, N_2, N_3)$, FFTW allocates $N_1 \times N_2 \times (N_3/2+1)$ dimensional complex array. The real variables are stored up to $(0 : N_1 - 1, 0 : N_2 - 1 : N_{3-1})$, but Fourier space variables are stored up to $\mathbf{k} = (-N_1/2+1 : N_1/2, -N_2/2+1 : N_2/2, 0 : N_2/2)$. Since $\hat{f}_{k_x+N_x,k_y+N_y,k_z} = \hat{f}_{k_x,k_y,k_z}$, the field variables with negative wavevector arguments are stored with a shift of N_x and N_y similar to the scheme for 2D. See Fig. ??? for an illustration.

In the energy computation, the sum over the wavenumber is for all the wavenumbers. Since we store wavenumber modes with $k_{lastimdex} \geq 0$ in our simulation, we double the contributions of all those modes for which a corresponding complex conjugate is not stored. The calculation of isotropic energy spectrum is done by summing the energy over the shells. Some of the shells are present only partially in our simulation box. For these shells we compute average energy per mode in the shell from the modes present in the shell, and then multiply the average energy per mode by the volume of the shell. In 2D, the volume factor is πK, while in 3D the factor is $2\pi K^2$. Note that our wavenumber summation is over the modes with $k_{lastindex} \geq 0$.

We also use two two definitions: `DP` that could take value `double` or `float`, and `complx` that takes value `complex<DP>`. The statements are

```
#define DP double
#define complx complex<DP>
```

2.4 Functions

We have divided the functions in two groups. The first group of functions deal with the implementation of Fourier transform, and the second group of functions deal with the computation of energy, energy spectrum etc. The details are as follows:

2.4.1 fourier.cc

This file contains functions related to Fourier transform. Here we describe the functions in some details. n takes values 1,2, or 3.

- `void Init_fftw_plan_FOUR(int NN[],Array<complx,n> A)`: Inializes fftw plans `r2c_plan_FOUR, c2r_plan_FOUR that are global variables.`

- `void ArrayFFTW_FOUR(fftw_plan r2c_plan_FOUR, Array<complx,n> A)`: FFTW Forward transform of array A.
- `void ArrayFFT_FOUR(fftw_plan r2c_plan_FOUR, int N[],Array<complx,n> A)`: Zeropad, FFTW forward transform, Normalize by Norm_FOUR.
- `void ArrayIFFT_FOUR(fftw_plan c2r_plan_FOUR, int N[], Array<complx,n> A)`: FFTW Inverse transform of array A.
- `void Norm_FOUR(int N[],Array<complx,n> A)`: Divides array A by $\prod N[i]$.
- `void Zero_pad_lastrow_FOUR(int N[],Array<complx,n> A)`: In 1D, $A(N/2) = 0$; In 2D, the row $k_2 = N_2/2$ is set to zero; In 3D, the plane $k_3 = N_3/2$ is set to zero.
- `void Xderiv_FFT(int NN[], Array<complx,n> A, Array<complx,n> B, DP kfactor[])`: $B(\mathbf{k}) = iK_xA(\mathbf{k})$ or $\mathcal{F}(B(\mathbf{x})) = \mathcal{F}(dA(\mathbf{x})/dx)$ where $\mathcal{F}$ is the Fourier transform operator. Note that $K_x = k_x * \text{kfactor}(1)$.
- `void Yderiv_FFT(int NN[], Array<complx,n> A, Array<complx,n> B, DP kfactor[])`: $B(\mathbf{k}) = iK_yA(\mathbf{k})$ or $\mathcal{F}(B(\mathbf{x})) = \mathcal{F}(dA(\mathbf{x})/dy)$.
- `void Zderiv_FFT(int NN[],Array<complx,n> A, Array<complx,n> B, DP kfactor[])`: $B(\mathbf{k}) = iK_zA(\mathbf{k})$ or $\mathcal{F}(B(\mathbf{x})) = \mathcal{F}(dA(\mathbf{x})/dz)$.

2.4.2 four_inline.h

- `inline int min(int a, int b)`: Returns minimum of a and b.
- `inline DP min(DP a, DP b)`: Returns minimum of a and b.
- `inline int Get_kx_FOUR(int i1, int N[])`: For 2D and 3D, the function returns k_x given i_x using the formula i_1: $(i_1 \le N_1/2)\ ?\ i_1 : (i_1 - N_1)$; here $-N_1/2 < k_1 \le N_1/2$. In 1D, $k_x = i_1$ that lies in the range of $(0 : N_1/2)$.
- `inline int Get_ix_FOUR(int kx, int N[])`: For 2D and 3D, the function returns i_x given k_x : $k_x \ge 0\ ?\ k_x : (k_x + N_1)$; here$-N_1/2 < k_x \le N_1/2$. In 1D, $i_x = k_x$ that lies in the range of $(0 : N_1/2)$.
- `inline int Get_ky3D_FOUR(int i2, int N[])`: For 3D, the function returns k_y given i_y using the formula $k_y = i_y$ if $i_y \le N_2/2$, else $k_y = i_y - N_2$. In 2D, $k_y = i_y$.
- `inline int Compute_iy3D_FOUR(int ky, int N[])`: Computes i_y given k_y in 3D.
- `inline DP Kmagnitude_FOUR(int i1, int i2, int N[], DP kfactor[])`: Returns kmagnitude $\sqrt{\sum(\text{kfactor}(i) \times k_i)^2}$ for 2D.

- `inline int Min_radius_outside_FOUR(int N[], DP kfactor[])`: Returns radius of the smallest sphere enclosing the cube $(\text{kfactor}(1)N_1/2, \text{kfactor}(2)N_2/2)$ in 2D and $(\text{kfactor}(1)N_1/2, \text{kfactor}(2)N_2/2, \text{kfactor}(3)N_3/2)$ in 3D. The returned value is `Kmagnitude`$(N_1/2,\ N_2/2,\ ..)+1$.

- `inline int Max_radius_inside_FOUR(int N[], DP kfactor[])`: Returns radius of the largest sphere that can fit inside the cube $(\text{kfactor}(1)N_1/2, \text{kfactor}(2)N_2/2)$ in 2D and $(\text{kfactor}(1)N_1/2, \text{kfactor}(2)N_2/2, \text{kfactor}(3)N_3/2)$ in 3D.

2.4.3 field_internal_four.cc

This file contains functions that operate on $\hat{f}(\mathbf{k})$ and compute total energy, energy spectrum etc. To compute these quantities, the sum is done over all the wavenumbers. However we store only modes with $k_{lastindex} \geq 0$ in our computer simulation. Therefore we double the contributions from the modes $k_{lastindex} \neq 0$; the contributions from the modes at the common surface $k_{lastindex} = 0$ are not doubled.

- `DP Total_abs_sqr_FOUR(int N[], Array<complx,n> A)`: Returns $|A(\mathbf{k})|^2/2$ for all $\mathbf{k}$ but without $\mathbf{k} = 0$.

- `complx Total_termbyterm_mult_FOUR(int N[], Array<complx,n> A, Array<complx,n> B)`: Returns $A(\mathbf{k})conj(B(\mathbf{k}))/2$ without $\mathbf{k} = 0$.

- `DP Total_Sn_FOUR(int N[], Array<complx,n> A, DP n, DP kfactor[])`: Returns $K^n|A(\mathbf{k})|^2/2$ without $\mathbf{k} = 0$.

- `void Compute_1D_Sk_FOUR(int N[], Array<complx,1> A, DP n, Array<DP,1> Sk, DP kfactor[])`: For $D = 1$, $Sk(k) = K^n|A(k)|^2/2$. Note that the array index is k not K. $Sk_{n=0}(0)$ is the mean energy.

- `void Compute_1D_Sk_FOUR(int N[], Array<complx,1> A, Array<complx,1> B, DP n, Array<DP,1> Sk, DP kfactor[])`: For $D = 1$, $Sk(k) = K^n\Re[A(k)\times conj(B(k))]/2$.

- `void Compute_isotropic_Sk_FOUR(int N[], Array<complx,n> A, DP n, Array<DP,1> Sk)`: For $D = 2, 3$ isotropic energy spectrum $Sk(K) = \sum K'^n|A(\mathbf{K}')|^2/2$ with $K \leq K' < K + 1$. $Sk_{n=0}(0)$ is the mean energy.

- `void Compute_isotropic_Sk_FOUR(int N[], Array<complx,n> A, Array<complx,n> B, DP n, Array<DP,1> Sk, DP kfactor[])`: $Sk(K) = \sum K'^n\Re[A(K')\times conj(B(K'))]/2$ with $K \leq K' < K + 1$.

- `DP Shell_termbyterm_mult_FOUR(int N[], Array<complx,n> A, Array<complx,n> B, DP inner_radius, DP outer_radius, DP kfactor[])`: Returns $\sum_K \Re(A(K)*$ $conj(B(K)))$ `for inner_radius` $\leq K <$ `outer_radius .`

- `void Shell_termbyterm_mult_real_imag_FOUR(int N[], Array<complx,2> A, Array<complx,2> B, DP inner_radius, DP outer_radius, DP kfactor[], DP& total_real, DP& total_imag)`: total_real $= \Re(A(K) * \Re(B(K)$, and total_imag $= \Im(A(K)) * \Im(B(K))$ for `inner_radius` $\leq K <$ `outer_radius` .
- `void Array_divide_ksqr_FOUR(int N[],Array<complx,n> A, DP kfactor[])`: $A(\mathbf{k}) \rightarrow A(\mathbf{k})/K^2$, $A(0) = 0$.
- `void Array_mult_ksqr_FOUR(int N[],Array<complx,n> A, DP kfactor[])`: $A(\mathbf{k}) \rightarrow K^2 A(\mathbf{k})$.
- `void Array_exp_ksqr_FOUR(int N[],Array<complx,n> A, DP factor, DP kfactor[])`: $A(\mathbf{k}) \rightarrow \exp(\texttt{factor} \times K^2) A(\mathbf{k})$.

Exercises

1. Derive a formula for the forward sin transform given the formula for the invese sin transform, both for continuous and discrete transforms (Eqs. (2.2, 2.3)
2. We wish to perform spectral simulation for the flow in a box of size 1m $\times$ 1m $\times$ 10m with a lowest scale of 1mm in each of the tree directions. What should be our strategy in terms of array allocation? What kfactor would you chooose along each direction?
3. How would the strategy change of the lowest resolution for the Exercise 2 was 1mm $\times$ 1mm $\times$ 5mm?

Chapter 3

Sin Cos transform

A library named libsincosfourier.a contains functions that performs *sin/cos* transforms along x direction, and Fourier transform along the other directions. In this chapter we will discuss sin and cos transform.

3.1 Definitions

3.1.1 Sin transform

Consider a continuous periodic function $f(x)$ with period $2L_x$ that is odd around $x = 0$. Clearly $f(0) = f(L_x) = 0$. We perform sin transform on this function:

$$\hat{f}_{K_x} = \frac{1}{2L_x} \int dx f(x) 2\sin(K_x x),$$

where $K_x = m\pi/L_x$ ($m \geq 0$) is the wavevector along x. The inverse-sin transform is defined as

$$f(x) \quad = \quad \sum_{K_x} \hat{f}_{K_x} 2\sin(K_x x).$$

We discretize the space along x direction into N_x segments, then $x = jL_x/N_x$ with $j = (0, N_x - 1)$. The component of wave vector along x direction is $K_x = m\pi/L_x = \pi \times \text{kfactor}(1)$ with $m > 0$. As a result of descretization, the above equations become

$$\hat{f}_m = \frac{1}{2N_x} \sum_j f_j 2\sin\left(\pi \frac{mj}{N_x}\right), \tag{3.1}$$

and

$$f_j = \sum_m \hat{f}_m 2\sin\left(\pi \frac{mj}{N_x}\right). \tag{3.2}$$

These formulas are called *Discrete Sin Transforms* (DST).

We can rewrite Eq. (4.1) as

$$f_j = \sum_m \frac{\hat{f}_m}{i} \exp\left(2\pi i \frac{mj}{2N_x}\right) + \left(-\frac{\hat{f}_m}{i}\right) \exp\left(-2\pi i \frac{mj}{2N_x}\right).$$

Hence, the sin transform is equivalant to the Fourier transform in a box twice as big with $[\mathrm{FT}(f)]_m = \hat{f}_m/i$ and $[\mathrm{FT}(f)]_{-m} = -\hat{f}_m/i$. The sin transform is useful for odd functions. Here we do not need to store the function over the complete period; only one half is enough.

After sin transform, we will discuss the cos transform.

3.1.2 Cos transform

Consider a continuous periodic function $f(x)$ with period $2L_x$ that is even around $x = 0$. We perform cos transform on this function:

$$\hat{g}_{K_x} = \frac{1}{2L_x} \int dx f(x) 2\cos(K_x x),$$

where $K_x = m\pi/L_x$ ($m \geq 0$) is the wavevector along x. The inverse-cos transform is defined as

$$g(x) = \hat{g}_0 + \sum_{K_x} \hat{g}_{K_x} 2\cos(K_x x).$$

Using the same discretization procedure as described for sin transform, the above equations become

$$\hat{g}_m = \frac{1}{2N_x} \sum_j g_j 2\cos\left(\pi \frac{mj}{N_x}\right) \tag{3.3}$$

$$g_j = \hat{g}_0 + \sum_m \hat{g}_m 2\cos\left(\pi \frac{mj}{N_x}\right). \tag{3.4}$$

These equations are called *Discrete Cosine Transforms* (DCT).

We can rewrite Eq. (4.3) as

$$g_j = \hat{g}_0 + \sum_m \hat{g}_m \exp\left(2\pi i \frac{mj}{2N_x}\right) + \hat{g}_m \exp\left(-2\pi i \frac{mj}{2N_x}\right).$$

Hence, the cos transform is equivalant to the Fourier transform in a box twice as big with $[\mathrm{FT}(fg)]_m = \hat{g}_m$ and $[\mathrm{FT}(g)]_{-m} = -\hat{g}_m$. The cos transform is useful for even functions. Here we do not need to store the function over the complete period; only one half is enough.

3.1.3 Energy computations

The total energy (strictly speaking energy density, i.e., energy per unit length) of a homogenous odd function $f(x)$ (sin transform) is defined as

$$E = \frac{1}{L_x} \frac{1}{2} \int dx |f(x)|^2,$$

which can be shown to be equal to

$$E = \sum_{m>0} \left| \hat{f}_m \right|^2$$

Total energy for an homogeneous even function g can be computed similarly:

$$E = \frac{1}{2} |\hat{g}_0|^2 + \sum_{m>0} |\hat{g}_m|^2 .$$

As discussed in the earlier chapter, the total energy in terms of Fourier transforms is

$$E = \frac{1}{2} \sum_{\mathbf{k}} |\hat{f}_{\mathbf{k}}|^2,$$

if consider the box of size $2L_x$. As discussed in the previous sections, the sin and cos transforms contain both $k = \pm m$ modes with $|\hat{f}_{-m}| = |\hat{f}_m|$ and $|\hat{g}_{-m}| = |\hat{g}_m|$. When we sum both positive and negative wavenumber modes, the factor $1/2$ cancels as seen in the expressions for the energy of the cos and sin transformed variables.

3.2 Implementation

We implement sin and cos transforms using FFTW functions. We use FFTW's RODFT01 (SFT-III) for the invese sin transform IST:

$$f_j = 2 \sum_{m=1}^{N_1-1} \hat{f}_m \sin\left[\pi(j+1/2)m/N_1\right]. \tag{3.5}$$

The basis function $\sin\left[\pi(j+1/2)m/N_1\right]$ is odd around $j = -0.5$ and $j = N_1 - 0.5$ as illustrated in Fig. .. (see notebook-) for $\hat{f}_0$ and $\hat{f}_1$ modes. In our simulation we store $f_0, f_1, ...f_{N_1-2}, f_{N_1-1}$ variables in real space. Note that neither f_0 nor f_{N_1-1} are zero because of our choice of basis functions.

We use RODFT10 (SFT-II) for the forward sin transform ST:

$$\hat{f}_m \quad = \quad 2 \sum_{j=0}^{N_1-1} f_j \sin\left[\pi(j+1/2)m/N_1\right] \tag{3.6}$$

for $m = 1, .., N_1 - 1$. In the Fourier (sin) space we store $m = 0 : (N_1 - 1)$ in the arrays $f(1), ..., f(N_1 - 1)$. The array $f(0)$ is set to zero is unused.

Note that our sin transforms are small modification of FFTW's sin transform. In FFTW transforms, the Fourier space variables $\hat{f}_m$ have range from $m = 1 : N_1$, and they are stored in $f(0), .., f(N_1 - 1)$. We achieve our transform by shifting right the FFTW sin transform output by one. To perform inverse sin transform, we shift left the variables $f(i)$s (stored in our notation) by one, and then apply FFTW's SFT-IIT. The result then is the variables in real space.

We use FFTW's REDFT01 (DCT-III) for the inverse cosine transform ICFT

$$g_j = \hat{g}_0 + 2\sum_{m=0}^{N_1-1} \hat{g}_m \cos\left[\pi(j+1/2)m/N_1\right]. \tag{3.7}$$

The basis function $\cos\left[\pi(j+1/2)m/N_1\right]$ is even around $j = -0.5$ and $j = N_1 - 0.5$ as illustrated in Fig. ..(see notebook) for $\hat{g}_1$ and $\hat{g}_2$ modes. We store $g_0, g_1, ..., g_{N_1-2}, g_{N_1-2}$ variables in real space. For the forward cos transform we use REDFT10 (DCT-II)

$$\hat{g}_m \quad = \quad 2\sum_{j=0}^{N_1-1} g_j \cos\left[\pi(j+1/2)m/N_1\right]. \tag{3.8}$$

with $m = 0, 1, ..., N_1 - 1$. These variables are stored in $g(0), g(1), ..., g(N_1 - 1)$. For the cos transforms, there is no need to shift the variables because the Fourier space variables $\hat{g}_m$s $(m = 0 : (N_1 - 1))$ are stored in arrays $g(0), ..., g(N_1 - 1)$.

3.3 Functions

In Chapter ... we will combine sin and cos transform defined here to Fourier transform along the perpendicular directions. We refer to these transforms as sincosFour transforms (SCFT). Most of the functions have been designed for these types of transforms that are useful for simulation Rayleigh Bénard. Here we list some of the functions defined in SCFT that performs sin or cos transform in 1D.

- `void Init_fftw_plan_SCFT(int NN[], Array<complx,1> A)`: Inializes fftw plans sintr_plan_SCFT, costr_plan_SCFT, isintr_plan_SCFT, icostr_plan_SCFT for 1D array.
- `void Norm_SCFT(int N[], Array<complx,1> A)`: Normalization of array A by $2N[1]$ during forward SCFT.
- `void ArraySFT_SCFT(fftw_plan sintr_plan_SCFT, int N[], Array<DP,1> A)`: Performs forward sin transform of a one-dimensional array A in three steps. First uses FFTW's sin tranform function, then shifts right the contents of array A, and finally divides the array by $2N[1]$ (normalization).
- `void ArrayCFT_SCFT(fftw_plan costr_plan_SCFT, int N[], Array<DP,1> A)`: Performs forward cos transform using FFTW function, and normalize the array by dividing by $2N[1]$.
- `void ArrayISFT_SCFT(fftw_plan isintr_plan_SCFT, int N[], Array<DP,1> A)`: Shift left the array A and apply inverse sin transform on A using FFTW function.

- `void ArrayICFT_SCFT(fftw_plan icostr_plan_SCFT, int N[], Array<DP,1> A)`: Performs inverse cos transform on A using FFTW function.

In the next chapter we will discuss mixed transform in which we combine sin/cos transform along x axis with Fourier transform along the perpendicular directions.

Exercise

1. Derive a formula for the forward sin transform given the formula for the invese sin transform, both for continuous and discrete transforms (Eqs. (4.2, 4.1, 3.5, 3.6).
2. Derive a formula for the forward cos transform given the formula for the invese cos transform, both for continuous and discrete transforms (Eqs. (4.2, 4.1, 3.7, 3.8).
3. Derive an expression for the energy of an odd (even) function in terms of sin (cos) transforms.

Chapter 4

SinCosFourier Transform

A library named libsincosfourier.a contains functions that performs sin/cos transforms along x direction, and Fourier transform along the other directions. The functions are written for 2D and 3D.

4.1 Definitions

4.1.1 SinFourier Transform

Consider a periodic function $f(x, \mathbf{y})$. We assume that along x, the function is odd around $x = 0$ and has period $2L_x$. The period along the other directions is L_s. We perform sin transform along x and Fourier transform along the perpendicular directions ($\mathbf{y}$ space). The mixed sin transform is defined as

$$\begin{aligned} f(x,\mathbf{y}) &= \sum_{K_x,\mathbf{K}} \hat{f}_{K_x,\mathbf{K}} 2\sin(K_x x)\exp(i\mathbf{K}\cdot\mathbf{y}) \\ \hat{f}_{K_x,\mathbf{K}} &= \frac{1}{2\prod L_s}\int dx d\mathbf{y} f(x,\mathbf{y}) 2\sin(K_x x)\exp(-i\mathbf{K}\cdot\mathbf{y}) \end{aligned}$$

where $K_x = m\pi/L_x$ ($m \geq 0$) is the wavevector along x, and $\mathbf{K}$ is the wavevector in the perpendicular space.

We discretize the space with N_s segments along the sth direction. Hence $x_s = j_s L_s/N_s$ with $j_s = (0, N_s - 1)$. The component of wave vector along x direction is $K_x = m\pi/L_x = m \times \text{kfactor}(1)$ with $m > 0$, and along the perpendicular direction it is $K_s = 2\pi k_s/L_s = k_s \times \text{kfactor}(s)$ with $\text{kfactor}(1) = \pi/L_x$ and $\text{kfactor}(s) = 2\pi/L_s$ for $s \neq 1$. As a result of descretization, the above equations become

$$f_{\mathbf{j}} = \sum_{m,\mathbf{k}} \hat{f}_{m,\mathbf{k}} 2\sin\left(\pi\frac{mj_x}{N_x}\right)\exp\left(2\pi i\sum_{s>1}\frac{j_s k_s}{N_s}\right), \tag{4.1}$$

$$\hat{f}_{m,\mathbf{k}} = \frac{1}{2\prod N_s}\sum_{\mathbf{j}} f_{\mathbf{j}} 2\sin\left(\pi\frac{mj_x}{N_x}\right)\exp\left(-2\pi i\sum_s \frac{j_s k_s}{N_s}\right), \quad (4.2)$$

where $\mathbf{j}$ and $\mathbf{k}$ are a vector comprising of j_s and k_s respectively. These formulas are called *Discrete Sin Fourier Transforms* (DSFT).

We can rewrite Eq. (4.1) as

$$f_{\mathbf{j}} = \sum_{m,\mathbf{k}} \frac{\hat{f}_{m,\mathbf{k}}}{i}\left[\exp\left(2\pi i\frac{mj_x}{2N_x}\right) - cc\right]\exp\left(2\pi i\sum_{s>1}\frac{j_s k_s}{N_s}\right).$$

Hence, the sin transform along the x direction is related to the Fourier transform in a box twice as big, as discussed in Chapter 2. As discussed in that chapter, we can convert the coefficients of SFT to those of Fourier transform. Also note that for real functions $f(x,\mathbf{y})$

$$\hat{f}_{m,-\mathbf{k}} = \left(\hat{f}_{m,\mathbf{k}}\right)^*.$$

Hence, in Fourier space we store only modes with $k_{lastindex} \geq 0$.

In Rayleigh Benard convection we choose $L_x = 1$, $L_y = 2\sqrt{2}$, and $L_z = 2\pi/q$. Hence kfactor(1) $= \pi$, kfactor (2) $= 2\pi/(2\sqrt{2}) = \pi/\sqrt{2}$, and kfactor(3) $= q$. Consequently $K_x = m\pi$ $(m > 0)$, $K_y = k_y\pi/\sqrt{2}$, and $K_z = qk_z$.

4.1.2 CosFourier Transform

Consider a periodic function $f(x,\mathbf{y})$ that is even around $x = 0$ along the x axis with a period of $2L_x$. The period along the other directions is L_s. We perform cos transform along x axis and Fourier transform along the perpendicular directions ($\mathbf{y}$ space). The mixed cos transform is defined as

$$\begin{aligned} g(x,\mathbf{y}) &= \sum_{\mathbf{K}} \hat{g}_{0,\mathbf{K}}\exp(i\mathbf{K}\cdot\mathbf{y}) + \sum_{K_x,\mathbf{K}} \hat{g}_{K_x,\mathbf{K}} 2\cos(K_x x)\exp(i\mathbf{K}\cdot\mathbf{y}) \\ \hat{g}_{K_x,\mathbf{K}} &= \frac{1}{2\prod L_s}\int dx d\mathbf{y} g(x,\mathbf{y}) 2\cos(K_x x)\exp(-i\mathbf{K}\cdot\mathbf{y}) \end{aligned}$$

where $K_x = m\pi/L_x$ $(m \geq 0)$ is the wavevector along x, and $\mathbf{K}$ is the wavevector in the perpendicular space. Using the same discretization procedure as described above, the above equations become

$$\begin{aligned} g_{\mathbf{j}} &= \sum_{\mathbf{k}} \hat{g}_{0,\mathbf{k}}\exp\left(2\pi i\sum_{s>1}\frac{j_s k_s}{N_s}\right) \\ &\quad + \sum_{m,\mathbf{k}} \hat{g}_{m,\mathbf{k}} 2\cos\left(\pi\frac{mj_x}{N_x}\right)\exp\left(2\pi i\sum_{s>1}\frac{j_s k_s}{N_s}\right), \quad (4.3) \\ \hat{g}_{m,\mathbf{k}} &= \frac{1}{2\prod N_s}\sum_{\mathbf{j}} g_{\mathbf{j}} 2\cos\left(\pi\frac{mj_x}{N_x}\right)\exp\left(-2\pi i\sum_s\frac{j_s k_s}{N_s}\right), \quad (4.4) \end{aligned}$$

These formulas are called *Discrete Cosine Fourier Transforms* (DCFT). We can rewrite Eq. (4.3) as

$$g_{\mathbf{j}} = \sum_{\mathbf{k}} \hat{g}_{0,\mathbf{k}} \exp\left(2\pi i \sum_{s>1} \frac{j_s k_s}{N_s}\right) + \sum_{m,\mathbf{k}} \hat{g}_{m,\mathbf{k}} \left[\exp\left(2\pi i \frac{m j_x}{2N_x}\right) + cc\right] \exp\left(2\pi i \sum_{s>1} \frac{j_s k_s}{N_s}\right).$$

Hence, cos transform along x direction is related to the Fourier transform along x in a box twice as big. As described above, for real $g(x, \mathbf{y})$, $\hat{g}_{m,-\mathbf{k}} = conj(\hat{g}_{m,\mathbf{k}})$.

4.1.3 Energy computations

The energy per unit volume for the function $f(x, \mathbf{y})$ (sin transform) is

$$E = \frac{1}{\prod L_i} \frac{1}{2} \int dx d\mathbf{y} |f(x, \mathbf{y})|^2,$$

which can be shown to be equal to

$$E = \sum_{m>0} \left|\hat{f}_{m,\mathbf{k}}\right|^2$$

energy per unit volume for the function $g(x, \mathbf{y})$ can be computed similarly:

$$E = \frac{1}{2} |\hat{g}_{0,\mathbf{k}}|^2 + \sum_{m>0} |\hat{g}_{m,\mathbf{k}}|^2 .$$

As discussed in Chapter 2, the cos and sin transforms contain the negative wavenumber Fourier modes. That is the reason why the factor of 1/2 does not appear the formulas for the sin and cos transformed variables.

We define isotropic energy spectrum based on the above energy formulas. We define the isotropic energy spectrum $e(K)$ of the field as the energy contained in the wavenumber shell K:

$$e(K) = \frac{1}{2} \sum_{K \le K' < K} |\hat{f}_{m,\mathbf{K}}|^2,$$

where the wavenumber of a mode $\hat{f}_{m,\mathbf{K}}$ is computed using

$$K' = \left[(m \times \text{kfactor}(1))^2 + \sum (k_s \times \text{kfactor}(s))^2\right]^{1/2} .$$

The function $e(K)$ contains the energy of the Fourier variables on the inner surface of the shell, but not that of the outer surface of the shell.

We also compute function

$$S_n = \sum_{m>0} (K_x^n + K_s^n) \left|\hat{f}_{m,\mathbf{k}}\right|^2 .$$

For Prandtl number greater than one, we have $K_x = m\pi$ and $K_s = k_s\pi/\sqrt{2}$ in two dimensions. Therefore,

$$S_n = \sum_{m>0} \left[(m\pi)^2 + (k_s\pi/\sqrt{2})^2\right] \left|\hat{f}_{m,\mathbf{k}}\right|^2 .$$

4.2 Implementation

For real fields, the SCFT transforms satisfy the property:

$$\hat{f}_{m,-\mathbf{k}} = \left(\hat{f}_{m,\mathbf{k}}\right)^* ; \hat{g}_{m,-\mathbf{k}} = (\hat{g}_{m,\mathbf{k}})^*$$

Therefore, only half of the SCFT modes need to be stored. Specifically

For a 2D real array $f(N_1, N_2)$, FFTW allocates $N_1 \times (N_2/2+1)$ dimensional complex array. In real space the variables are stored up to $(0 : N_1-1, 0 : N_2-1)$, however in SCFT space the variables with wavenumbers $m = 0 : (N_1 - 1)$ and $k_y = 0 : N_2/2$ are stored. Note however that in sinFour transform, $\hat{f}_{0,k_y} = 0$. See Fig... for illustration.

For a 3D real array $f(N_1, N_2, N_3)$, FFTW allocates $N_1 \times N_2 \times (N_3/2 + 1)$ dimensional complex array. The real space the variables are stored up to $(0 : N_1 - 1, 0 : N_2 - 1 : N_{3-1})$. In SCFT space the wavenumbers of the stored variables are $m = 0 : (N_1 - 1)$, $\mathbf{k} = (0 : N_2 - 1, 0 : N_3/2)$. Here too for sinFourier transform $\hat{f}_{0,k_y,k_z} = 0$. See Fig. .. for illustration of the storage mechanism. In Fig.... we illustrate one of the $k_y - k_z$ planes. The variable $\hat{f}_{m,-k_y-k_z}$ is not stored in the array since it is related to $\hat{f}_{m,k_yk_z}$ with relation

$$\hat{f}_{m,-k_y-k_z} = \left(\hat{f}_{m,k_y,k_z}\right)^* .$$

We implement SCFT functions using FFTW functions. The sin and cos transforms are implemented using RODFT10 (SFT-II), RODFT01 (SFT-III), REDFT10 (DCT-II), and REDFT01 (DCT-III) respectively as described in Chapter..., and the Fourier transform is implemented using the FFTW functions.

In FFTW convention the cosine transformed variables are stored for $m = 0 : N_1 - 1$, but the sin transformed variables are stored for $m = 1 : N_1$. In our scheme we store only $m = 0 : N_1 - 1$ for both sin and cos variables (of course $m = 0$ mode is zero for sin transform). Our scheme is advantangeous for spectral simulations in many ways, e.g., while taking derivatives. To implement our scheme we shift right the sin-transformed variables from FFTW function by one, and set $\hat{f}_{0,\mathbf{k}}$ to zero. Before performing the inverse sin transform using FFTW functions, we need to shift left the variables that are stored according to our scheme.

We implement the forward SCFT by performing sin or cosine transform of along all the rows, and then performing FFT along the vertical column (2D) or along the vertical planes (3D). For the inverse transform, we first perform IFFT along the vertical direction, and then sin/cos transform along the horizontal rows.

[CHANGE WORDING] In the energy computation, the sum over the wavenumber is for all the wavenumbers. Since we store wavenumber modes with $k_{lastimdex} \geq 0$ in our simulation, we double the contributions of all those modes for which a corresponding complex conjugate is not stored. The calculation of isotropic energy spectrum is done by summing the energy over the shells. Some of the

shells are present only partially in our simulation box. For these shells we compute average energy per mode in the shell from the modes present in the shell, and then multiply the average energy per mode by the volume of the shell. In 2D, the volume factor is $\pi K/2$, while in 3D the factor is πK^2. Note that our wavenumber summation is over the modes with $k_{lastindex} \geq 0$.

In the following sections we will describe the functions defined in sincosfour library.

4.3 Functions

We have divided the functions in to two classes- one related to the definitons of the transforms and the derivative of the functions (in sincosfour.cc), and the second of functions are related to the computations of energy, energy spectrum etc. (in field_internal_sincosfour.cc) Firstly we describe functions in sincosfour.cc

4.3.1 sincosfour.cc

The file sincosfour.cc contains functions related to sin, cosine, and mixed (sin/cos+Fourier) transforms. The external `fftw_plan` variables `r2c_SCFT, c2r_SCFT` perform forward and inverse Fourier transforms, while the fft_plan variables `sintr_SCFT, costr_SCFT, isintr_SCFT, icostr_SCFT` perform forward and inverse SFT and CFT. These variables defined appropriately according to the dimensionality D.

- For 1D:
  - `void Init_fftw_plan_SCFT(int NN[], Array<complx,1> A)`: Inializes fftw plans sintr_plan_SCFT, costr_plan_SCFT, isintr_plan_SCFT, icostr_plan_SCFT for 1D array.
  - `void Norm_SCFT(int N[], Array<complx,1> A)`: Normalization of array A by $2N[1]$ during forward SCFT.
  - void ArraySFT_SCFT(fftw_plan sintr_plan_SCFT, int N[], Array<DP,1> A): Performs forward sin transform of a one-dimensional array A in three steps. First uses FFTW's sin tranform function, then shifts right the contents of array A, and finally divides the array by $2N[1]$ (normalization).
  - void ArrayCFT_SCFT(fftw_plan costr_plan_SCFT, int N[], Array<DP,1> A): Performs forward cos transform using FFTW function, and normalize the array by dividing by $2N[1]$.
  - void ArrayISFT_SCFT(fftw_plan isintr_plan_SCFT, int N[], Array<DP,1> A): Shift left the array A and apply inverse sin transform on A using FFTW function.
  - void ArrayICFT_SCFT(fftw_plan icostr_plan_SCFT, int N[], Array<DP,1> A): Performs inverse cos transform on A using FFTW function.

- Derivative computation like 2D and 3D functions described below.

- For 2D and 3D
  - `void Init_fftw_plan_SCFT(int NN[], Array<complx,n> A)`: Inializes fftw plans sintr_plan_SCFT, costr_plan_SCFT, isintr_plan_SCFT, icostr_plan_SCFT (sin and cos transforms along x axis), and r2c_plan_SCFT, c2r_plan_SCFT (Fourier transforms along the perpendicular directions).
  - `void Zero_pad_SCFT_lastrow(int N[], Array<complx,n> A)`: In 2D, the row $k_y = N_2/2$ is set to zero; In 3D, the plane $k_3 = N_3/2$ is set to zero.
  - `void Norm_SCFT(int N[], Array<complx,n> A)`: Normalization of array A by $2\prod N_i$ during forward SCFT.
  - `void Sintr_row_SCFT(fftw_plan sintr_plan_SCFT, int N[], Array<DP,1> Row)`: Sin transform of data stored in row. `Costr_...`, `Isintr_...`, `Icostr_...` perform cos, Invere sin, and Inverse cosine transforms respectively using corresponding FFTW plans.
  - `void FT_Col_SCFT(fftw_plan r2c_plan_SCFT, int N[], Array<complx,1> Col)`: Fourier transform along perpendicular column using data stored in col. `FT_plane_...`, `IFT_Col_...`, `IFT_plane_...` perform Fourier transform along the plane, Inverse Fourier transform along columns and planes respectively.
  - void ArrayShiftRight_SCFT(int N[], Array<complx,n> A): Shifts right the contents of array A along x axis by one. That is, $A(i,..) \rightarrow A(i+1,..)$. This operation is performed after FFTW's forward sin transform.
  - void ArrayShiftLeft_SCFT(int N[], Array<DP,n> A): Shifts left the contents of array A along x axis by one. That is, $A(i,..) \rightarrow A(i-1,..)$. This operation is performed before FFTW's inverse sin transform.
  - `void ArraySFT_SCFT(fftw_plan sintr_plan_SCFT, fftw_plan r2c_plan_SCFT, int N[], Array<complx,2/3> A)`: Perfoms sin transform along rows; Zeropad last row; Forward Fourier transform along column/planes; Normalize by `SCFT_norm`, i.e., divide by $\prod N_i$; Shift array A to the right by one column/plane; Put zeros in the first colum/plane.
  - `void ArrayISFT_SCFT(fftw_plan isintr_plan_SCFT, fftw_plan c2r_plan_SCFT, int N[], Array<complx,2> A)`: Shift array A to the left; Perfoms Inverse Fourier transform along perpendicular column/plane; sin transform along the row.
  - `void ArrayCFT_SCFT(fftw_plan costr_plan_SCFT, fftw_plan r2c_plan_SCFT, int N[], Array<complx,2> A)`: Do the same operations and `ArraySFT_SCFT`. However no array shift is applied.

- `void ArrayICFT_SCFT(fftw_plan icostr_plan_SCFT, fftw_plan c2r_plan_SCFT, int N[], Array<complx,2> A)`: Same as `ArrayISFT_SCFT` withour array_shift.
- `void Xderiv_SCFT_SIN(int NN[], Array<complx,n> A,Array<complx,n> B, DP kfactor[])`: Array A contains sin transformed function. Hence $B(m, \mathbf{k}) = m \times \text{kfactor}(1) \times A(m, \mathbf{k})$. Array B contains cos transformed function.
- `void Xderiv_SCFT_COS(int NN[], Array<complx,n> A,Array<complx,n> B, DP kfactor[])`: Array A contains cos transformed variable. Hence $B(m, \mathbf{k}) = -m \times \text{kfactor}(1) \times A(m, \mathbf{k})$. Array B contains sin transformed function.
- `void Yderiv_FFT(int NN[], Array<complx,n> A, Array<complx,n> B, DP kfactor[])`: $B(\mathbf{k}) = iK_yA(\mathbf{k})$ with $K_y = \text{kfactor}(2) \times k_y$.
- `void Zderiv_FFT(int NN[], Array<complx,n> A,Array<complx,n> B, DP kfactor[])`: $B(\mathbf{k}) = iK_zA(\mathbf{k})$ with $K_z = \text{kfactor}(3) \times k_z$.

4.4 sincosfour_inline.h

- `inline int Compute_ky3D_SCFT(int i2, int N[])`: Returns k_y given i_y, i. e., $k_y = i_y$ if $i_y \leq N_2/2$, else $k_y = i_y - N_2$. This function is useful for 3D arrays. In 2D, $k_y = i_y$.
- `inline int Compute_iy3D_SCFT(int ky, int N[])`: Computes i_y given k_y in 3D.
- `inline DP Kmagnitude_SCFT(int i1, int i2, int N[], DP kfactor[])`: Computes kmagnitude $\sqrt{\sum(\text{kfactor}(i) \times k_i)^2}$. Similar operation for 3D.
- inline int Min_radius_outside_SCFT(int N[], DP kfactor[]): Returns radius of the smallest sphere enclosing the cube $(\text{kfactor}(1)(N_1-1), \text{kfactor}(2)N_2/2)$ in 2D and $(\text{kfactor}(1)(N_1-1), \text{kfactor}(2)N_2/2, \text{kfactor}(3)N_3/2)$ in 3D.
- inline int Max_radius_inside_SCFT(int N[], DP kfactor[]): Returns radius of the largest sphere that can fit inside the cube $(\text{kfactor}(1)(N_1-1), \text{kfactor}(2)N_2/2)$ in 2D and $(\text{kfactor}(1)(N_1-1), \text{kfactor}(2)N_2/2, \text{kfactor}(3)N_3/2)$ in 3D.

4.5 field_internal_four.cc

This file contains functions that operate on $\hat{f}(m, \mathbf{k})$ and $\hat{g}(m, \mathbf{k})$. They compute energy spectrum etc. To compute these quantities, the sum is done over all the revelant wavenumbers. However we store only modes with $k_{lastindex} \geq 0$ in computer simulation. Therefore we double the contributions from most of the modes; the modes at the common surface $k_{lastindex} = 0$ is not to be doubled. Here n takes values 2 or 3.

- `DP Total_abs_sqr_SCFT(int N[], Array<complx,n> A)`: Returns total fluctuating energy $|\hat{A}(m,\mathbf{k})|^2$ (without $\mathbf{k}=0$).

- `complx Total_termbyterm_mult_SCFT(int N[], Array<complx,n> A, Array<complx,n> B)`: Returns $\hat{A}(m,\mathbf{k}) \times conj(\hat{B}(m,\mathbf{k}))$ without $\mathbf{k}=0$.

- `DP Total_Sn_SCFT(int N[], Array<complx,n> A, DP n, DP kfactor[])`: Returns total fluctuating quantity $k^n|\hat{A}(m,\mathbf{k})|^2$ (without $\mathbf{k}=0$).

- void Compute_1D_Sk_SCFT(int N[], Array<DP,1> A, DP n, Array<DP,1> Sk, DP kfactor[]): Compute $Sk(k) = K^n|\hat{A}(m)|^2/2$.

- void Compute_1D_Sk_SCFT(int N[], Array<DP,1> A, Array<DP,1> B, DP n, Array<DP,1> Sk, DP kfactor[]): Computes $Sk(k) = K^n\Re(A(k)*conj(B(k)))/2$.

- `void Compute_isotropic_Sk_SCFT(int N[], Array<complx,n> A, DP n, Array<DP,1> Sk, DP kfactor[])`: For $D = 2, 3$ isotropic spectrum $Sk(K) = \sum K'^n|\hat{A}(m,\mathbf{k}')|^2/2$ with $K \le K' < K+1$. $Sk_{n=0}(K)$ is the energy spectrum with $k=0$ providing the mean energy.

- `DP Shell_termbyterm_mult_SCFT(int N[], Array<complx,n> A, Array<complx,n> B, DP inner_radius, DP outer_radius, DP kfactor[])`: Returns $\sum_K \Re(A(K)*conj(B(K)))$ `for inner_radius` $\le K <$ `outer_radius .`

- `void Shell_termbyterm_mult_real_imag_SCFT(int N[], Array<complx,n> A, Array<complx,n> B, DP inner_radius, DP outer_radius, DP kfactor[], DP& total_real, DP& total_imag)`: Computes total_real $= \Re(A(K)*\Re(B(K)$, and total_imag $= \Im(A(K))*\Im(B(K))$ for `inner_radius` $\le K <$ `outer_radius .`

- `void Array_divide_ksqr_SCFT(int N[], Array<complx,n> A, DP kfactor[])`: $A(\mathbf{k}) \to A(\mathbf{k})/K^2$, $A(0) = 0$. Note that $K^2 = K_x^2 + \sum_s K_s^2$.

- `void Array_mult_ksqr_SCFT(int N[],Array<complx,n> A, DP kfactor[])`: $A(\mathbf{k}) \to K^2 A(\mathbf{k})$.

- `void Array_exp_ksqr_SCFT(int N[],Array<complx,n> A, DP factor, DP kfactor[])`: $A(\mathbf{k}) \to \exp(factor \times K^2)A(\mathbf{k})$.

Chapter 5

Basic array functions

We define some basic array function that are very useful. They are

- void Array_real_mult(int N[], Array<complx,1> A, Array<complx,1> B, Array<complx,1> C): Multiply term by term, i.e., $C(i) = A(i) \times B(i)$.
- void Output_asreal(ofstream& fileout, int N[], Array<complx,n> A): Outputs real and imaginary parts element by element, i.e., $\Re[A(1)]$, $\Im[A(1)]$, $\Re[A(2)]$, $\Im[A(2)]$, ...

These functions are in basic_array.h

Chapter 6

Universal functions

For each `basis_type` we have various functions like ArrayFFT_FOUR(..) in FOUR basis peforms forward Fourier transform. In SCFT basis, the forward sin-Fourier transform is performed by ArraySFT_SCFT(..). While simulating flows we invoke these functions depending on the choice of basis_type. To make the functions used in flow simulation more general, we have defined *universal functions* that invoke particular functions depending on the `basis_type`. For exam*ple* the function ,

`void Forward_transform_array(string basis_type, int N[], Array<complx,n> A, int parity)`

perfoms inplace Forward Transform on array A depending on basis_type.

Most of the univesal functions are straight-forward apart from a special variable named `parity`. For `basis_type` SCFT we have either cos or sin transform. We adopt a convection that `parity` = 0 is even parity and applies to cos transform, and `parity` = 1 is odd parity and applies to sin transform. Also we take `parity` = 0 if there is no choice to make, e.g., for FOUR basis type.

6.1 universal_inline.h

We define a constant `const DP INF_RADIUS` = 10000 that is supposed to represent the infinite radius of the wavenumber sphere. This constant is useful in functions involving energy transfer computations.

In the following discussion we describe the universal functions that use basis_type to yield the required quantities.

- `inline int Get_kx(string basis_type, int i1, int N[])`: Returns k_x given i_1.
- `inline int Get_ky3D(string basis_type, int i2, int N[])`: Returns k_y given i_2.
- `inline DP Kmagnitude(string basis_type, int i1, int N[], DP kfactor[])`: Returns kmagnitude kfactor(1) $\times$ k_1 in 1D.

- `inline DP Kmagnitude(string basis_type, int i1, int i2, int N[], DP kfactor[])`: Returns kmagnitude $\sqrt{\sum(\text{kfactor}(i) \times k_i)^2}$ in 2D.
- `inline DP Kmagnitude(string basis_type, int i1, int i2, int i3, int N[], DP kfactor[])`: Returns kmagnitude $\sqrt{\sum(\text{kfactor}(i) \times k_i)^2}$ in 3D.
- `inline int Get_ix(string basis_type, int kx, int N[])`: Returns i_1 given k_x.
- `inline int Get_iy3D(string basis_type, int ky, int N[])`: Returns i_y given k_y.
- `inline int Min_radius_outside(string basis_type, int N[], DP kfactor[])`: Returns the radius of the smallest sphere containing wavenumber grid N.
- `inline int Max_radius_inside_FOUR(string basis_type, int N[], DP kfactor[])`: Returns the radius of the largest sphere that can fit inside the wavenumber grid N.
- `inline complx DxVx(string basis_type, DP kfactor[], int kx, complx Vx)`: Computes $D_x(V_x)$. The result is iK_xV_x for FOUR and K_xV_x for SCFT. Note that V_x has odd parity in SCFT basis.
- `inline complx DxVx_plus_DyVy(string basis_type, DP kfactor[], int kx, int ky, complx Vx, complx Vy)`: Computes $D_xV_x + D_yV_y$.

6.2 universal_fn.cc

- `void Init_fftw_plan_array(string basis_type, int N[], Array<complx,n> A)`: Initializes fftw_plans.
- `void Forward_transform_array(string basis_type, int N[], Array<complx,n> A, int parity)`: Inplace Forward Transform(A).
- `void Inverse_transform_array(string basis_type, int N[], Array<complx,n> A, int parity)`: Inplace Inverse Transform(A).
- `void Xderiv(string basis_type, int N[], Array<complx,n> A, Array<complx,n> B, DP kfactor[])`: $B(\mathbf{k}) = \mathcal{F}[\partial A(\mathbf{x})/\partial x]$, where $\mathcal{F}$ is the Forward transform operation.
- `void Yderiv(string basis_type, int N[], Array<complx,n> A, Array<complx,n> B, DP kfactor[])`: $B(\mathbf{k}) = \mathcal{F}[\partial A(\mathbf{x})/\partial y]$.
- `void Zderiv(string basis_type, int N[], Array<complx,n> A, Array<complx,n> B, DP kfactor[])`: $B(\mathbf{k}) = \mathcal{F}[\partial A(\mathbf{x})/\partial z]$
- `DP Total_abs_sqr(string basis_type, int N[], Array<complx,n> A)`: Returns the fluctuating energy $\sum |A(\mathbf{k})|^2/2$ except $\mathbf{k} = 0$.

- `DP Total_Sn(string basis_type, int N[], Array<complx,n> A, DP n, DP kfactor[])`: Returns the fluctuating energy $\sum K^n |A(\mathbf{K})|^2/2$ except $\mathbf{k} = 0$.

- `DP Total_termbyterm_mult(string basis_type, int N[], Array<complx,n> A, Array<complx,n> B)`: Returns the fluctuating energy $\Re\left[\sum A(\mathbf{k}) \times B(\mathbf{k})^*\right]/2$ except $\mathbf{k} = 0$.

- `void Compute_1D_Sk(string basis_type, int N[], Array<complx,1> A, DP n, Array<DP,1> Sk, DP kfactor[])`: For 1D array A, the function computes $Sk(A,k) = (1/2)K^n|A(k)|^2$ where $K = k \times \text{factor}(1)$.

- `void Compute_1D_Sk(string basis_type, int N[], Array<complx,1> A, Array<complx,1> B, DP n, Array<DP,1> Sk, DP kfactor[])`: For 1D array A, the function computes $Sk(A,k) = (1/2)K^n\Re\left[A(k) \times B(k)^*\right]$ where $K = k \times \text{factor}(1)$.

- `void Compute_isotropic_Sk(string basis_type, int N[], Array<complx,n> A, DP n, Array<DP,1> Sk, DP kfactor[])`: For $D = 2$ and 3 computes $Sk(K) = \sum[K'^n|A(K')|^2/2]$ with $K \leq K' < (K+1)$. $Sk(0)$ for $n = 0$ is the mean energy.

- `void Compute_isotropic_Sk(string basis_type, int N[], Array<complx,n> A, Array<complx,n> B, DP n, Array<DP,1> Sk, DP kfactor[])`: Computes $Sk(A,B,k) = (1/2)\sum K'^n\Re\left[A(K) \times B(K)^*\right]$ with $K \leq K' < (K+1)$.

- `void Array_mult_ksqr(string basis_type, int N[], Array<complx,n> A, DP kfactor[])`: $A(\mathbf{k}) = A(\mathbf{k})K^2$ where K is the waveumber for array index $\mathbf{k}$ computed using $K^2 = \sum_s \left(\text{kfactor}(s) \times k_s\right)^2$.

- `void Array_divide_ksqr(string basis_type, int N[], Array<complx,3> A, DP kfactor[])`: $A(\mathbf{k}) = A(\mathbf{k})/K^2$.

- `void Array_exp_ksqr(string basis_type, int N[], Array<complx,n> A, DP factor, DP kfactor[])`: $A(\mathbf{k}) = A(\mathbf{k}) \times \exp(\text{factor} \times K^2)$.

Library Classes

Chapter 7

Fields

In library libblitz.a we define functions for vector and scalar fields. In spectral simulation, the fields could be represented in either real or Fourier space. For this reason we define four different classes

We do this through four classes

- Complex vector field: CVF
- Complex scalar field: CSF
- Real vector fields: RVF
- Real scalar fields: RSF

Note however that for a given field, the array size in both real and Fourier space is the same. We implement the arrays using *blitz++* library. We define the above fields in dimensions one, two, and three.

The fields could be defined in Fourier, SinCosFourier, Chebychev-Fourier, or anyother basis. We define the functions of the above classes in terms of universal functions with a switch called `basis_type`. The functions of these classes remain general under this design. At present we have implemented Fourier (`basis_type=FOUR`) and SinCosFourier (`basis_type=SCFT`) basis.

In the following discussion we describe the classes in some detail.

7.1 Complex vector field CVF

CVF is the backbone of our program. They represent vector flow fields like velocity or magnetic fields in spectral space. A CVF comprises of complex *dynamic* arrays equal to number of dimensions, i.e., $*V1$ in 1D, $*V1, *V2$ in 2D, and $*V1, *V2, *V3$ in 3D as defined below.

```
class CVF {
 public:
```

```
#ifdef D1
 Array<complx,1> *V1;
#endif

#ifdef D2
 Array<complx,2> *V1;
 Array<complx,2> *V2;
#endif

#ifdef D3
  Array<complx,3> *V1;
  Array<complx,3> *V2;
  Array<complx,3> *V3;
#endif
```

CVF is initialized using constructor `CVF(int N[])`. Note that the wavenumber range in the ith direction is defined as $k_i = -N_i/2 + 1 : N_i/2$.

The other public variables of CVF are

- `D`: Dimensionality of the vector field.
- CV_basis_type: Takes values either as FOUR or SCFT (sincosFour).
- `*Ncv: Ncv[i]` contains size of Arrays `*Vi` in ith direction.
- *CV_kfactor[i]: The real wavevector K_i is related to the array index k_i: $K_i = k_i \times$ CV_kfactor(i).
- `*CV_ek`: In 1D, the energy contained in $V_1(k)$, i.e., $|V_1(k)|^2/2$.
- `*CV_dissk`: In 1D, $K^2|V_1(k)|^2/2$.
- `CV_iso_ek_size`: Size of isotropic $E(k)$, which is the radius of the smalles wavenumber sphere enclosing the simulation box in the wavenumber space.
- `*CV_iso_ek`i: Isotropic energy spectrum of the ith component of the vector $e_i(k) = \sum_{K \le K' < K+1} |V_i(K')|^2/2$.
- `*CV_iso_dissk:` Isotropic spectrum $\sum_{k \le k' < k+1} K'^2|V(K')|^2/2$; summed over all the components.
- `CV_totalenergy:` Total energy of the vector field, $\sum_{k'} |V(k')|^2/2$ summed over the whole spectral space.
- `CV_totaldiss:` $\sum_{k'} K'^2|V(k')|^2/2$ summed over the whole spectral space.
- `CV_entropy:` Entropy of vector field V.

The public functions defined for CVF are

- `CVF(int *NN, string basis_type, DP *kfactor)`: Constructor; $Ncv[] = NN[]$; Allocates memory to arrays V_i of size $NN[]$; Allocates memory to `CV_iso_Ek`i as a one dimensional array of size `CV_iso_ek_size`; Initalizes these arrays to zero. Also, `CV_basis_type = basis_type`, and `*CV_kfactor = *kfactor`.
- `void CV_Copy_to(CVF& to)`: Copy V_i's to CVF `to`.V_i.
- `void CV_Inialize()`: Sets `*Vi = 0`.
- `void Init_fftw_plan()`: Creates fftw_plan `r2c_plan_FOUR, c2r_plan_FOUR` for `basis_type FOUR`; and `r2c_plan_SCFT, c2r_plan_SCFT, sintr_plan_SCFT, costr_plan_SCFT, isintr_plan_SCFT, icostr_plan_SCFT` for `basis_type SCFT`.
- `void CV_Forward_Transform()`: Inplace forward transform of $V_i(\mathbf{x})$; The trasforms could be either FOUR or SCFT (sin/cos along x and Fourier along the perpendicular directions).
- `void CV_Inverse_Transform()`: Inplace Inverse FOUR or SCFT transform of $V_i(\mathbf{k})$.
- `void CV_output()`: Output all the components of the CVF.
- `void CV_Compute_totalenergy_diss()`: Computes `CV_totalenergy` = $\sum_K |V(K)|^2$ and `CV_totaldiss` = $\sum_K K^2|V(K)|^2$.
- `void CV_Compute_entropy()`: Computes entropy of the vector field.
- `void CV_Compute_1D_spectrum()`: Computes energy spectrum and dissipation for 1D.
- `void CV_Compute_isotropic_spectrum()`: Computes isotropic energy spectrum and and dissipation for 2D and 3D for Fourier and SCFT transformed CVF. $e_i(K) = \sum_{K \le K' < K+1} |V_i(K')|^2$ and $D(K) = \sum_{K \le K' < K+1} K'^2 |V(K')|^2$.

7.2 Complex Scalar Field (CSF)

CSF represents a scalar field like temperature, pressure, etc. in spectral space. A CSF comprises of complex *dynamic* arrays $*F$. The other public variables of CSF are

- `D`: Dimensionality of the vector field.
- CS_basis_type: Takes values either as FOUR or SCFT (sincosFour).
- `*Ncs: Ncv[i]` contains size of Arrays `*F` in ith direction.
- *CS_kfactor[i]: Connects real wavevector K_i with FFT index k_i using $K_i = k_i \times$ CS_kfactor(i).

- `*CS_ek`: In 1D, $e(k) = |F(k)|^2/2$.
- `*CS_dissk`: In 1D, $D(k) = K^2|F(k)|^2/2$.
- `CS_iso_ek_size`: same as CVF size variable.
- `*CS_iso_ek`: Isotropic energy spectrum of the scalar field $e(K) = \sum_{K \le K' < K+1} |F(K')|^2/2$.
- `*CV_iso_dissk`: Isotropic spectrum $D(K) = \sum_{k \le k' < k+1} K'^2|F(K')|^2/2$.
- `CS_totalenergy`: Total energy of the vector field, $\sum_{k'} |F(k')|^2/2$ summed over the whole spectral space.
- `CS_totaldiss`: $\sum_{K'} K'^2|F(K')|^2/2$ summed over the whole spectral space.

The public functions defined for CVF are

- `CSF(int *NN, string basis_type, DP *kfactor)`: Similar to CVF except that it creates `*F`.
- `void CS_Forward_Transform()`: Inplace forward transform of $F(\mathbf{x})$.
- `void CS_Inverse_Transform()`: Inplace Inverse transform of $F(\mathbf{k})$.
- `void CS_divide_ksqr()`: $F(k) = F(k)/K^2$ that is useful while computing pressure.
- `void CS_Compute_totalenergy_diss()`: Computes `CS_totalenergy` $= \sum_K |F(K)|^2$ and `CS_totaldiss` $= \sum_K K^2|F(K)|^2$.
- `void CV_Compute_1D_spectrum()`: Computes energy spectrum and dissipation for 1D.
- `void CV_Compute_isotropic_spectrum()`: Computes isotropic energy spectrum and and dissipation for 2D and 3D: $e(K) = \sum_{K'} |F(K')|^2$ and $D(K) = \sum_K K'^2|F(k')|^2$ where the sum is performed over $K \le K' < K + 1$.

7.3 Real Vector Field (RVF)

This class is for real vector field. We use this class for storing $\mathbf{V}(\mathbf{x})$ that is obtained aftering perforing inverse transform of $\mathbf{V}(\mathbf{k})$. The names of these variables are Vir where i takes on values 1,2, or 3. The functions in these class are

- `RVF(int *NN, string basis_type)`: Constructor with similar function as CVF constructor.

- `void RV_Forward_transform()`: Forward transform Vir.
- `void RV_inverse_transform()`: Inverse transform Vir.
- `void RV_outout(ofstream& fileout)`: Outputs $V_i(\mathbf{x})$ to file `fileout`.
- `void RV_Forward_transform_RSprod()`: The field components V_is are mulitplied in real space for computing the nonlinear terms in spectral simulation. After this multiplication, we have $V_1 = V_1V_2$ in 2D, and $V_1 = V_1V_2$, $V_2 = V_2V_3$, and $V_3 = V_3V_1$. Since V_1 has odd parity (sin), and other components have even parity (cos), V_1 has odd parity in 2D, However, in 3D V_1 and V_3 have odd parity, and V_2 has even parity. The function RV_Forward_transform_RSprod() performs
  - in 2D: $\mathcal{F}(V_1, \text{parity} = 1)$ where $\mathcal{F}$ is the forward transform operator.
  - in 3D: $\mathcal{F}(V_1, \text{parity} = 1)$, $\mathcal{F}(V_2, \text{parity} = 0)$, $\mathcal{F}(V_3, \text{parity} = 1)$

7.4 Real Scalar Field (RSF)

This class is for real scalar field. It contais real scalar field $Fr(\mathbf{x})$. The functions of RSF are

- `RSF(int *NN, string basis_type)`: Constructor with similar function as CSF constructor.
- `void RS_Forward_transform()`: Forward transform of Fr.
- `void RS_Inverse_transform()`: Inverse tranform of Fr.
- `void RS_Output(ofstream& fileout)`: Output Fr to file `fileout`.

Chapter 8

IncFlow: Library for Incompressible Flow

An incompressible flow consists of a flow field $\mathbf{V}$, its associated pressure p, and its diffusive parameters like ν. The evolution of the flow field requires the computations of its nonlinear terms. We declare a class IncVF that contains the contains the vector field $\mathbf{V}$ in Fourier and real spaces through inherited classes CVF and RVF. It also inherits class NLIN that contains nonlinear terms $\mathcal{F}[\partial_j(V_jV_i)]$ ($\mathcal{F}$:Forward transform) and pressure through an inherited CSF. See diagram for an illustration.

A flow field could interact with a scalar field like temperature. To handle scalar fields we define a class IncSF. Note that the nonlinear terms of scalar field are of the type $\mathcal{F}[\partial_j(V_jT)]$ that assumes incompressibility ($\partial_jV_j = 0$).

The vector field $\mathbf{V}$ could also interact with another vector field, e.g., magnetic field in magnetohydrodynamics. We compute the interactions between these fields through functions defined here.

In library libIncFlow.a we have the following classes:

- Nonlinear: `NLIN`.
- Incompressible Vector Field: `IncVF`
- Incompressible Scalar Field: `IncSF`
- Energy transfer vars and functions: `EnergyTr`

As discussed above these classes inherit some basic classes like `CVF, RVF` etc. We classes are defined for dimensions two, and three. Note that incompressible flow in one dimension is trivially a constant. Compiler directives are used to make a choice of dimensionality. The choice of basis function, FOUR or SCFT, is stored in variable `basic_type`.

8.1 Classes

The library IncFlow contains the following important classes.

8.1.1 Incompressible Vector Field: IncVF

`IncVF` inherits

- `CVF` that contains the field variables in Fourier space $[V_i(\mathbf{k})]$.
- `RSF` that contains the field variables in real space $[V_i(\mathbf{r})]$.
- `NLIN` that contains nlin$i$ that would contain $\mathcal{F}[\partial_j(V_j V_i)]$, and pressure field (an inherited CSF).
- `EnergyTr` that contains the arrays for wavenumber shells and spheres to store various energy transfers. This class will be discussed in the next chapter.

and contains public variables

- *`Force`i: Force acting on the vector field. The array size of `Force`i is the same as V_i.
- *`iso_force_Vk:` Contains $\sum_{K\leq K'<K+1} \mathbf{Force}(\mathbf{K}') \cdot [\mathbf{V}(\mathbf{K}')]^*$, i.e., the sum is over all the modes in the shell.
- `*iso_ek_cross:` Contains the cross helicity spectrum if $\mathbf{V}$ is interacting with another vector field $\mathbf{W}$. The size size of `iso_ek_cross` is the same as `CV_ek`.
- `D:` The dimensionality of space.
- `*N:` The size of the arrays V_i s
- `*kfactor:` Contains kfactor(s).
- `basis_type:` Basis type being used.
- `VF_temp[N[D]]: used` for temporary storage.
- `dissipation_coefficient:` Dissipation coefficient of the vector field appearing in front of $\nabla^2\mathbf{V}$.

There are many functions defined in the class `IncVF`. We categorize them in the following divisions:

- Constructor- `IncVF(int *NN, string prog_basis_type, DP *prog_kfactor, DP diss_coefficient, int shell_input_sheme, int nospheres, int noshells, Array<DP,1> Rshells)`: The first four arguments would be passed to the corresponding variables described above. The last four variables are related to the energy transfer that would be described in the next chapter.

- Compute nonlinear terms: Several functions compute the nonlinear term $\mathcal{F}[\partial_j(V_jV_i)]$, $\mathcal{F}[\partial_j(V_jF)]$, $\mathcal{F}[\partial_j(V_jW_i)]$ etc.
- Compute pressures
- Compute products of the field variables, e.g., V_iV_j.
- Useful function described in Sec.
- Functions related to energy transfers.

8.1.2 Class Nonlinear: NLIN

The function `NLIN` inherits a CSF that contains pressure $p(\mathbf{k})$. It also contains public variables

- nlin$i$: It contains $nlin_i(\mathbf{k})$.
- `D:` The dimensionality of space.
- `*Nn:` Size of `nlini` that is the same as V_i.
- `NLIN_basis_type`
- *`NLIN_kfacfor`

The functions in this class are

- NLIN(int *NN, string basis_type, DP *kfactor): Constructor
- void NLIN_diag_Forward_transform_derivative(): Before entering this function, $nlin_i$ contains either V_i^2 or V_iW_i. This function performs forward transform of $nlin_i$ and take derivative, i.e., $nlin_s \leftarrow D_s\mathcal{F}[nlin_i]$. In FOUR basis $D_s\mathcal{F}(f)$ is simply $iK_s\mathrm{FT}(f)$. However in SCFT, $D_1\mathcal{F}(f) = -$`kfactor`$(1) \times [$`CFT`$(f)]$ and $D_s\mathcal{F}(f) = iK_s[$`CFT`$(f)]$ `for` $s \geq 2$.
- void NLIN_Compute_divergence(): Before entering this function, $nlin_i = \mathcal{F}[\partial_j(V_jV_i)]$. This function computes the divergence of $nlin$ amd puts it in F, i.e., $F = D_i(nlin_i)$.

8.1.3 Class Incompressible Scalar Field: IncSF

The class `IncSF` inherits

- `CSF` that contains the field variables in Fourier space $F(\mathbf{k})$ like temperature
- `RSF` that contains the field variables in real space $F(\mathbf{r})$.

and contains public variables

- `nlin`: it contains $\mathcal{F}[\partial_j(V_jF)]$.

- *`Force`: Force acting on the scalar field.
- `SF_temp[Nn[D]]`: Used for temporary storage. The sizes of `nlin, Force,` and `SF_temp` are the same as V_i.
- *`iso_force_Vk`: Contains $\sum_{K\le K'<K+1} Force(\mathbf{K}') \cdot [F(\mathbf{K})]^*$
- `D`: The dimensionality of space.
- *`ISF_kfactor`
- *NIs
- `diffusion_coeff`icient: Diffusion coefficient of the scalar field appearing in front of $\nabla^2 F$.

Many functions in the class `IncVF` also act on the scalar variable of the class `IncSF`. Some useful functions of the class `IncSF` are described in Sec...

8.2 Forcing

The flow equations are typically forced. We have a force array for vector $*\mathbf{Force}_i$ for a vector field (defined in class `IncVF`), and $*Force$ for a scalar field (defined in class `IncSF`). The following functions are related to the forcing

- `void IncVF::Compute_force_spectrum()`: Computes Force spectrum $\sum_{K\le K'<K+1} \mathbf{Force}(\mathbf{K}') \cdot [\mathbf{V}(\mathbf{K}')]^*$. Note that there is no factor of $1/2$. The result stored in *`iso_force_Fk`.
- `void IncSF::Compute_force_spectrum()`: Computes Force spectrum $\sum_{K\le K'<K+1} Force(\mathbf{K}') \cdot [F(\mathbf{K})]^*$. Note that there is no factor of $1/2$. The result stored in *iso_force_Fk.*

8.3 Real Space products

The following functions compute products in real space.

8.3.1 Derivative of product V_iV_j

- `void Xderiv_RSprod(Array<complx,n> A, Array<complx,n> B)`: Computes $B(\mathbf{k}) = \mathcal{F}[\partial_x A(\mathbf{x})]$. We argue below that the derivative along x axis has odd parity.
- `void Yderiv_RSprod(Array<complx,n> A, Array<complx,n> B)`: $B(\mathbf{k}) = iK_yA(\mathbf{k})$.
- `void Zderiv_RSprod(Array<complx,n> A, Array<complx,n> B)`: $B(\mathbf{k}) = iK_zA(\mathbf{k})$.

8.3.2 Computation of V_iV_j in fluid simulation

- `void Compute_RSprod_diag()`: Computes the diagonal product terms $nlin_i = V_i^2$.
- `void Compute_RSprod_offdiag()`: Computes the off-diagonal product terms; In 2D: $V_1 = V_1V_2$; In 3D: $V_1 = V_1V_2$, $V_2 = V_2V_3$, and $V_3 = V_3V_1$.
- `void Derivative_RSprod()`: We need to compute $\mathcal{F}[D_j(V_jV_i)]$. Before entering the function, $\mathcal{F}[V_jV_i]$ has been computed. For y and z components, we multiply the above by $D_j = iK_j$. Since V_jV_1 $(j \neq 1)$ has odd parity, D_1 for the RSprod is alway of odd parity. Note that the terms $D_1[V_jV_1]$ appearing in the $nlin_j$ for V_j $(j \neq 1)$ has even parity, while $D_j[V_jV_1]$ appearing in $nlin_1$ has odd parity consistent with the fact that V_1 has odd parity.

8.3.3 Computation of V_iW_j in MHD simulation

- `void Compute_RSprod_diag(IncVF& W)`: Computes the diagonal product term $nlin_i = V_iW_i$.
- `void Compute_RSprod_offdiag(IncVF& W)`: Computes the off-diagnoal product term; In 2D: $V_1 = V_1W_2$ and $W_1 = W_1V_2$; In 3D: $V_1 = V_1W_2$, $V_2 = V_2W_3$, $V_3 = V_3W_1$, $W_1 = W_1V_2$, $W_2 = W_2V_3$, $W_3 = W_3V_1$.
- `void Derivative_RSprod(IncVF& W)`: We need to compute $nlin_i = D_j[\mathcal{F}(V_iW_j)]$, and $W.nlin_i = D_j[\mathcal{F}(W_iV_j)]$. Before entering the function, $\mathcal{F}[W_jV_i]$ has been computed. For y and z components, we multiply the above by $D_j = iK_j$. As argued above, the derivative along x direction has odd parity.

The most important function in IncVF is the computation of the nonlinear terms that is being described below.

8.4 Computation of nlin

8.4.1 IncVF::Compute_nlin()

This function computes $\mathcal{F}[\partial_j(V_jV_i)]$ and stores it in $nlin_i$. We perform this operation in several steps through various functions.

1. Compute $\mathbf{V}(\mathbf{r})$ by Inverse_transform (`RVF::RV_Inverse_transform`).
2. Diagonal product V_s^2 performed and put in $nlin_s$, i.e., $nlin_s \leftarrow V_s^2$ (`IncVF::Compute_RSprod_diag()`)
   The diagonal products have even parity.
3. Perform forward transform of V_i^2 and take derivative, i.e., $nlin_s \leftarrow D_s\mathcal{F}[V_s^2]$.
   Choose even parity derivative long x axis. The function is `NLIN::NLIN_diag_Forward_transform_der`

4. Compute off-diagonal products V_jV_i (`IncVF::Compute_RSprod_offdiag()`)
   (a) 2D: $V_1 = V_1V_2$. For SCFT, the product has odd parity.
   (b) 3D: $V_1 = V_1V_2$, $V_2 = V_2V_3$, $V_3 = V_3V_1$. For SCFT, the product V_1 and V_3 has odd parity, while V_2 has even parity.
5. void RV_Forward_transform_RSprod(): Forward transform the product V_is keeping parity in mind.
6. `IncVF::Derivative_RSprod()`: The products V_jV_1 have odd parity, hence D_1 for the RSprod is alway of odd parity.
   (a) 2D:
   $$\begin{aligned} nlin_1 &= nlin_1 + D_2V_1, \\ nlin_2 &= nlin_2 + D_1V_1 \end{aligned}$$
   (b) 3D:
   $$\begin{aligned} nlin_1 &= nlin_1 + D_2V_1 + D_3V_3 \\ nlin_2 &= nlin_2 + D_1V_1 + D_3V_2, \\ nlin_3 &= nlin_3 + D_1V_3 + D_2V_2. \end{aligned}$$

The resultant $nlin_s$ is the desired $\mathcal{F}[\partial_j(V_jV_s)]$.

8.4.2 IncVF::Compute_nlin(IncSF &T)

In this function we compute $\mathcal{F}[\partial(V_jV_i)]$ and $\mathcal{F}[\partial_j(V_jF)]$. The procedure to compute $\mathcal{F}[\partial_j(V_jV_i)]$ is the same as above. To compute $\mathcal{F}[\partial_j(V_jF)]$ we perform the following operations:

1. Compute $F(\mathbf{r})$ by inverse transform of $F(\mathbf{k})$.
2. Compute $T.nlin = V_1(\mathbf{r})F(\mathbf{r})$ and $T.SFtemp = V_2(\mathbf{r})F(\mathbf{r})$. In SCFT, V_1F has even parity, and V_2F has odd parity.
3. Perform Forward tranform of $T.nlin$ and $T.SFtemp$. We take into account parity in SCFT basis.
4. Take derivatives: $D_1(T.nlin)$ (even parity) and $D_2(T.SFtemp)$ (odd parity).
5. $T.nlin = T.nlin + T.SFtemp$.

   Now $T.nlin$ contains $\mathcal{F}[\partial_j(V_jF)]$ for 2D flows. For 3D flows we need to perform several more steps.
6. Compute $T.SFtemp = V_3(\mathbf{r})F(\mathbf{r})$, and take take derivate $D_3(T.SFtemp)$ (odd parity).
7. $T.nlin = T.nlin + T.SFtemp$.

Now $T.nlin$ has $\mathcal{F}[\partial_j(V_jF)]$ for 3D flows.

8.4.3 IncVF::Compute_nlin(IncSF &T, string Pr_switch)

If Prandtl number is zero, then we do not need to compute nonlinear terms $(\mathbf{u}\cdot\nabla)T$. Hence we just invoke IncVF::Compute_nlin(). When Prantdl number is nonzero, we use IncVF::Compute_nlin(T) to compute all the nonlinear terms.

8.4.4 IncVF::Compute_nlin(IncVF & W)

For MHD, we need to compute the nonlinear terms

$$V.\mathbf{nlin} = [(\mathbf{V}\cdot\nabla)\mathbf{V} - (\mathbf{B}\cdot\nabla)\mathbf{B}] = \left[(\mathbf{Z}^-\cdot\nabla)\mathbf{Z}^+ + (\mathbf{Z}^+\cdot\nabla)\mathbf{Z}^-\right]/2.$$

In `IncVF::Compute_nlin(W)` we compute $\mathbf{nlin} = (\mathbf{Z}^-\cdot\nabla)\mathbf{Z}^+$ and $\mathbf{W}.\mathbf{nlin} = (\mathbf{Z}^+\cdot\nabla)\mathbf{Z}^-$. Using these we can compute the nonlinear term for the NS equation

$$V.\mathbf{nlin} = (\mathbf{nlin} + \mathbf{W}.\mathbf{nlin})/2. \tag{8.1}$$

We can also derive the dynamical equation for the magnetic field that yields

$$B.\mathbf{nlin} = [(\mathbf{V}\cdot\nabla)\mathbf{B} - (\mathbf{B}\cdot\nabla)\mathbf{V}] = \left[(\mathbf{Z}^-\cdot\nabla)\mathbf{Z}^+ - (\mathbf{Z}^+\cdot\nabla)\mathbf{Z}^-\right]/2.$$

The strategy is to compute nonlinear terms $T[\partial_j(Z_j^- Z_i^+)]$ and $T[\partial_j(Z_j^+ Z_i^-)]$ first, and then compute $V.\mathbf{nlin}$ and $B.\mathbf{nlin}$. We compute these nonlinear terms in the following steps:

1. The input to the functions are velocity ($\mathbf{V}$) and magnetic fields ($\mathbf{B}$) in MHD.
2. We compute Elsasser variable $\mathbf{V} \leftarrow \mathbf{Z}^+ = \mathbf{V} + \mathbf{B}$ and $\mathbf{W} \leftarrow \mathbf{Z}^- = \mathbf{V} - \mathbf{B}$. From this point to the step 9, the variables $\mathbf{V}$ and $\mathbf{W}$ are Elsasser variables.
3. Compute $\mathbf{V}(\mathbf{r})$, $\mathbf{W}(\mathbf{r})$ using Inverse_transform.
4. We compute the diagonal products V_sW_s and put them in $nlin_s$, i.e., $nlin_s \leftarrow V_sW_s$. The diagonal products have even parity. (`IncVF::Compute_RSprod_diag(W)`).
5. Perform forward transform of $nlin_s$ and take derivative, i.e., $nlin_s \leftarrow D_s\mathcal{F}[V_sW_s]$. For SCFT D_1 has odd parity. The function is `(NLIN::NLIN_diag_Forward_transform_d`
6. Since the diagonal terms are the same in both $nlin_s$ and $W.nlin_s$, we take $W.nlin_s = nlin_s$.
7. Compute off-diagonal products V_jV_i (`IncVF::Compute_RSprod_offdiag()`)
   (a) 2D: $V_1 = V_1W_2$; $W_1 = W_1V_2$. (SCFT: both odd parity).
   (b) 3D: $V_1 = V_1W_2$, $V_2 = V_2W_3$, $V_3 = V_3W_1$;
   $W_1 = W_1V_2$, $W_2 = W_2V_3$, $W_3 = W_3V_1$. (SCFT: The product V_2 and W_2 have even parity, and all others have odd parity.
8. `void RV_Forward_transform_RSprod()`: Foward transform the product V_i and W_is keeping parity in mind.

9. `IncVF::Derivative_RSprod()`:

   (a) 2D:

$$\begin{aligned} nlin_1 &= nlin_1 + D_2 V_1 \\ nlin_2 &= nlin_2 + D_1 W_1 \\ W.nlin_1 &= W.nlin_1 + D_2 W_1 \\ W.nlin_2 &= W.nlin_2 + D_1 V_1. \end{aligned}$$

   (b) 3D:

$$\begin{aligned} nlin_1 &= nlin_1 + D_2 V_1 + D_3 W_3 \\ nlin_2 &= nlin_2 + D_1 W_1 + D_3 V_2, \\ nlin_3 &= nlin_3 + D_1 V_3 + D_2 W_2, \\ W.nlin_1 &= W.nlin_1 + D_2 W_1 + D_3 V_3 \\ W.nlin_2 &= W.nlin_2 + D_1 V_1 + D_3 W_2, \\ W.nlin_3 &= W.nlin_3 + D_1 W_3 + D_2 V_2. \end{aligned}$$

10. The resultant $nlin_s = T[\partial_j(Z_j^- Z_s^+)]$ and $W.nlin_s = T[\partial_j(Z_j^+ Z_s^-)]$. From this variables we compute the nlin for the velocity and magnetic fields as $V.\mathbf{nlin} = (\mathbf{nlin}+\mathbf{W}.\mathbf{nlin})/2$ and $W.\mathbf{nlin} = (\mathbf{nlin}-\mathbf{W}.\mathbf{nlin})/2$ as described above.

11. We revert back to the velocity and magnetic fields from Elsasser variables: $\mathbf{V} = (\mathbf{Z}^+ + \mathbf{Z}^-)/2$ and $\mathbf{W}(\mathbf{B}) = (\mathbf{Z}^+ - \mathbf{Z}^-)/2$.

8.4.5 IncVF::Compute_nlin(IncVF& W, IncSF& T)

For magnetoconvection we compute nonlinear terms $V.\mathbf{nlin}$, $B.\mathbf{nlin}$, and $T.nlin = \mathcal{F}[\partial_j(V_j T)]$. For this

1. We compute $V.\mathbf{nlin}$ and $B.\mathbf{nlin}$ using IncVF::Compute_nlin(IncVF& W).

2. We compute $T.nlin = \mathcal{F}[\partial_j(V_j T)]$ using the procedure outlined in IncVF::Compute_nlin(IncSF& T).

8.4.6 IncVF::Compute_nlin(IncVF& W, CVF& nlinWdU, CVF& nlinUdW)

In this function we compute all the nonlinear terms $\mathcal{F}[\partial_j(W_j V_i)]$, $\mathcal{F}[\partial_j(V_j W_i)]$, $\mathcal{F}[\partial_j(V_j V_i)]$, and $\mathcal{F}[\partial_j(W_j W_i)]$. We compute them using the following steps.

1. Compute $\mathbf{V}(\mathbf{r})$ and $\mathbf{W}(\mathbf{r})$ by taking Inverse transform of $\mathbf{V}(\mathbf{k})$ and $\mathbf{W}(\mathbf{k})$.

2. Save $\mathbf{nlinWdU} = \mathbf{V}$ and $\mathbf{nlinUdW} = \mathbf{W}$.

3. Compute $nlin_s = T[\partial_j(W_j V_s)]$ and $W.nlin_s = T[\partial_j(V_j W_s)]$ using the procedure given in IncVF::`Compute_nlin(W)`. We cannot use IncVF::`Compute_nlin(W)` directly because it yields combinatin of $T[\partial_j(W_j V_s)]$ and $T[\partial_j(W_j V_s)]$ (see above).

4. Restore $\mathbf{V} = \mathbf{nlinWdU}$ and $\mathbf{W} = \mathbf{nlinWdW}$.

5. $\mathbf{nlinWdU} = \mathbf{nlin}$ and $\mathbf{nlinUdW} = \mathbf{W}.\mathbf{nlin}$.

6. Compute $nlin_s = T[\partial_j(V_j V_s)]$ using `IncVF::Compute_nlin()`;

7. Compute $nlin_s = T[\partial_j(W_j W_s)]$ using `W.Compute_nlin()`. The result is stored in `W.nlin`i .

8.5 Computation of Pressure

The operation is done by `void IncVF::Compute_pressure()` in two steps. Before entering the function, $nlin_i = \mathcal{F}[\partial_j(V_j V_i)]$, where $\mathbf{V}$ is the velocity field. The steps are

1. We compute the divergence of $F(\mathbf{k}) = D_i\mathcal{F}[\partial_j(V_j V_i)]$ using `IncVF::Compute_divergence_nlin()`.

2. We compute $F(\mathbf{k}) = F(\mathbf{k})/k^2$ using `NLIN::CS_divide_ksqr()`. The result is the pressure field. Note that

$$-\nabla^2 p = \nabla \cdot [(\mathbf{u}\cdot\nabla)\mathbf{u}].$$

8.6 Simple useful functions

Here we list functions in `IncCF` and IncSF that were not convered above.

8.6.1 Elsasser variables

- `void UB_to_Elsasser(IncVF& W)`: Before entering this function, $\mathbf{V}$ and $\mathbf{W}$ are the velocity and magnetic fields respectively. This function yields $\mathbf{V} \leftarrow \mathbf{Z}^+ = (\mathbf{V} + \mathbf{W})$ and $\mathbf{W}(\mathbf{B}) \leftarrow \mathbf{Z}^- = (\mathbf{V} - \mathbf{W})$. The fields $\mathbf{Z}^\pm$ are called Elasser variables.

- `void Elsasser_to_UB(IncVF& W)`: Before entering this function, $\mathbf{V}$ and $\mathbf{W}$ are the $\mathbf{Z}^+$ and $\mathbf{Z}^-$ respectively. This function yields $\mathbf{V} = (\mathbf{Z}^+ + \mathbf{Z}^-)/2$ and $\mathbf{W}(\mathbf{B}) = (\mathbf{Z}^+ - \mathbf{Z}^-)/2$. Now $\mathbf{V}, \mathbf{W}$ represent velocity and magnetic field respectively.

- `void UB_to_Elsasser_nlin(IncVF& W)`: The dynamical equations for the Elasser variables are

$$\frac{\partial \mathbf{Z}^\pm}{\partial t} = -(\mathbf{Z}^\mp\cdot\nabla)\mathbf{Z}^\pm - \nabla p + \nu_+\nabla^2\mathbf{Z}^\pm + \nu_-\nabla^2\mathbf{Z}^\mp,$$

where $\nu_\pm = (\nu \pm W.\nu)/2$. Adding the equations for $\mathbf{Z}^\pm$ and by dividing by 2, we obtain

$$\frac{\partial \mathbf{V}}{\partial t} = -\left[(\mathbf{Z}^- \cdot \nabla)\mathbf{Z}^+ + (\mathbf{Z}^+ \cdot \nabla)\mathbf{Z}^-\right]/2 - \nabla p + \nu\nabla^2\mathbf{V}.$$

When we compare with the NS equation we obtain

$$V.\mathbf{nlin} = [(\mathbf{V} \cdot \nabla)\mathbf{V} - (\mathbf{W} \cdot \nabla)\mathbf{W}] = \left[(\mathbf{Z}^- \cdot \nabla)\mathbf{Z}^+ + (\mathbf{Z}^+ \cdot \nabla)\mathbf{Z}^-\right]/2.$$

In the following function IncVF::Compute_nlin(W) we compute $\mathbf{nlin} = (\mathbf{Z}^- \cdot \nabla)\mathbf{Z}^+$ and $\mathbf{W}.\mathbf{nlin} = (\mathbf{Z}^+ \cdot \nabla)\mathbf{Z}^-$ using which we can compute the nonlinear term for the NS equation

$$V.\mathbf{nlin} = (\mathbf{nlin} + \mathbf{W}.\mathbf{nlin})/2. \tag{8.2}$$

We can also derive the dynamical equation for the magnetic field by the operation $\partial(\mathbf{Z}^+ - \mathbf{Z}^-)/\partial t$:

$$\frac{\partial \mathbf{B}}{\partial t} = -\left[(\mathbf{Z}^- \cdot \nabla)\mathbf{Z}^+ - (\mathbf{Z}^+ \cdot \nabla)\mathbf{Z}^-\right]/2 + B.\nu\nabla^2\mathbf{B},$$

where $\mathbf{B}$ is the magnetic field. Therefore

$$B.\mathbf{nlin} = \left[(\mathbf{V} \cdot \nabla)\mathbf{B} - (\mathbf{B} \cdot \nabla)\mathbf{V}^-\right] = \left[(\mathbf{Z}^- \cdot \nabla)\mathbf{Z}^+ - (\mathbf{Z}^+ \cdot \nabla)\mathbf{Z}^-\right]/2.$$

Hence

$$B.\mathbf{nlin} = (\mathbf{nlin} - \mathbf{W}.\mathbf{nlin})/2. \tag{8.3}$$

The computations of $V.\mathbf{nlin}$ and $B.\mathbf{nlin}$ using Eqs. (8.2,n8.3) are done in the function UB_to_Elsasser_nlin(IncVF& W).

- `void Elsasser_to_UB_nlin(IncVF& W)`: This function is opposite of the above. We can compute nonlinear terms $(\mathbf{Z}^\mp \cdot \nabla)\mathbf{Z}^\pm$ using $V.\mathbf{nlin}$ and $B.\mathbf{nlin}$.

- `void IncVF::UB_to_Elsasser_force(IncVF& W)`: Computes forces for $\mathbf{Z}^\pm$ given the forces for $\mathbf{U}$ and $\mathbf{B}$.

- `void IncVF::Elsasser_to_UB_force(IncVF& W)`: Computes forces for $\mathbf{U}$ and $\mathbf{B}$ given the force for $\mathbf{Z}^\pm$.

8.6.2 Compute divergence

- `void IncVF::Compute_divergence_nlin()`: This function computes the divergence of the field $\mathbf{nlin}(\mathbf{k})$ and puts the result in the CSF $F(\mathbf{k})$, $F(\mathbf{k}) = T[D_j(nlin_j)]$. Pressure is computed after the divergence computation (see Sec. 8.5).

- `void IncVF::Compute_divergence_field`: Compute divergence of the field $\mathbf{V}(\mathbf{k})$ and puts the result in the CSF $F(\mathbf{k})$, i.e., $F(\mathbf{k}) = T[D_j(V_j)]$. Since $F(\mathbf{k})$ is supposed to contain pressure, divergence of a field should be computed when $F(\mathbf{k})$ is free.

8.6.3 Multiply $\exp(-\nu K^2 dt)$ and $\exp(-\kappa K^2 dt)$

- `void IncVF::Mult_field_exp_ksqr_dt(DP dt)`: $\mathbf{V}(\mathbf{k},t) = \mathbf{V}(\mathbf{k},t)\exp(-\nu K^2 dt)$, where ν is the `dissipation_coefficient` of the `IncVF`.
- `void IncVF::Mult_nlin_exp_ksqr_dt(DP dt)`: $\mathbf{nlin}(\mathbf{k},t) = \mathbf{nlin}(\mathbf{k},t)\exp(-\nu K^2 dt)$.
- `void IncSF::Mult_field_exp_ksqr_dt(DP dt)`: $F(\mathbf{k},t) = F(\mathbf{k},t)\exp(-\kappa K^2 dt)$, where κ is the `difussuion_coefficient` of the `IncSF`.
- `void IncSF::Mult_nlin_exp_ksqr_dt(DP dt)`: $nlin(\mathbf{k},t) = nlin(\mathbf{k},t)\exp(-\kappa K^2 dt)$.

8.6.4 Add nlin$\times dt$ to the field

- `void IncVF::Add_nlin_dt(DP dt)`: $\mathbf{V} = \mathbf{V} + \mathbf{nlin} \times dt$.
- `void IncSF::Add_nlin_dt(DP dt)`: $F = F + nlin \times dt$.

8.6.5 Add nlin to field

- void IncVF::Add_nlin_to_field(CVF& W, DP factor): $W.\mathbf{V} = W.\mathbf{V} +$ factor $\times$ **nlin**.
- void IncSF::Add_nlin_to_field(CSF& T, DP factor): $T.F = T.F +$factor$\times$ $nlin$.

8.6.6 Copy fields

- `void IncVF::Copy_field_to`(CVF& W): $W.\mathbf{V} = \mathbf{V}$.
- `void IncVF::Copy_field_from`(CVF& W): $\mathbf{V} = W.\mathbf{V}$.
- `void IncSF::Copy_field_to`(CSF& T): $T.F = F$.
- `void IncSF::Copy_field_from`(CVF& T): $F = T.F$

8.6.7 Compute cross helicity

- `IncVF::Get_cross_helicity(IncVF& W)`: returns $H_c = \frac{1}{2}\mathbf{V} \cdot \mathbf{W}$.

8.6.8 Compute Nusselt Number

- `IncVF::Get_Nusselt_no(IncSF& T)`: returns $Nu = 1 + \Re(V_1(\mathbf{k})F^*(\mathbf{k}))$.

8.6.9 Fourier space point functions

These functions are useful for manipulating a vector at a given point in Fourier space.

- `void Last_component(int kx, int ky, complx &Vx, complx &Vy)`: In 2D, if $k_y \neq 0$, $V_y(\mathbf{k}) = -D_x V_x(\mathbf{k})/ik_y$. If $k_y = 0$, $V_x = 0$ and $V_y = V_x$.

- `void Last_component(int kx, int ky, int kz, complx &Vx, complx &Vy, complx &Vz)`: In 3D, if $k_z \neq 0$, $V_z(\mathbf{k}) = -(D_x V_x + D_y V_y)/ik_z$. If $k_z = 0$, then the read components are taken to be V_x and V_z. We compute V_y using `void Last_component(kx, ky, Vx, Vy)`.

- `void Compute_VyVz(int kx, int ky, int kz, complx Vx, complx Omega, complx &Vy, complx &Vz)`: The incompressibilty equation and vorticity (along x) yield

$$D_y V_y + D_z V_z = -D_x V_x \tag{8.4}$$
$$-D_z V_y + D_y V_z = \Omega_x. \tag{8.5}$$

  The solution of the above equations yield

$$V_y = \frac{D_y D_x V_x + D_z \omega_x}{K_\perp^2},$$
$$V_z = \frac{D_z D_x V_x - D_y \omega_x}{K_\perp^2},$$

  where $K_\perp^2 = (k_y \text{kfactor}(2) + k_z \text{kfactor}(3))^2$. Note that $D_s = ik_s \times \text{kfactor}(s)$ for $s \geq 2$, and $D_{_1} = k_1 \text{kfactor}(1)$ in SCFT basis and $D_1 = ik_1 \text{kfactor}(1)$ in FOUR basis.

  If $k_y = 0$ and $k_z = 0$ (consequently $K_\perp = 0$), then the solution V_y and V_z are indeterminate because the determinant of the matrix formed from the linear equations (8.4, 8.5) is zero. For this case we set $\mathbf{V}(\mathbf{k}) = 0$.

- `void IncVF::Add_complex_conj(int kx, int ky, complx Vx, complx Vy)`: If $\mathbf{V}(k_x, k_y) = (V_x, V_y)$ is added to the vector field, then this function adds complex conjugate in the vector field if required. The function in IncSF `void IncSF::Add_complex_conj(int kx, int ky, complx G)` has similar function.

  - FOUR: $-N_1/2 < k_x \leq N_1/2$ and $k_y \leq N_2/2$. If $k_y > 0$, program assumes that $\mathbf{V}(-k_x, -k_y) = conj(\mathbf{V}(k_x, k_y))$, hence reality condition is satisfied. When $k_y = 0$, then we need to assign $\mathbf{V}(-k_x, 0) = conj(\mathbf{V}(k_x, 0))$.
  - SCFT: Since $k_x \geq 0$, no need for any mode to be added to satisfy reality condition.

- `void IncVF::Add_complex_conj(int kx, int ky, int kz, complx Vx, complx Vy, complx Vz)`: If $\mathbf{V}(k_x, k_y, k_z) = (V_x, V_y, V_z)$ is added to the vector field, then this function adds complex conjugate in the vector field if required. The function in IncSF `void IncSF::Add_complex_conj(int kx, int ky, int kz, complx G)` has similar function.
  - FOUR: When $k_z = 0$, we need to assign $\mathbf{V}(-k_x, -k_y, 0) = conj(\mathbf{V}(k_x, k_y, 0))$ to satisfy the reality condition.
  - SCFT: When $k_z = 0$, we need to assign $\mathbf{V}(k_x, -k_y, 0) = conj(\mathbf{V}(k_x, k_y, 0))$ to satisfy the reality condition.
- `void IncVF::Assign_field_comp_conj(int kx, int ky, complx Vx, complx Vy)`: Assigns $(*\mathbf{V})(k_x, k_y) = (V_x, V_y)$. If $k_y = 0$ and basis_type is FOUR, then $(*\mathbf{V})(-k_x, 0) = (V_x, V_y)^*$.
- `void IncVF::Assign_field_comp_conj(int kx, int ky, int kz, complx Vx, complx Vy, complx Vz)`: Assigns $(*\mathbf{V})(k_x, k_y, k_z) = (V_x, V_y, V_z)$. If $k_z = 0$, then add complex conjugate according to the function `Add_complex_conj(..)`.
- `void IncSF::Assign_field_comp_conj(int kx, int ky, complx G)`: Assigns $(*F)(k_x, k_y) = G$. If $k_y = 0$ and basis_type is FOUR, then $(*F)(-k_x, 0) = G^*$.
- `void IncSF::Assign_field_comp_conj(int kx, int ky, int kz, complx G)`: Assigns $(*F)(k_x, k_y, k_z) = G$. If $k_z = 0$, then add complex conjugate according to the function `Add_complex_conj(...)`.
- `DP IncVF::Get_Tk(int kx, int ky)`: For 2D flows returns $-\Re[\mathbf{nlin}(k_x, k_y) \cdot (\mathbf{V}(k_x, k_y))^*]$. Note that $T(k_x, k_y)$ is the energy input to the Fourier mode $\mathbf{V}(k_x, k_y)$ due to the nonlinear interactions. Similar definition works for 3D.
- `DP IncSF::Get_Tk(int kx, int ky)`: For 2D flows returns $-\Re[nlin(k_x, k_y) \times (F(k_x, k_y))^*]$. Note that $T(k_x, k_y)$ is the energy input to the Fourier mode $F(k_x, k_y)$ due to the nonlinear interactions. Similar definition works for 3D.

Here we end our discussion on the library IncVF.

Chapter 9

Computation of energy transfers in turbulence

In turbulent flow energy is transferred from one region of wavenumber space to another. The study of energy transfer is one of they area of research in turbulence. In the following discussion we will discuss some of the functions that compute the energy transfers in turbulence.

9.1 Introduction

9.1.1 Energy transfers in turbulence

Let us consider fluid turbulence first. The energy transfer from a region A_1 to another region A_2 is

$$\begin{aligned} T^{u,A_1}_{u,A_2} &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \int_{\mathbf{p}\in A_1} d\mathbf{p} \Re[(-i\mathbf{k}\cdot\mathbf{u}(\mathbf{k}-\mathbf{p}))(\mathbf{u}(\mathbf{p})\cdot\mathbf{u}^*(\mathbf{k})] \\ &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \Re\left[-u_i^*(\mathbf{k}) \int_{\mathbf{p}\in A_1} d\mathbf{p}[ik_j u_j(\mathbf{k}-\mathbf{p})u_i(\mathbf{p})]\right] \\ &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \Re\left[-u_i^*(\mathbf{k}) \times nlin_i^{uu}(\mathbf{k})\right]. \end{aligned}$$

We compute $Nlin_i(\mathbf{u},\mathbf{u})$ using the same procedure as outlined in IncFlow chapter. The only difference is that $\mathbf{u}(\mathbf{p})$ is nonzero only in the region A_1.

For scalar θ, the energy transfer among the scalar mode is

$$\begin{aligned} T^{\theta,A_1}_{\theta,A_2} &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \int_{\mathbf{p}\in A_1} d\mathbf{p} \Re[(-i\mathbf{k}\cdot\mathbf{u}(\mathbf{k}-\mathbf{p}))(\theta(\mathbf{p})\theta^*(\mathbf{k})] \\ &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \Re\left[-\theta^*(\mathbf{k}) \int_{\mathbf{p}\in A_1} d\mathbf{p}[ik_j u_j(\mathbf{k}-\mathbf{p})\theta(\mathbf{p})]\right] \\ &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \Re\left[-\theta^*(\mathbf{k}) \times nlin^{\theta\theta}(\mathbf{k})\right]. \end{aligned}$$

For MHD, the velocity-to-velocity transfer is the same as T^{u,A_1}_{u,A_2} . The other transfers are

$$\begin{aligned} T^{b,A_1}_{b,A_2} &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \int_{\mathbf{p}\in A_1} d\mathbf{p} \Re[(-i\mathbf{k}\cdot\mathbf{u}(\mathbf{k}-\mathbf{p}))(\mathbf{b}(\mathbf{p})\cdot\mathbf{b}^*(\mathbf{k})] \\ &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \Re\left[-b_i^*(\mathbf{k}) \int_{\mathbf{p}\in A_1} d\mathbf{p}[ik_j u_j(\mathbf{k}-\mathbf{p}) b_i(\mathbf{p})]\right] \\ &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \Re\left[-b_i^*(\mathbf{k}) \times nlin_i^{ub}(\mathbf{k})\right]. \end{aligned}$$

$$\begin{aligned} T^{u,A_1}_{b,A_2} &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \int_{\mathbf{p}\in A_1} d\mathbf{p} \Re[(i\mathbf{k}\cdot\mathbf{b}(\mathbf{k}-\mathbf{p}))(\mathbf{u}(\mathbf{p})\cdot\mathbf{b}^*(\mathbf{k})] \\ &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \Re\left[b_i^*(\mathbf{k}) \int_{\mathbf{p}\in A_1} d\mathbf{p}[ik_j b_j(\mathbf{k}-\mathbf{p}) u_i(\mathbf{p})]\right] \\ &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \Re\left[b_i^*(\mathbf{k}) \times nlin_i^{bu}(\mathbf{k})\right]. \end{aligned}$$

$$\begin{aligned} T^{b,A_1}_{u,A_2} &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \int_{\mathbf{p}\in A_1} d\mathbf{p} \Re[(i\mathbf{k}\cdot\mathbf{b}(\mathbf{k}-\mathbf{p}))(\mathbf{b}(\mathbf{p})\cdot\mathbf{u}^*(\mathbf{k})] \\ &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \Re\left[u_i^*(\mathbf{k}) \int_{\mathbf{p}\in A_1} d\mathbf{p}[ik_j b_j(\mathbf{k}-\mathbf{p}) b_i(\mathbf{p})]\right] \\ &= \int_{\mathbf{k}\in A_2} d\mathbf{k} \Re\left[u_i^*(\mathbf{k}) \times nlin_i^{bb}(\mathbf{k})\right]. \end{aligned}$$

The nonlinear terms of T's arise from the following nonlinear terms of the equations

$$\begin{aligned} nlin_i^{uu}(\mathbf{k}) &\rightarrow \partial_j(u_j u_i), \\ nlin_i^{bb}(\mathbf{k}) &\rightarrow \partial_j(b_j b_i), \\ nlin_i^{ub}(\mathbf{k}) &\rightarrow \partial_j(u_j b_i), \\ nlin_i^{bu}(\mathbf{k}) &\rightarrow \partial_j(b_j u_i). \end{aligned}$$

The first superscript of $nlin_i$ is the helper and the second superscript is the giver. Note that $nlin_i^{uu}(\mathbf{k})$ and $nlin_i^{bb}(\mathbf{k})$ are the nonlinear terms of the Navier-Stokes equation ($\dot{\mathbf{u}}$), and $nlin_i^{ub}(\mathbf{k})$ and $nlin_i^{bu}(\mathbf{k})$ are the nonlinear terms of the induction equation ($\dot{\mathbf{b}}$).

9.1.2 Energy fluxes

For fluid turbulence we define energy leaving a wavenumber sphere of radius K_0 as the energy flux. This energy transfer takes place from the modes inside the above wavenumber sphere to the modes outside the sphere. We denote this quantity by $\Pi^{u<}_{u>}(K_0)$ that is computed by

$$\Pi^{u<}_{u>}(K_0) = \sum_{k \geq K_0} -\Re\left[u_i^*(\mathbf{k}) \times nlin_i^{u,u<}(\mathbf{k})\right],$$

where the $\mathbf{u}^<$ is the giver field that naturally resides inside the sphere.

We can define energy flux $\Pi^{\zeta<}_{\zeta>}(K_0) =$ for scalar field ζ in the similar manner as the energy of the scalar field leaving the wavenumber sphere of radius of K_0. This

COMPLETE..

9.1.3 Shell-to-shell energy transfers

The shell-to-shell energy transfer from the shell m to the shell n is defined as

$$T^{uu}_{nm} = \sum_{k \in n} -\Re\left[u_i^*(\mathbf{k}) \times nlin_i^{u,u(m)}(\mathbf{k})\right],$$

where $\mathbf{u}(m)$ is the giver field that contains that is nonzero only for $k \in m$.

9.2 Wavenumber spheres and shells for energy transfer studies

In this chaper we will discuss the energy flux from a wavenumber sphere, and the shell-to-shell energy transfers between two wavenumber shells. In the following discussion we will describe how we setup the wavenumber spheres and shells in our simulation.

We consider `Nspheres` spheres of increasing radius. The last sphere contains all the modes, and the last but one sphere just fits inside the wavenumber box. We define *sphere*(n) as a sphere containing waveumbers $0 \leq k' <$ sphereradius(n). Unless specified, we take sphererеadius(0) = 0, sphererеadius(1) = 2, sphererеadius(2) = 4, sphererеadius(3) = 8, sphererеadius(4) = $8s, ...$, sphererеadius(Nspheres-2) = Maxpossible_inner_radius/2, sphererеadius(Nspheres-1) = Maxpossible_inner_radius, sphererеadius(Nspheres) = ∞.

In simulation we take a const variable `INF_RADIUS` whose value is 10000.0 is takne to be ∞. The *Maxpossible_ inner_ radius* is the radius of the largest wavenumber sphere that fits inside the box. The radii of the spheres are distributed logarithmically for $4 <$ K < Maxpossible_inner_radius/2 . Hence the radius of the 4^{th} sphere to the $(\text{Nsphere} - 3)^{th}$ sphere is given by

$$\text{sphereradius}(n) = 8 \times 2^{s(n-3)}$$

where

$$8 \times 2^{s(\text{Nsphers}-5)} = \text{Maxpossible\_inner\_radius}/2.$$

The minimum number Nspheres required for this scheme is 5, and Maxpossible_inner_radius $\geq$ 16. For Maxpossible_inner_radius = 16, the radii are 2,4,8,16, ∞ respectively, and the parameter s is 0. If Nspheres = 6, then the radii are 2,4,8,

Maxpossible_inner_radius/2, Maxpossible_inner_radius, ∞. A nontrivial s appears only for $Nspheres > 6$. [Exercise: Workout radii for $Nsphere = 7$ and $Maxpossible\_inner\_radius = 32$.

The shells are prescribed in a similar manner. We define the shell(n) as the shell that contains wavenumbers shellradius($n-1$) $\leq K' <$ shellradius(n). Clearly the wavenumber range of the shells $1, 2, 3, 4, ..., Nshells - 1, Nshells$ are $[0, 2), [2, 4), [4, 8), [8, 8\times 2^s), ...,$ [Maxpossible_inner_radius/2, Maxpossible_inner_radius), [Maxpossible_
respectively.

The wavenumber spheres, shells, energy fluxes, and shell-to-shell transfers are defined in the class `EnergyTr` that will be described below.

9.3 Variables of class EnergyTr

The class `EnergyTr` has the following public variables:

- `*Vfromi`: Contains the giver vector field. The size of the array assigned for `Vfromi` is the same as `Vi`.
- `*tempET`: Temporary array that has the same size as `Vi`.
- `D, *NET, ET_basis_type, *ET_kfactor`: Space dimension, array size of `Vi`, basis type, and kfactor[] respectively.
- `no_spheres`: Number of wavenumber spheres for energy flux calculations.
- `no_shells`: Number of wavenumber shells used in shell-to-shell transfers. In our simulation we take `no_spheres = no_shells`.
- `*sphereradius`: An array containing the radii of the wavenumber spheres. See details in the constructor function.

The energy fluxes are stored in the following arrays. We consider a wavenumber sphere with index `sphere_index`.

- `*iso_flux_self(sphere_index)`: The energy flux from the modes of field $\mathbf{V}$ *strictly* inside the sphere to the modes of the same field outside the sphere. The surface of the sphere is not included in the *"inside sphere"*, but included in the "*outside sphere*".
- `*iso_flux_SF(sphere_index)`: The energy flux from the modes of scalar field ζ *strictly* inside the sphere to the modes of the same field (ζ) outside the sphere.
- `*iso_flux_VF_in_in(sphere_index)`: The energy flux from the modes of the field $\mathbf{V}$ *strictly* inside the sphere to the modes of field $\mathbf{W}$ *strictly* inside the sphere.
- `*iso_flux_VF_in_out(sphere_index)`: The energy flux from the modes of the field $\mathbf{V}$ *strictly* inside the sphere to the modes of field $\mathbf{W}$ outside the sphere.

- `*iso_flux_VF_out_out(sphere_index)`: The energy flux from the modes of the field $\mathbf{V}$ outside the sphere to the modes of field $\mathbf{W}$ outside the sphere.
- `*iso_flux_Elsasser(sphere_index)`: The energy flux from the modes of field $\mathbf{Z}^+$ *strictly* inside the sphere to the modes of $\mathbf{Z}^+$ outside the sphere.
- `*W.iso_flux_Elsasser(sphere_index)`: The energy flux from the modes of field $\mathbf{Z}^-$ *strictly* inside the sphere to the modes of $\mathbf{Z}^-$ outside the sphere.
- `*iso_flux_self_real(sphere_index)`: Contributions to the above flux from $\Re(\mathbf{nlin})$ and $\Re(\mathbf{V}_{to})$.
- `*iso_flux_self_imag(sphere_index)`: Contributions to the above flux from $\Im(\mathbf{nlin})$ and $\Im(\mathbf{V}_{to})$. Note that `*iso_flux_self = *iso_flux_self_real + *iso_flux_self_imag`.

The shell-to-shell energy transfers are stored in the following arrays. The giver shell is m and the receive shell is n.

- `*shell_to_shell_self(m, n)`: The shell-to-shell energy transfer from the modes of field $\mathbf{V}$ in the shell m to the modes of field $\mathbf{V}$ in the shell n. Note that modes on the inner surface of the shell are included in the shell, but not the modes on the outer shell.
- `*shell_to_shell_VF(m, n)`: The shell-to-shell energy transfer from the modes of field $\mathbf{V}$ in the shell m to the modes of field $\mathbf{W}$ in the shell n.
- `*shell_to_shell_SF(m, n)`: The shell-to-shell energy transfer from the modes of scalar field ζ in the shell m to the modes of field ζ in the shell n.
- `*shell_to_shell_self(m, n)`: The shell-to-shell energy transfer from the modes of field $\mathbf{V}$ in the shell m to the modes of field $\mathbf{V}$ in the shell n.
- `*shell_to_shell_Elsasser(m, n)`: The shell-to-shell energy transfer from the modes of field $\mathbf{Z}^+$ in the shell m to the modes of field $\mathbf{Z}^+$ in the shell n.
- `*W.shell_to_shell_Elsasser(m, n)`: The shell-to-shell energy transfer from the modes of field $\mathbf{Z}^-$ in the shell m to the modes of field $\mathbf{Z}^-$ in the shell n.
- `*shell_to_shell_self_real(m, n)`: Contribution from the interactions of $\Re(\mathbf{V})$ with the $\Re(\mathbf{nlin})$. Similar interpretations for the other shell-to-shell transfers.
- `*shell_to_shell_self_imag(m, n)`: Contribution from the interactions of $\Re\Im(\mathbf{V})$ with the $\Re\Im(\mathbf{nlin})$. Similar interpretations for the other shell-to-shell transfers.

Note that

- `*iso_flux_self(0)` = 0; All other fluxes are zero for sphere = 0. This is because the the sphere(0) has zero radius.

- `*iso_flux_self(nosphere) = *iso_flux_SF(nosphere) = 0,` and `*iso_flux_Elsasser(nosphere = 0`and because no energy flows out of the last sphere whose radius is infinite.

- `*iso_flux_VF_in_in(nosphere)` is the total energy transfer from field **V** to field **W**.

- `*iso_flux_VF_out_out(nosphere) = 0.`

- `*shell_to_shell_self(0,n) = *shell_to_shell_self(m,0) = 0` because the 0th shell contains no modes.

9.4 Functions of class EnergyTr

9.4.1 Constructor

`EnergyTr(int *NN, string prog_basis_type, DP *prog_kfactor, int shell_input_sheme, int nospheres, int noshells, Array<DP,1> Rshells):`

The constructor allocates array **Vfrom** and `tempET`. Also initializes the `*sphereradius`. If the variable `shell_input_scheme` in the main program is 0, then the radii are generated by the program as discussed above: $[0, 2, 8, 8s, ..,$ Maxpossible_inner_radius/2,Maxpossible_inner_radius, $\infty]$. If `shell_input_scheme` is 1, then the radii are prescribed in the input. The inputs are `*sphereradius(1),..,*sphereradius(nospl` `*sphereradius(0)=0`, and `*sphereradius(nospheres)`=∞. The radius of nth sphere is `*sphereradius`(n).

9.4.2 Fill sphere and shells

- `void IncVF::Fill_in_sphere(int m):` Fills the sphere m with **V** by setting $\mathbf{Vfrom}(\mathbf{k}) = \mathbf{V}(\mathbf{k})$ for $\mathbf{k} \in m$, and $\mathbf{Vfrom}(\mathbf{k}) = 0$ otherwise. Modes on the surface of the sphere are not included in **Vfrom**, to ascertain that the included modes are *strictly* inside the sphere.

- `void IncVF::Fill_in_sphere(int m, IncSF& T):` Fills the sphere m with $T.\zeta$ by setting $Vfrom_1(\mathbf{k}) = T.\zeta(\mathbf{k})$ for $\mathbf{k} \in m$, and $\mathbf{Vfrom}(\mathbf{k}) = 0$ otherwise.

- `void IncVF::Fill_in_sphere(int m, IncVF& W):` Fills the sphere m with $W.\mathbf{V}$ by setting $\mathbf{Vfrom}(\mathbf{k}) = W.\mathbf{V}(\mathbf{k})$ for $\mathbf{k} \in m$, and $\mathbf{Vfrom}(\mathbf{k}) = 0$ otherwise.

- `void IncVF::Fill_out_sphere(int m):` Fills the sphere m with **V** by setting $\mathbf{Vfrom}(\mathbf{k}) = \mathbf{V}(\mathbf{k})$ for $\mathbf{k} \notin m$, and $\mathbf{Vfrom}(\mathbf{k}) = 0$ for $\mathbf{k} \in m$,. Modes on the surface of the sphere are included in **Vfrom**. Similar operations are done for functions with `IncSF&` and `IncVF&`.

- `void IncVF::Fill_shell(int n)`: Fills the shell n with $\mathbf{V}$ by setting $\mathbf{Vfrom}(\mathbf{k}) = \mathbf{V}(\mathbf{k})$ for $\mathbf{k} \in n$, and $\mathbf{Vfrom}(\mathbf{k}) = 0$ otherwise. Modes on the inner-surface of the shell are not included in **Vfrom**, but not the modes on the outer-surface of the shell. Similar operations for functions with `IncSF&` and `IncVF&`.

9.4.3 Product of field with nlin

- `DP IncVF::Prod_out_sphere_nlinV(int m)`: Returns $\sum_{k \notin m} \Re[\mathbf{V}(\mathbf{k}) \cdot (\mathbf{nlin}(\mathbf{k}))^*]$, i.e., the sum is done over all the modes outside the sphere m. The modes on the surface of the sphere is included in the sum.

- `DP IncVF::Prod_out_sphere_nlinV(int m, IncSF& T)`: Returns $\sum_{k \notin m} \Re[T.\zeta(\mathbf{k}) \times (nlin_1(\mathbf{k}))^*]$, where $nlin_1(\mathbf{k})$ contains $\mathcal{F}[\mathbf{u} \cdot \nabla\zeta]$.

- `DP IncVF::Prod_out_sphere_nlinV(int m, IncVF& W)`: Returns $\sum_{k \notin m} \Re[W.\mathbf{V}(\mathbf{k}) \cdot (\mathbf{nlin}(\mathbf{k}))^*]$.

- `DP IncVF::Prod_in_sphere_nlinV(int m)`: Returns $\sum_{k \in m} \Re[\mathbf{V}(\mathbf{k}) \cdot (\mathbf{nlin}(\mathbf{k}))^*]$, i.e., the sum is done over all the modes inside the sphere m. The modes on the surface of the sphere is mot included in the sum. Similar operations are done for functions with `IncST` and `IncVF` arguments.

- `DP IncVF::Prod_shell_nlinV(int n)`: Returns $\sum_{k \in n} \Re[\mathbf{V}(\mathbf{k}) \cdot (\mathbf{nlin}(\mathbf{k}))^*]$, i.e., the sum is done over all the modes inside the shell n. The modes on the inner-surface of the shell is included in the sum, but not the modes on the outer surface of shell. Similar operations are done for functions with `IncST` and `IncVF` arguments.

- `void IncVF::Prod_out_sphere_nlinV_real_imag(int sphere_index, DP& tot_real, DP& tot_imag)`: Computes `tot_real` = $\sum_{k \in n} \Re[\mathbf{V}(\mathbf{k})] \cdot \Re[\mathbf{nlin}(\mathbf{k})]]$ and `tot_imag` = $\sum_{k \in n} \Im[\mathbf{V}(\mathbf{k})] \cdot \Im[\mathbf{nlin}(\mathbf{k})]]$. Note that `Prod_out_sphere_nlinV = tot_real + tot_imag`. Similar operations for other real_imag functions.

- `DP IncVP::Prod_shell_forceV(int n)`: Returns $\sum_{k \in n} \Re[\mathbf{V}(\mathbf{k}) \cdot (\mathbf{F}^{\mathbf{V}}(\mathbf{k}))^*]$.

- `DP IncVP::Prod_shell_forceV(int n, IncSF& W)`: Returns $\sum_{k \in n} \Re[T.\zeta(k) \times (F^{\zeta}(\mathbf{k}))^*]$.

- `DP IncVP::Prod_shell_forceV(int n, IncVF& W)`: Returns $\sum_{k \in n} \Re[W.\mathbf{V}(\mathbf{k}) \cdot (\mathbf{F}^{\mathbf{W}}(\mathbf{k}))^*]$.

9.4.4 Computation of real space products

Computes products $H_j V_{fi}$ where H_i could be either V_i or W_i.

- `void IncVF::Compute_RSprod_diagET()`: Computes $nlin_i(\mathbf{r}) = V_i(\mathbf{r}) V_{fi}(\mathbf{r})$. In the present and the next function $\mathbf{V}_f$ is the giver, and $\mathbf{V}$ is the helper.

- `void IncVF::Compute_RSprod_offdiagET()`: Computes Computes the off-diagnoal product term; In 2D: $V_1 = V_1 V_{f2}$ and $V_{f1} = V_{f1} V_2$; In 3D: $V_1 = V_1 V_{f2}$, $V_{f1} = V_{f1} V_2$, $V_2 = V_2 V_{f3}$, $V_{f2} = V_{f2} V_3$, $V_3 = V_3 V_{f1}$, $V_{f3} = V_{f3} V_1$.

- `void IncVF::Compute_RSprod_diagET(IncVF& W)`: Computes $nlin_i(\mathbf{r}) = W_i(\mathbf{r}) V_{fi}(\mathbf{r})$. In the present and the next function $\mathbf{V}_f$ is the giver, and $\mathbf{W}$ is the helper.

- `void IncVF::Compute_RSprod_offdiagET(IncVF& W)`: Computes Computes the off-diagnoal product term; In 2D: $V_1 = W_1 V_{f2}$ and $V_{f1} = V_{f1} W_2$; In 3D: $V_1 = W_1 V_{f2}$, $V_{f1} = V_{f1} W_2$, $V_2 = W_2 V_{f3}$, $V_{f2} = V_{f2} W_3$, $V_3 = W_3 V_{f1}$, $V_{f3} = V_{f3} W_1$.

Computes transforms

- `void IncVF::ET_Forward_RSprod_Vfrom()`: We forward transform of $\mathbf{V}_f$ after the above product computations. Note that in SCFT basis in 2D, V_{f1} has odd parity; and in 3D, V_{f1} and V_{f3} have odd parity, but V_{f2} has even parity.

- `void RV_Forward_transform_RSprod()`: We forward transform vector $\mathbf{V}(\mathbf{r})$ using the RVF procedure exactly in the same way as above.

Take derivative after the transform and construct nlin

- `void IncVF::Derivative_RSprod_VV_ET()`: We compute $nlin_i = nlin_i + D_j \mathcal{F}(H_j V_{fi})$. As discussed in Chapter IncFlow, D_1 for real space products has odd parity.

9.4.5 Inverse transform of Vfrom

`void IncVF::ET_Inverse_transform_Vfrom()`: Performs the inverse transform of $\mathbf{V}_f$. Note that in SCFT basis V_{f1} has odd parity, and the other components have even parity.

9.5 Computation of nlin for energy transfer

The nonlinear terms for the energy transfer are of the type $nlin_i = D_j \mathcal{F}[H_j (V_f)_i]$ where $\mathbf{V}_f$ is the giver field, and $\mathbf{H}$ is the helper field. A important point to note that the helper field $\mathbf{H}$ could be either be $\mathbf{V}$ or another field $\mathbf{W}$. `EnergyTr_Compute_nlin()` has $\mathbf{V}$ as the helper field, while `EnergyTr_Compute_nlin(W)` has $\mathbf{W}$ as the helper field.

A detailed description of the functions are as follows.

9.5.1 IncVF::EnergyTr _Compute_nlin()

The function yields $nlin_i = D_j\mathcal{F}[V_j(V_f)_i]$. Here the field $\mathbf{V}$ is the helper field, and the giver $\mathbf{V}_f$ could be either $\mathbf{V}$ or $\mathbf{W}$.

1. Compute $\mathbf{V}(\mathbf{r})$ by Inverse_transform (`RVF::RV_Inverse_transform()`).
2. Compute $\mathbf{V}_f(\mathbf{r})$ by Inverse_transform (`IncVF::ET_Inverse_transform_Vfrom()`).
3. Compute the diagonal product $V_i(V_f)_i$; put the products in $nlin_i$, i.e., $nlin_i \leftarrow V_i(V_f)_i$ (`Compute_RSprod_diag()`). The diagonal products have even parity.
4. Perform forward transform of $nlin_s$ and take derivative, i.e., $nlin_i \leftarrow D_i\mathcal{F}[V_i(V_f)_i]$. Choose even parity derivative long x axis. The function is `NLIN_diag_Forward_transform_derivative()`.
5. Compute off-diagonal products $(V_f)_iV_j$ (`IncVF::Compute_RSprod_offdiagET()`). Both $\mathbf{V}(\mathbf{r})$ and $\mathbf{V}_f(\mathbf{r})$ contain the products.
6. `void RV_Forward_transform_RSprod()`: Forward transform the product V_is keeping parity in mind.
7. `void ET_Forward_RSprod_Vfrom()`:Forward transform the product $(V_f)_i$s keeping parity in mind.
8. `void Derivative_RSprod_ET()`: Compute derivative and add to *nlin* in the same way as in `IncVF::Derivative_RSprod()`.

The result of the operation is the desired $nlin_i = D_j\mathcal{F}[V_j(V_f)_i]$.

9.5.2 IncVF::EnergyTr _Compute_nlin(IncVF& W)

The function yields $nlin_i = D_j\mathcal{F}[W_j(V_f)_i]$. Here the field $\mathbf{W}$ is the helper field, and the giver $\mathbf{V}_f$ could be either $\mathbf{V}$ or $\mathbf{W}$. The procedure for this computation is very similar to the above computation, yet we describe it here for completeness.

1. Compute $\mathbf{V}(\mathbf{r})$ by Inverse_transform (`RVF::RV_Inverse_transform()`).
2. Compute $\mathbf{V}_f(\mathbf{r})$ by Inverse_transform (`IncVF::ET_Inverse_transform_Vfrom()`).
3. Compute the diagonal product $W_i(V_f)_i$; put the products in $nlin_i$, i.e., $nlin_i \leftarrow W_i(V_f)_i$ (`Compute_RSprod_diag()`). The diagonal products have even parity.
4. Perform forward transform of $nlin_i$ and take derivative, i.e., $nlin_i \leftarrow D_i\mathcal{F}[W_i(V_f)_i]$. Choose even parity derivative long x axis. The function is `NLIN_diag_Forward_transform_derivative()`.
5. Compute off-diagonal products $(V_f)_iW_j$ (`IncVF::Compute_RSprod_offdiagET()`). Both $\mathbf{V}(\mathbf{r})$ (not $\mathbf{W}(\mathbf{r})$) and $\mathbf{V}_f(\mathbf{r})$ contain the products.

6. `void RV_Forward_transform_RSprod()`: Forward transform the product V_is keeping parity in mind.

7. `void ET_Forward_RSprod_Vfrom()`: Forward transform the product $(V_f)_i$s keeping parity in mind.

8. `void Derivative_RSprod_ET()`: Compute derivative and add to *nlin* in the same way as in `IncVF::Derivative_RSprod()`.

The result is $nlin_i = D_j\mathcal{F}[W_j(V_f)_i]$.

9.5.3 `IncVF::EnergyTr_Compute_nlin(IncSF &T)`

For scalars, the nonlinear terms for the energy transfer is $nlin = D_i\mathcal{F}[V_i\zeta]$ where ζ is the giver field, and $\mathbf{V}$ is the helper field. The steps to compute the nonlinear term are as follows:

1. Compute $\mathbf{V}(\mathbf{r})$ by Inverse_transform (`RVF::RV_Inverse_transform()`).

2. Place ζ into V_{from1} and compute $\zeta(\mathbf{r})$ by Inverse_transform (`Inverse_transform_array(F)`).

3. Compute the products $nlin_i = V_i\zeta$. Here, $nlin_1$ has even parity, and $nlin_{2,3}$ have odd parity.

4. Perform forward transform of $nlin_i$ and take derivative, i.e., $nlin_i \leftarrow D_i\mathcal{F}[V_i\zeta]$. Choose even parity derivative long x axis, and odd parity along other two directions.

5. Compute derivatives and add them. $nlin_1$ contains $D_i\mathcal{F}[V_i\zeta]$.

9.5.4 `IncVF::EnergyTr_Compute_nlin(IncSF &T, string Pr_switch)`

If Prandtl number is zero, then we do not need to compute nonlinear terms $(\mathbf{u}\cdot\nabla)\zeta$. When Prantdl number is nonzero, we use `IncVF::EnergyTr_Compute_nlin(T)` to compute the nonlinear terms.

9.6 Computation of isotropic flux

Now we can use all our auxiliary functions to compute the energy flux from a wavenumber sphere of radius K_0. We discuss the functions for fluid, MHD, and scalar turbulence.

9.6.1 Fluid turbulence

The energy flux $\Pi^{u<}_{u>}(K_0)$ for fluid turbulence was defined in Sec.... We compute the energy flux $\Pi^{u<}_{u>}(K_0)$ in the following steps.

- We fill the field $\mathbf{V}_f$ within the sphere excluding the surface of the sphere at the radius K_0. The field $\mathbf{V}_f$ is zero outside the sphere. This function is performed by `Fill_in_sphere(sphere_index)`. This operation implies that the giver field is *strictly* inside the sphere K_0.

- We compute the nonlinear term $nlin_i = D_j T[V_j(V_f)_i]$ using the function `EnergyTr_Compute_nlin()`. Here V_j is acting as a helper.

- The flux is
$$\Pi^{u<}_{u>}(K_0) = \sum_{k>K^0} -\Re[u_i^*(\mathbf{k}) \times nlin_i^{uu}(\mathbf{k})].$$
This operation is performed using `Prod_out_sphere_nlinV(sphere)`.

The above quantity is the energy transfer from all the modes strictly within the wavenumber sphere to the modes outside the sphere including the sphere's surface.

We compute the energy flux for all the spheres. The fluxes obey some interesting properties. The energy flux for the last sphere is zero because no energy is flowing outside the last sphere. Note however that the energy flux from the penultimate sphere is nonzero.

9.6.2 Scalar

For scalar field ζ we compute the energy flux $\Pi^{\zeta<}_{\zeta>}(K_0)$ in the following steps:

- we fill the field $(V_f)_1$ within the sphere with the scalar field. The field ζ is zero outside the sphere. This function is performed by `Fill_in_sphere(sphere_index, zeta)`.

- We compute the nonlinear term $nlin = D_i \mathcal{F}[V_i \zeta]$ using the function `EnergyTr_Compute_nlin(T)`. Here V_j is acting as a helper.

- The flux is
$$\Pi^{\zeta<}_{\zeta>}(K_0) = \sum_{k>K_0} -\Re[\zeta^*(\mathbf{k}) \times nlin^{\zeta\zeta}(\mathbf{k})].$$
This operation is performed using `Prod_out_sphere_nlinV(sphere, zeta)`.

We compute the energy flux for all the spheres.

9.6.3 MHD

For MHD turbulence we compute

- $\Pi^{u<}_{u>}$ using the method described above.

- $\Pi^{u<}_{b>}$: Here $nlin^{u,u<}$ is multiplied by $b^{>}$.

- $\Pi^{b<}_{b>}$: Here $nlin^{u,b<}$ is multiplied by $b^{>}$.

- $\Pi^{b<}_{u>}$: Here $nlin^{u,b<}$ is multiplied by $u^{>}$.
- $\Pi^{b<}_{u<}$: Here $nlin^{u,b<}$ is multiplied by $u^{<}$.
- $\Pi^{u>}_{b>}$: Here $nlin^{b,u>}$ is multiplied by $b^{>}$.
- $\Pi^{z^+<}_{z^+>}$: Here $nlin^{z^-,z^+}$ is multiplied by $z^{+>}$.
- $\Pi^{z^{+-}<}_{z^->}$: Here $nlin^{z^+,z^-}$ is multiplied by $z^{->}$.

9.6.4 RB Convection

For nonzero Pr we compute both $\Pi^{u<}_{u>}$ and $\Pi^{\zeta<}_{\zeta>}$. However for zero Pr, we compute only $\Pi^{u<}_{u>}$. We use the `Pr_switch` for the above selection.

9.7 Compute Shell-to-shell transfer

Intro...

9.7.1 Fluid

We compute the shell-to-shell energy transfer T^{uu}_{nm} in the following way:

- We fill the field $\mathbf{V}_f$ in wavenumber shell m. The modes on the inner surface of the shell is included, but not the ones on the outer surface. All other modes are set to zero. This function is performed by `Fill_shell(m)`.
- We compute the nonlinear term $nlin_i = D_j\mathcal{F}[V_j(V_f)_i]$ using the function `EnergyTr_Compute_nlin()`. Here V_j is acting as a helper.
- The shell-to-shell transfer $m \to n$ is

$$T^{uu}_{nm} = \sum_{k\in n} -\Re[u^*_i(\mathbf{k}) \times nlin^{uu}_i(\mathbf{k})].$$

  This operation is performed using `Prod_shell_nlinV(sphere)`. While computing the product_shell_nlinV, we include the inner surface of the shell n and exclude the outer surface of the shell n.
- For a given m, we perform T^{uu}_{nm} for all ns. This is the efficient way of computing the shell-to-shell transfers because we need to compute *nlin*s only once for each m.
- We repeat the above process for all m shells.

9.7.2 MHD

There are more types of shell-to-shell energy transfers in MHD turbulence. They are T^{uu}_{nm}, T^{bb}_{nm}, T^{bu}_{nm} $(b \leftarrow u)$. The u-to-u transfer is computed as above. While b-to-b transfer is computed using

$$T^{bb}_{nm} = \sum_{k \in n} -\Re[b^*_i(\mathbf{k}) \times nlin^{ub}_i(\mathbf{k})],$$

and u-to-b transfer is computed using

$$T^{bu}_{nm} = \sum_{k \in n} -\Re[b^*_i(\mathbf{k}) \times nlin^{bu}_i(\mathbf{k})].$$

The nonlinear terms are computed using appropriate functions.

9.7.3 Scalar turbulence

For scalar turbulence, we compute shell-to-shell energy transfers T^{uu}_{nm} and $T^{\zeta\zeta}_{nm}$. The u-to-u transfer is computed as described above. The ζ-to-ζ transfer is computed using

$$T^{bb}_{nm} = \sum_{k \in n} -\Re[\zeta^*(\mathbf{k}) \times nlin^{\zeta\zeta}(\mathbf{k})].$$

We compute nonlinear term term $nlin^{\zeta\zeta}(\mathbf{k})$ as described in Sec.

9.7.4 Rayleigh Benard convection

For nonzero Prandtl number we compute both T^{uu}_{nm} and $T^{\zeta\zeta}_{nm}$. However for zero Pr we only compute T^{uu}_{nm}.

9.8 Energy input from the forcing

Vector field $\mathbf{V}$ and scalar field ζ receive energy from their respective forcing $\mathbf{F^v}$ and F^ζ . We compute these energy transfers for each shell mentioned above. The energy input to the fields $\mathbf{V}$ and ζ in wavenumber shell n are $\sum_{k \in n} \Re[\mathbf{F^v}(\mathbf{k}).\mathbf{V}^*(\mathbf{k})]$ and $\sum_{k \in n} \Re[F^\zeta(\mathbf{k})\zeta^*(\mathbf{k})]$ respectively. These quantities are computed by functions Compute_force_feed(). These functions use `Prod_shell_forceV()` to compute the above sum.

Chapter 10

IncFluid: Library for Incompressible Fluid

This is the main library that interfaces with the main program. It contains a class `IncFluid` that inherits class `IncVF` described earlier, class `Time`, and functions for time-advance, and input output operations. As described earlier, the class `IncVF` contains important functions like computations of nonlinear terms `nlin`, pressure, energy fluxes, and shell-to-shell transfers. The class `Time` contains variables for initial, current, and file time, as well as the iterations at which we save output fields. In addition to

In the following discussion we will describe the class `IncFluid` and the class `Time` in some detail.

10.1 Class Time

The public variables in this class are

- `DP Tinit, Tfinal:` Intial and final time of the simulation.
- `DP Tdt:` Time-step variable
- `DP Tnow:` Current time
- `DP Tdiagnostics_init:` Diagnostic computation of the spectrum, the energy flux, and the shell-to-shell energy transfer starts after Tdiag_init. At present this feature is not being used.
- `int Tglobal_save:` Saves global variables like energy every Tglobal_save iterations.
- `int Tfield_save:` Saves fields at every Tfield_save iterations.
- `int Trealfield_save:` Saves fields at every Trealfield_save iterations.

- `int Tfield_reduced_save:` Saves reduced_field (smaller size) every Tfield_reduced_save iterations.
- `int Tfield_k_save:` Saves the fileld variables at specified wavenumbers (`*output_k_array`) every Tfield_k_save iterations.
- `int Tspectrum_save:` Saves the spectrum of fields at every Ttriad_save iterations.
- `int Tflux_save:` Saves the energy fluxes at every Tflux_save iterations.
- `int Tshell_to_shell_save:` Saves the shell-to-shell transfers at every Tshell_to_shell_save iterations.
- `int Tcout_save:` Prints some global variables every Tcout_save iters to indicate the status of the program.

The constructor of the class Time uses array time_para and time_save.

`Time(Array<DP,1> time_para, Array<int,1> time_save):` The variables `Tinit, Tfinal, Tdt,` and `Tdiagnostics_init` are taken from `time_para(1..4)`, while the variables `Tglobal_save`, `Tfield_save, Trealfield_save`, `Tfield_reduced_save, Tfield_k_save, Tspectrum_save, Tflux_save, Tshell_to_shell_save,` and `Tcout_save` are taken from `time_save(1..9)`.

10.2 IncFluid Class

This class contains the following public variables:

- `string data_dir_name:` The name of the directory where the input field and the output fields are saved.
- `string no_output_mode:` The mode of the output numbers (either ASCII or BINARY).
- `ifstream field_in_file:` Contains the field configurations to be read as *initial conditions* in init_cond().
- `ofstream global_file:` Global variables outputted here.
- `ofstream field_out_file:` The fields like $\mathbf{V}(\mathbf{k})$ outputted here.
- `ofstream realfield_out_file:` The fields in real space like $\mathbf{V}(\mathbf{r})$ outputted here.
- `ofstream field_out_reduced_file:` The field in reduced dimension is outputted here.
- `ofstream field_k_out_file:` The field variables for a given set of variables (`*output_k_array`) is outputted here.

- `ofstream spectrum_file:` The spectra of the fields are outputted here.
- `ofstream flux_file:` The energy fluxes of the fields are outputted here.
- `ofstream shell_to_shell_file:` The shell-to-shell transfers are outputted here.
- `ofstream pressure_file:` The pressure spectrum is outputted here.
- `int nos_output_triads:` Number of wavenumbers at which the field variables are outputted.
- `Array<int,2> *triad_array_output:` The wavenumbers at which the field variables are outputted. The constructor allocates `*triad_array_output(nos_output_triad+1 4)` where (k_x, k_y, k_z) = `triad_array_output(i,1..3)`. Naturally k_z exists only for 3D simulations.
- `string integ_scheme:` Integrating scheme for time stepping. It could take values EULER, RK2, or RK4.

The functions in IncFluid class are

10.2.1 Compute and add force

Force function is present in many spectral simulation. We denote the forcing for the velocity $\mathbf{V}$, scalar ζ, and vector $\mathbf{W}$ by $\mathbf{f^v}$, f^ζ, and $\mathbf{f^w}$ respectively. Since the force function appears in the rhs of the evolution equations, and nlin is present in the left hand side of the equation, we add the force to nlin using

$$\begin{aligned} \mathbf{nlin^v} &= \mathbf{nlin^v} - \mathbf{f^v}, \\ nlin^\zeta &= nlin^\zeta - f^\zeta, \\ \mathbf{nlin^W} &= \mathbf{nlin^W} - \mathbf{f^W}. \end{aligned}$$

The relevant functions are Compute_force(), and Add_force().

The following form of forcing has been implemented in the spectral code.

10.2.2 RB convection

In RB convection, the buoyancy term acts as a forcing in the equation for velocity, and V_1 acts as forcing for temperature field. However these terms depend on the nondimensionalization procedure adopted. We have the following scenarios (see Appendix):

- *Pr_switch = PRLARGE,* and *RB_Uscaling = USMALL:* $\mathbf{f^v} = R \times Pr \times \zeta\hat{\mathbf{z}}$, $f^\zeta = V_1$.
- *Pr_switch = PRLARGE,* and *RB_Uscaling = ULARGE:* $\mathbf{f^v} = \zeta\hat{\mathbf{z}}$, $f^\zeta = V_1$.

- *Pr_ switch = PRSMALL,* and *RB_ Uscaling = USMALL:* $\mathbf{f^v} = R\zeta\hat{\mathbf{z}}$, $f^\zeta = \frac{1}{Pr}V_1$.
- *Pr_ switch = PRSMALL,* and *RB_ Uscaling = ULARGE:* $\mathbf{f^v} = Pr \times \zeta\hat{\mathbf{z}}$, $f^\zeta = \frac{1}{Pr}V_1$.

10.3 Add_pressure_gradient()

The pressure in the NS equation is computed using the equation

$$div(\mathbf{nlin} - \mathbf{f}) = -\nabla^2 p.$$

This task is done using the function `Compute_pressure`(). For computing the rhs of the evolution equation for the velocity, we add pressure gradient Dp to **nlin**, i.e.,

$$\mathbf{nlin} = \mathbf{nlin} + \mathbf{D}p.$$

Note that $\mathbf{rhs} = -\mathbf{nlin} - \mathbf{D}p$.

Since p has even parity, Xderiv is provided with 0 option for the parity.

10.4 Compute_rhs()

We compute the rhs of all the equations using these functions. Clearly

$$\mathbf{rhs}^{\mathbf{V},\zeta,\mathbf{W}} = -\mathbf{nlin}^{\mathbf{V},\zeta,\mathbf{W}}.$$

Note that $\mathbf{nlin}^{\mathbf{V},\zeta,\mathbf{W}}$ contains $-\mathbf{f}^{\mathbf{v},\zeta,\mathbf{w}}$ along with the nonlinear terms like $D_j\mathcal{F}[V_jV_i]$ and the pressure gradient $\mathbf{D}p$.

For RB convection, $\mathbf{rhs}^\zeta$ is not computed when $Pr = 0$.

10.5 Single_time_step

We have implemented three kinds of Single_time_step functions.

- `void IncFluid::Single_time_step_EULER(DP dt):` Here the field is advanced to time $t + dt$ using the slope computed at t.

  $$\mathbf{V}(\mathbf{k}, t + dt) = [\mathbf{V}(\mathbf{k}, t) + dt \times \mathbf{rhs}(\mathbf{k}, t)] \exp(-\nu K^2 dt).$$

  For the scalar field ζ

  $$\zeta(\mathbf{k}, t + dt) = [\zeta(\mathbf{k}, t) + dt \times rhs(\mathbf{k}, t)] \exp(-\kappa K^2 dt).$$

- `void IncFluid::Single_time_step_RK2(DP dt):` Here the field is advanced to time $t + dt$ using the slope at the midpoint $t = t + dt/2$.

  $$\mathbf{V}(\mathbf{k}, t{+}dt) = [\mathbf{V}(\mathbf{k}, t) \exp(-\nu K^2 dt/2){+}dt{\times}\mathbf{rhs}(\mathbf{k}, t{+}dt/2)] \exp(-\nu K^2 dt/2).$$

  For the scalar field ζ, replace $\mathbf{V}$ by ζ and ν by κ.

- `void IncFluid::Single_time_step_Semi_implicit(DP dt)`: In this scheme, the field is advanced to time $t + dt$ using the slope at the time $t = t + dt$:

$$\mathbf{V}(\mathbf{k}, t + dt) = \mathbf{V}(\mathbf{k}, t) \exp(-\nu K^2 dt) + dt \times \mathbf{rhs}(\mathbf{k}, t + dt).$$

For RB-convection, the procedure is same as scalar if $Pr \neq 0$. For $Pr = 0$, the scalar function ζ is computed from V_1 using

$$\zeta(\mathbf{k}) = \frac{V_1(\mathbf{k})}{K^2}. \tag{10.1}$$

10.6 Time advancing of fields

We have implemented three time-advancing schemes: Euler, second-order Runge-Kutta, and fourth-order Runge Kutta.

10.6.1 Time advance for fluid

We use the function `void Time_advance()` for advancing the velocity field to time $t = t+dt$. Before entering this function, $nlin_i = D_j \mathcal{F}[V_j V_i]$ and F contains the pressure. The time advance could be in one of the above schemes depending on the variable `integ_scheme.`

For time-advancing using EULER scheme (`integ_scheme=EULER`) the steps are

1. `Compute_rhs().`
2. `Single_time_step_EULER(dt).`

For RK2 scheme (`integ_scheme=RK2`), the steps are

1. Save $\mathbf{V}$ in $\mathbf{Vcopy}$, i.e., $\mathbf{Vcopy} = \mathbf{V}$.
2. Go to the midpoint
   - Compute_rhs().
   - Go to the mid point using `Single_time_step_EULER(dt/2).`
3. Compute RHS at the midpoint and time advance to time $t + dt$.
   - `Compute_force().`
   - `Compute_nlin()` using the fields at the mid point.
   - `Add_force().`
   - `Compute_pressure().`
   - `Compute_rhs().`
   - Restore $\mathbf{V}$ from $\mathbf{Vcopy}$, i.e., $\mathbf{V} = \mathbf{Vcopy}$.

- `Single_time_step_RK2(dt)` using the rhs computed using the fields at the midpoint.

For RK4 scheme (`integ_scheme=RK4`) the steps are

1. Save $\mathbf{V}$ in $\mathbf{Vcopy}$, i.e., $\mathbf{Vcopy} = \mathbf{V}$.
2. **RK4-Step 1:** Compute $\mathbf{V}_{mid1}(t+dt/2)$ using Euler's scheme. Also compute C_1 defined below.
   - Compute_rhs().
   - Compute $\mathbf{V}_{mid1}(t+dt/2)$ at the midpoint using `Single_time_step_EULER(Tdt/2)`.
   - Compute $\mathbf{C}_1 = \mathbf{rhs}(t)\exp(-\nu K^2 dt) \times dt$.
3. **RK4-Step 2:** Compute RHS using $\mathbf{V}_{mid1}$ and time advance to time $t + dt/2$ using this rhs. This is a semi_implicit step. Also compute C_2 defined below.
   - `Compute_force()`.
   - `Compute_nlin()` using the fields at the mid point.
   - `Add_force()`.
   - `Compute_pressure()`.
   - `Compute_rhs()`.
   - Restore $\mathbf{V}$ from $\mathbf{Vcopy}$, i.e., $\mathbf{V} = \mathbf{Vcopy}$.
   - Compute $\mathbf{V}_{mid2}(t+dt/2)$ at the midpoint using `Single_time_step_Semi_implicit(Tdt/2)`.
   - Compute $\mathbf{C}_2 = \mathbf{rhs}(t)\exp(-\nu K^2 dt/2) \times dt$.
4. **RK4-Step 3:** Compute RHS using $\mathbf{V}_{mid2}$ and time advance to time $t+dt$ using `Single_time_step_RK2(Tdt)`. Also compute C_3 defined below.
   - `Compute_force()`.
   - `Compute_nlin()` using the fields at the mid point.
   - `Add_force()`.
   - `Compute_pressure()`.
   - `Compute_rhs()`.
   - Restore $\mathbf{V}$ from $\mathbf{Vcopy}$, i.e., $\mathbf{V} = \mathbf{Vcopy}$.
   - Compute $\mathbf{V}_3(t+dt)$ at time $t+dt$ using `Single_time_step_RK2(Tdt)`.
   - Compute $\mathbf{C}_3 = \mathbf{rhs}(t)\exp(-\nu K^2 dt/2) \times dt$.
5. **RK4-Step 4:** Compute RHS using $\mathbf{V}_{mid2}$ and time advance to time $t+dt$ using `Single_time_step_RK2(Tdt)`. Also compute $\mathbf{C}_4$ defined below.
   - `Compute_force()`.

- `Compute_nlin()` using the fields at the mid point.
- `Add_force()`.
- `Compute_pressure()`.
- `Compute_rhs()`.
- Compute $\mathbf{C}_4 = \mathbf{rhs}(t) \times dt$.
- Restore $\mathbf{V}$ from $\mathbf{Vcopy}$, i.e., $\mathbf{V} = \mathbf{Vcopy}$.
- Compute the final velocity field $\mathbf{V}(t + dt)$ using

$$\mathbf{V}(\mathbf{k}, t + dt) = \mathbf{V}(\mathbf{k}, t) \exp(-\nu K^2 dt) + \frac{1}{6}(\mathbf{C}_1 + 2\mathbf{C}_2 + 2\mathbf{C}_3 + \mathbf{C}_4).$$

10.6.2 Time Advance for velocity and scalar field

The procedure for time advancing a scalar field is the same as the velocity field. We perform the similar functions on the scalar field as well. The nonlinear terms are computed using `Compute_nlin(T)`.

10.6.3 Time advance for velocity and magnetic field (MHD)

For MHD, the procedure is the same as that given for the velocity field. We advance both velocity and magnetic field. Typically, the magnetic field is not forced. The nonlinear terms are computed using `Compute_nlin(W)`.

10.6.4 Time Advance for RB Convection

For RB convection, the procedure is identical to that for scalar if $Pr \neq 0$. However, when $Pr = 0$, the temperature field is computed from in the last step (Single_time_step) from V_1 field using Eq. (10.1). The forcing for the velocity and the temperature field is computed using `Compute_force(T, Ra, Pr, Pr_switch, RB_Uscaling)`.

Follow the similar procedure for magnetoconvection.

10.7 Input Output operations

We describe the input output operations in the next chapter.

Chapter 11

Input and output in IncFluid

In this chapter we will describe how the initial conditions are read, and various quantities are written in output files.

11.1 Files and file operations

As described in the chapter on `IncFluid,` the input and output files are resident in the directory `data_dir_name`. The initial condition is read from the file `field_in_file` that is in the directory `data_dir_name/in`. Various quantities are sent as outputs to various output files. The following output files are resident in the director `data_dir_name/out`.

- `global_file`: Global variables
- `field_out_file`: Field configurations in Fourier space or SCF space
- `realfield_out_file`: Field configuration in real space.
- `field_out_reduced_file`: Smaller field configuration in Fourier space or SCF space.
- `field_k_out_file`: Field variables for a given set of wavenumbers (`*output_k_array`)
- `spectrum_file`: Energy spectrum
- `flux_file`: Energy flux
- `shell_to_shell_file`: Shell-to-shell energy trasfers
- `pressure_file`: Pressure spectrum.

We have the following functions for the file operations.

- `void Open_input_files()`: Opens the file `field_in_file` at `data_dir_name/in/field_in.d`. This file contains the initial field configurations.

- `void Close_input_files()`: Closes the input file `field_in_file`.
- `void Output_prefix(ofstream& X, string prefix_str)`: The string `prefix_str` is sent to the file `X` along with the grid size, initial, final, and timestep.
- `void Open_output_files()`: Opens the output files `global_file, field_out_file` etc. in the directory `data_dir_name/out`. We invoke `Output_prefix(..)` for all these files.
- `void Close_output_files()`: Close all the output files.

We first discuss how we read the initial configuration of the fields.

11.2 Init_cond()

These functions initialize the fields. They are

- `void Init_cond(), void Init_cond(IncSF& T), void Init_cond(IncVF& W), void Init_cond(IncVF& W, IncSF& T)`: Reads values of the fields from file `field_in_file`. The size of the read-fields is same as **N** of IncFluid.
- `void Init_cond(int Nreduced[])`: The input file contains $\bar{\mathbf{V}}(\mathbf{Nreduced})$ that is fed in the lower part of $\mathbf{V}(\mathbf{N})$. Similar operations for additional fields like T and W.
- `void Init_cond(string field_in_k_type)`:
  - For `field_in_k_type==SIMPLE`: Read ($\mathbf{k}$, V_x) in 2D and ($\mathbf{k}$, V_x, V_y) in 3D. The last component is computed using the function `Last_component(...)`. Complex conjuage of the above is added in the field if $k_y = 0$ in 2D (in FOUR basis) and $k_z = 0$ in 3D.
  - For `field_in_k_type==VORTICITY`: Read ($\mathbf{k}$, V_x, Ω_x) in 3D. The components V_y and V_z are computed using function `Compute_VyVz(...)`. Complex conjuage of the above is added in the field if $k_y = 0$ in 2D (in FOUR basis) and $k_z = 0$ in 3D.
  - Similar procedure is adopted in the presence of scalar and another vector field.
- `void Init_cond(IncSF& T, DP W101, DP T101, DP T200)`: In 2D, $V_x(11) = W101$, $T(11) = T101$, and $T(20) = T200$. In 3D, add $V_x(101) = W101$, $V_y(101) = 0$, $T(101) = T101$, and $T(200) = T200$. The last component $V_y(11)$ in 2D or $V_z(101)$ in 3D are found using the function `Last_component(...)`.
- `void Init_cond(IncSF& T, string Pr_switch)`: This function is used for reading initial condition for RB convection. If Pr is equal to zero, then we call `Init_cond()` to read the velocity field **V**, and compute $T(\mathbf{k}) = V_1(\mathbf{k})/K^2$. When Pr is not equal to zero, call `Init_cont(T)`. Other functions for RB convection have similar structure.

11.3 Output Results

We output the results of our simulations in various files. These files are declared in class `IncFluid`. The functions that write on these files are

- `void Output_prefix(ofstream& X, string prefix_str)`: prefix_str containing parameter values are added at the beginning of the output files. This is invoked inside the function `Open_output_files(prefix_str)`.

1. Global variables

   - `void Output_global()`: t, $\frac{1}{2}\sum |V(\mathbf{k})|^2$, $\nu\frac{1}{2}\sum K^2|V(\mathbf{k})|^2$. Note however that ($\mathbf{k}=0$) is note included.
   - `void Output_global(IncSF& T)`: t, $\frac{1}{2}\sum |V(\mathbf{k})|^2$, $\frac{1}{2}\sum |F(\mathbf{k})|^2$, $\nu\sum K^2|V(\mathbf{k})|^2$, $\kappa\sum K^2|F(\mathbf{k})|^2$.
   - `void Output_global(IncVF& W)`: t, $\frac{1}{2}\sum |V(\mathbf{k})|^2$, $\frac{1}{2}\sum |B(\mathbf{k})|^2$, $\nu\sum K^2|V(\mathbf{k})|^2$, $\frac{1}{2}\eta\sum K^2|B(\mathbf{k})|^2$, $\Re\frac{1}{2}\sum \mathbf{V}(\mathbf{k})\cdot\mathbf{B}^*(\mathbf{k})$.
   - `void Output_global(IncSF& T, string Pr_switch)`: t, $\frac{1}{2}\sum |V(\mathbf{k})|^2$, $\frac{1}{2}\sum |F(\mathbf{k})|^2$, $\nu\sum K^2|V(\mathbf{k})|^2$, $\kappa\sum K^2|F(\mathbf{k})|^2$, Nusselt number $(1+\Re(V_1(\mathbf{k})F^*(\mathbf{k})))$.
   - `void Output_global(IncVF& W, IncSF& T, string Pr_switch)`: t, $\frac{1}{2}\sum |V(\mathbf{k})|^2$, $\frac{1}{2}\sum |B(\mathbf{k})|^2$, $\frac{1}{2}\sum |F(\mathbf{k})|^2$, $\nu\sum K^2|V(\mathbf{k})|^2$, $\eta\sum K^2|B(\mathbf{k})|^2$, $\kappa\sum K^2|F(\mathbf{k})|^2$, $\Re\frac{1}{2}\sum \mathbf{V}(\mathbf{k})\cdot\mathbf{B}^*(\mathbf{k})$, Nusselt number.

2. Fields

   - `void Output_field()`:
     - 2D: $V_1(\mathbf{k})$, $V_2(\mathbf{k})$;
     - 3D: $V_1(\mathbf{k})$, $V_2(\mathbf{k})$, $V_3(\mathbf{k})$.
   - `void Output_field(IncSF& T)`:
     - 2D: $V_1(\mathbf{k})$, $V_2(\mathbf{k})$, $F(\mathbf{k})$;
     - 3D: $V_1(\mathbf{k})$, $V_2(\mathbf{k})$, $V_3(\mathbf{k})$, $F(\mathbf{k})$.
   - `void Output_field(IncVF& W)`:
     - 2D: $V_1(\mathbf{k})$, $V_2(\mathbf{k})$, $W_1(\mathbf{k})$, $W_2(\mathbf{k})$;
     - 3D: $V_1(\mathbf{k})$, $V_2(\mathbf{k})$, $V_3(\mathbf{k})$, $W_1(\mathbf{k})$, $W_2(\mathbf{k})$, $W_3(\mathbf{k})$
   - `void Output_field(IncVF& W, IncSF& T)`:
     - 2D: $V_1(\mathbf{k})$, $V_2(\mathbf{k})$, $W_1(\mathbf{k})$, $W_2(\mathbf{k})$,$F(\mathbf{k})$
     - 3D: $V_1(\mathbf{k})$, $V_2(\mathbf{k})$, $V_3(\mathbf{k})$, $W_1(\mathbf{k})$, $W_2(\mathbf{k})$, $W_3(\mathbf{k})$,$F(\mathbf{k})$.
   - `void Output_field(IncSF& T, string Pr_switch)`: If $Pr \neq 0$, `Output_field(T)`, else `Output_field()`.
   - `void Output_field(IncVF& W, IncSF& T, string Pr_switch)`: If $Pr \neq 0$, `Output_field(W, T)`, else `Output_field(W)`.

3. Fields in a reduced box

   - `void Output_field_reduced(int Nreduced[])`: Same as above but for $\mathbf{V}(\mathbf{Nreduced})$.

4. Real field

   - `void Output_realfield()`: Same as above but fields in real space.

5. Isotropic spectrum

   - `void Output_isotropic_spectrum()`: Isotropic energy spectrum–
     - 2D: K, $e_1(K)$, $e_2(K)$, $e^{force_v}(K)$, $D^v(K)$ for $0 \leq K < K_{max}$ in steps of one where $e^{force_u}(K) = \Re(\mathbf{F}(\mathbf{k}) \cdot [\mathbf{v}(\mathbf{k})]^*)$, and $D^u(K) = \nu|\mathbf{v}(K)|^2$.
     - 3D: K, $e_1(K)$, $e_2(K)$, $e_3(K)$, $e^{force_v}(K)$, $D^v(K)$ for $0 \leq K < K_{max}$.
   - `void Output_isotropic_spectrum(IncSF& T)`: 2D: K, $e_1(K)$, $e_2(K)$, $e^F(\mathbf{k})$, $e^{force_v}(K)$, $e^{force_F}(K)$, $D^v(K)$, $D^F(K)$. 3D: add $e_3(K)$.
   - `void Output_isotropic_spectrum(IncVF& W)`: 2D: K, $e_1(K)$, $e_2(K)$,$e_1^W(K)$, $e_2^W(K)$, $H_c(K)$, $e^{force_v}(K)$, $e^{force_W}(K)$, $D^v(K)$, $D^W(K)$. 3D: add $e_3(K)$.
   - `void Output_isotropic_spectrum(IncVF& W, IncSF &T)`: 2D: K, $e_1(K)$, $e_2(K)$,$e_1^W(K)$, $e_2^W(K)$, $e^F(K)$, $H_c(K)$, $e^{force_v}(K)$, $e^{force_W}(K)$, $e^{force_F}(K)$, $D^v(K)$, $D^W(K)$, $D^F(K)$.
   - `void Output_isotropic_spectrum(IncSF& T, string Pr_switch)`: If $Pr \neq 0$, `Output_isotropic_spectrum(T)`, else `Output_isotropic_spectrum()`.
   - `void Output_isotropic_spectrum(IncVF& W, IncSF& T, string Pr_switch)`: If $Pr \neq 0$, `Output_isotropic_spectrum(W,T)`, else `Output_isotropic_spectrum(W)`.
   - `void Output_pressure_spectrum(string prefix_str)`: Outputs pressure spectrum $|p(K)|^2/2$ in pressure_file.

6. Energy flux

   - `void Output_flux(int real_imag_switch)`: sphereindex, iso_flux_self, `iso_forceV,` iso_flux_self_real, iso_flux_self_imag. (real and imag depending on a switch: 1 if yes)
   - `void Output_flux(IncSF& T, int real_imag_switch)`: sphereindex, iso_flux_self, iso_flux_SF, iso_forceV, iso_forceSF, and real and imag if the switch is 1.
   - `void Output_flux(IncVF& W, int real_imag_switch)`: sphereindex, iso_flux_self, iso_flux_VF_in_out, iso_flux_VF_in_in, W.iso_flux_self, W.iso_flux_VF_in_out, W.iso_flux_VF_in_in, W.iso_flux_VF_out_out, iso_flux_Elsasser, W.iso_flux_Elsasser, iso_forceV, W.iso_forceV, and real and imag if the switch is 1.

- `void Output_flux(IncVF& W, IncSF &T, int real_imag_switch)`: sphereindex, iso_flux_self, iso_flux_VF_in_out, iso_flux_VF_in_in, W.iso_flux_self, W.iso_flux_VF_in_out, W.iso_flux_VF_in_in, W.iso_flux_VF_out_out, iso_flux_SF, iso_flux_Elsasser, W.iso_flux_Elsasser, iso_forceV, W.iso_forceV, iso_forceSF, and real and imag if the switch is 1.
- `void Output_flux(IncSF& T, string Pr_switch, int real_imag_switch)`: If $Pr \neq 0$, `Output_flx(T, real_imag_switch)`, else `Output_flx(real_imag_switch)`.
- `void Output_flux(IncVF& W, IncSF& T, string Pr_switch, int real_imag_switch)`: If $Pr \neq 0$, `Output_flx(W, T, real_imag_switch)`, else `Output_flx(W, real_imag_switch)`.

7. Shell to shell energy transfers

   - `void Output_shell_to_shell(real_imag_switch)`: shell_to_shell_self, and real, imag if the switch is 1. The range is 1-Nshell. The last shell is the Maxrad to ∞.
   - `void Output_shell_to_shell(IncSF& T, real_imag_switch)`: shell_to_shell_self, shell_to_shell_SF, and real and imag if the switch is 1.
   - `void Output_shell_to_shell(IncVF& W, real_imag_switch)`: shell_to_shell_self, W.shell_to_shell_self, shell_to_shell_VF, shell_to_shell_Elsasser, W.shell_to_shell_Elsasser, and real and imag if the swithc is 1.
   - `void Output_shell_to_shell(IncVF& W, IncSF& T, real_imag_switch)`: shell_to_shell_self, W.shell_to_shell_self, shell_to_shell_VF, shell_to_shell_Elsasser, W.shell_to_shell_Elsasser, shell_to_shell_SF, and real and imag if the switch is 1.
   - `void Output_shell_to_shell(IncSF& T, string Pr_switch, real_imag_switch)`: If $Pr \neq 0$, `Output_shell_to_shell(T, real_imag_switch)`, else `Output_shell_to_shell(real_imag_switch)`.
   - `void Output_shell_to_shell(IncVF& W, string Pr_switch, IncSF& T, real_imag_switch, string Pr_switch)`: If $Pr \neq 0$, `Output_shell_to_shell(W, T, real_imag_switch)`, else `Output_shell_to_shell(W,real_imag_switch)`.

8. Field at a given wavenumbers

   - `void Output_field_k(string prefix_str)`: Each lines contains a single wavenumber.
     - 2D: t, k_x, k_y, $V_x(\mathbf{k})$, $V_y(\mathbf{k})$, $T^u(\mathbf{k}) = \Re(\mathbf{nlin}(\mathbf{k}) \cdot conj(\mathbf{V}(\mathbf{k})))$
     - 3D: t, k_x, k_y, k_z, $V_x(\mathbf{k})$, $V_y(\mathbf{k})$, $V_z(\mathbf{k})$, $T^u(\mathbf{k}) = \Re(\mathbf{nlin}(\mathbf{k}) \cdot conj(\mathbf{V}(\mathbf{k})))$
   - `void Output_field_k(IncSF& T)`: 2D: t, k_x, k_y, $V_x(\mathbf{k})$, $V_y(\mathbf{k})$, $F(\mathbf{k})$ $T^u(\mathbf{k})$, $T^F(\mathbf{k}) = \Re(T.nlin(\mathbf{k}) \cdot conj(F(\mathbf{k})))$. 3D: add third components

- `void Output_field_k(IncVF& W)`: t, k_x, k_y, $V_x(\mathbf{k})$, $V_y(\mathbf{k})$, $W_x(\mathbf{k})$, $W_y(\mathbf{k})$, $T^u(\mathbf{k})$, $T^W(\mathbf{k})$.
- `void Output_triad(IncVF& W, IncSF& T)`: t, k_x, k_y, $V_x(\mathbf{k})$, $V_y(\mathbf{k})$, $W_x(\mathbf{k})$, $W_y(\mathbf{k})$, $F(\mathbf{k})$, $T^u(\mathbf{k})$, $T^W(\mathbf{k})$, $T^F(\mathbf{k})$.
- `void Output_field_k(IncSF& T, string Pr_switch)`: If $Pr \neq 0$, `Output_field_k(T)`, else `Output_field_k()`.
- `void Output_field_k(IncVF& W, IncSF& T, string Pr_switch)`: If $Pr \neq 0$, `Output_field_k(W, T)`, else `Output_field_k(W)`.

9. Output on stdio

   - `void Output_cout()`: t, $\frac{1}{2}\sum |V(\mathbf{k})|^2$
   - `void Output_cout(IncSF& T)`: t, $\frac{1}{2}\sum |V(\mathbf{k})|^2$, $\frac{1}{2}\sum |F(\mathbf{k})|^2$
   - `void Output_cout(IncVF& W)`: t, $\frac{1}{2}\sum |V(\mathbf{k})|^2$, $\frac{1}{2}\sum |W(\mathbf{k})|^2$
   - `void Output_cout(IncVF& W, IncSF& T)`: t, $\frac{1}{2}\sum |V(\mathbf{k})|^2$, $\frac{1}{2}\sum |W(\mathbf{k})|^2$, $\frac{1}{2}\sum |F(\mathbf{k})|^2$.

10. Output all functions in the loop

    - `void Output_all_inloop(int iter, int Nreduced[], int shell_real_imag_switch)`: Outputs global, cout, fields, field_reduced, real field, isotropic spectrum, energy flux, shell-to-shell energy tranfsers at appropriate iterations. This operation is done at the beginning of the main loop. We do not output field_k, Tk, and pressure here. See below. Similar actions with additional `IncVF, IncSF,` etc.

11. Output field variables at specific k in the loop

    - `void Output_field_k_inloop(int iter)`: This operation requires the values of `nlin`. Therefore this function is invoked after `Compute_nlin()`. Similar actions with additional `IncVF, IncSF` etc.

12. Output pressure spectrum in the loop

    - `void Output_pressure_spectrum_inloop(int iter)`: We invoke this function after `Compute_pressure()`. The same function is used in all situations, i.e., in MHD, passive scalar etc.

Application Solvers

Chapter 12

Turbulence Simulation

After our discussion on the library functions and classes, we now come to the main program where we integrate them and use them to simulate turbulence in fluid, scalars, MHD, Rayleigh Benard convection etc. We have a solver for each of the problems, and they are invoked from the main program. The details are given below.

12.1 The main program

The main program does the following

- It reads program parameters from file `prog_para_file` (file `prog_para.d` resident in the src directory). The parameters are
  - `string prog_kind:` It could take values `INC_FLUID`, `INC_PASSIVE_SCALAR`, `INC_MHD`, `RB_SLIP`, and `INC_SLIP_MHD`.
  - `string data_dir_name`: The place where input and output files are stored.
  - `int D:` The dimensionality of the simulation.
- Depending on the `prog_kind`, the main program invokes a solver. For example, main program call `RB_slip_main(data_dir_name,D)` if the `prog_kind=RB_SLIP`.
- The main program also computes the time elapsed for the simulation using `time` library.

We declare some variables as global variables as described below.

12.2 Global variables

The global variables of the simulation are

- `fftw_plan r2c_plan_FOUR, c2r_plan_FOUR:` fftw_plan variables for FOUR basis
- `fftw_plan r2c_plan_SCFT, c2r_plan_SCFT, sintr_plan_SCFT, costr_plan_SCFT, isintr_plan_SCFT, icostr_plan_SCFT:` fftw_plan variables for SinCos-Four (SCFT) basis. Here `i` stands for the inverse.
- `Uniform<DP> SPECrand:` A class instance of `Uniform<DP>`.

We also declare several constants in `main.h`

- `const int MAXSIZE_R_SHELL = 41` (maximum no of radius vars in *sphereradius=40)
- `const int MAXSIZE_out_k_array = 51` (maximum no of wavenumbers at with the fields are outputted).

We have structured our simulation in such a way that each solver reads a standard set of parameters, e.g., size of the grid, basis type etc. All these parameters are read by a function `read_field_para(..)` that is described below

12.3 Reading field parameters

The function `void read_file_para(...)` reads the parameters from a file `field_para_file`. The parameters supplied to the function are

- `field_para_file:` The parameters are read from this file that resides in `data_dir_name/in`.
- `int D:` The dimensionality of space.
- `int number_of_fields:` The number of fields in the simulation, e.g., in MHD the number of fields is 2.

The parameters read by the function are

- **Connected to the field and integration of the field**
  - `int N[]:` The size of the array of the field variables. Read into `N[1],..N[D]`.
  - `double diss_coefficient[]:` The function reads the dissipatipation or diffusion coefficients for each of the fields.
  - `string& basis_type`
  - `string& integ_scheme:` The integrating scheme used in the solver.
  - `Array<DP,1> time_para:` The array is fed with the values of `Tinit, Tfinal, Tdt, Tdiagnostics_init` in index 1..4. These values are assigned to the class variables in the `IncFluid` constructor.

- **Connected to input of the initial condition**
  - `int& field_input_proc:` The field input procedure for reading the initial conditions.
    - 1: Read the read the complete field.
    - 2: Read the field contained in `N_in_reduced[]`.
    - 3: Read the field for a set of wavenumbers in `field_in.d` in `SIMPLE` format. As an example, for fluid simulation the inputs in this format are (k_x, k_y, V_x) in 2D and $(k_x, k_y, k_z, V_x, V_y)$ in 3D. The last component is computed by a function `IncVF::Last_component(...)`. Read Sec... for details.
    - 4: Read the field for a set of wavenumbers in `field_in.d` in `VORTICITY` format. As an example, for fluid simulation the inputs in this format are $(k_x, k_y, k_z, V_x, \Omega_x)$ in 3D. This format is not applicable in 2D. Read Sec. .. for details.
  - `int N_in_reduced[]:` The size of the reduced array of the initial field configurations.
- **Connected to the output**
  - `Array<int,1> time_save:` The array is fed with the values of `Tglobal_save, Tfield_save, Trealfield_save, Tfield_reduced_save, Tfield_k_save, Tspectrum_save, Tflux_save, Tshell_to_shell_save, Tcout_save` in index 1..9. These values are assigned to the class variables in the constructor.
  - `int N_out_reduced[]:` We output fields in smaller grid `N_out_reducedp[]`.
  - `string& nos_output_mode:` ASCII or BINARY mode for the outputs of the fields.
  - `Array<int,2> out_k_array:` Reads the wavenumbers at which the field variables are outputted.
  - `int& N_output_waveno:` The total number of wavenumbers at which the field variables are outputted.
- **Connected to the energy transfer calculations**
  - `int& nospheres:` The number of wavenumber spheres for flux caclulations.
  - `int& noshells:` The number of wavenumber shells for shell-to-shell energy transfer calculations. In out calculations we take `nospheres = noshells`.
  - `int& real_imag_switch:` 1 if contributions from the real and imag parts of the fields and nlin are to computed separately. 0 otherwise.

– `int& shell_input_scheme:` The scheme for the assignment of the shell radii. If 0, the radii are computed according to the scheme described in Sec... If 1, then the function reads the radii from the file.

– `Array<DP,1> Rshell:` If `shell_input_scheme = 1`, then function reads radii from the file for spheres 1..(`n0shells`-1). The first radius is zero, and hte last one is `INF_RADIUS`.

Chapter 13

Incompressible fluid simulation

This solver invokes the library functions and solves the equations for free slip boundary condition. The solver is a function `Ifluid_main(..)` that simulates the equations from time `Tinit` to `Tfinal`. It uses the FOUR basis. We will describe this function is some detail here.

13.1 Variables of the main program for fluid simulation

- **Connected to the field variables and integration**
  - `int* N[D+1]:` Array size of the field variables, `N[1],..,N[D]`.
  - `int no_of_fields = 1:` (the velocity field $\mathbf{V}$).
  - `DP diss_coefficients[1]:` The dissipation coefficient for $\mathbf{V}$ (ν).
  - `string basis_type:` `FOUR`.
  - `string integ_scheme:` The integrating scheme for time-stepping: `EULER, RK2, RK4`.
  - `Array<DP, 1> time_para(5): Tinit, Tfinal, Tdt, Tdiagnostics_init` in index 1..4 while reading from the file `field_para_file`.
  - `ifstream field_para_file:` File from where field parameters are read. It is in `data_dir_name/in`.
- **Connected to the energy transfer**
  - `int ETnospheres:` No of wavenumber spheres for the energy flux calculations.

- `int ETnoshells`: No of shells for the shell-to-shell energy transfer calculations. We take `ETnoshells = ETnospheres`.
- `int ET_shell_input_scheme`: 0 if shell radii are computed according to the scheme of Sec...; 1 if it is read from the `field_para_file`.
- `Array<DP, 1> ET_Rshell(MAXSIZE_R_shell)`: Contains the shell radii that are read from the file `field_para_file` if `ET_shell_input_scheme = 1`.
- `int real_imag_switch`: Switch for computing the contributions from real and imaginary parts to the energy flux and shell-to-shell transfers.

- **Connected to the input of initial conditions**
  - `int field_input_proc`: The field input procedure for reading initial condition.
  - `int* N_in_reduced[D+1]`: Reduced array size for reading the fields in a smaller grid. Index:1..D.

- **Connected to the output functions**
  - `int* N_out_reduced[D+1]`: Reduced array size for outputting the fields in a smaller grid.
  - `Array<int, 2> out_k_array(MAXSIZE_out_k_array,4)`: Contains the set wavenumbers that are read from the file `field_para_file`.
  - `int N_output_waveno`: Number of wavenumbers that are read from the file `field_para_file`.
  - `string nos_output_mode`: The output mode of the field variables: `ASCII, BINARY`.
  - `Array<int,1> time_save(10)`: `Tglobal_save, Tfield_save, Trealfield_save, Tfield_reduced_save, Tfield_k_save, Tspectrum_save, Tflux_save, Tshell_to_shell_save, Tcout_save` in index 1..9 while reading from the file `field_para_file`.

13.2 Main program for fluid simulation

The steps in the solver are

- Read_field_para(): Reads parameters of the field from the file `field_para_file` which resides in `data_dir_name/in` directory. It reads `N[], diss_coefficients[]`, etc. See Sec. src_main for details.
- Constructor IncFluid U for the velocity field.
- Create fftw_plans.

- Open input files
- Initialize the field configurations using `U.Init_cond(...)`. The choice of the function depends on `field_input_proc`.
- Close input files.
- Open output files.
- Start the Computation:
- For (`U.Tnow` = `U.Tinit`, `U.Tnow <= U.Tfinal+0.0000001; U.Tnow +=` `U.Tdt`) {
  - `U.Compute_force();`
  - `U.Compute_nlin();`
  - `U.Add_force();`
  - `U.Compute_pressure();`
  - `U.Time_advance();`
- Close output files.
- Output results inside the loop.

13.3 Basic tests of the solver

13.3.1 Test the conservation of energy when dissipation and forcing are turned off

- Initial conditions
  - 2D: $\mathbf{V}(2,1) = (2+3i, -4-6i)$, $\mathbf{V}(1,1) = (5+5i, -5-5i)$, $\mathbf{V}(3,2) = (6+6i, -9-9i)$.
  - 3D: $\mathbf{V}(2,2,1) = (2+3i, 2+3i, -8-12i)$, $\mathbf{V}(1,1,1) = (5+5i, 5+5i, -10-10i)$, $\mathbf{V}(3,3,2) = (6+6i, 6+6i, -18-18i)$
- Parameter values
  - 2D: $N = 32 \times 32$, $\nu = 0.0$, $dt = 10^{-3}$, `basis_type = FOUR`
  - 3D: $N = 32 \times 32 \times 32$, $\nu = 0.0$, $dt = 10^{-3}$, `basis_type = FOUR`
- Results:
- Note: If we keep only one mode with $\nu = 0$, the field configuration does not change, and the energy is conserved.

Table 13.1: Simulation results for 2D when $\nu = 0$

	Euler	RK2	RK4
$E^u(t=0)$	399	399	399
$E^u(t=0.1)$	399.508	399	399
$V_x(1,1)(t=0.1)$	$2.90075 + 7.62898i$	$2.89982 + 7.62239i$	$2.89982 + 7.62238i$
$V_x(3,2)(t=0.1)$	$4.16542 + 3.80809i$	$4.166 + 3.8077i$	$4.166 + 3.8077i$
$E^u(t=0.2)$	405.365	402.855	402.854
$V_x(1,1)(t=0.2)$	$-2.48577 + 0.631538i$	$-2.49237 + 0.608368i$	$-2.49237 + 0.608345i$
$V_x(3,2)(t=0.2)$	$1.54597 + 1.63347i$	$1.54203 + 1.63646i$	$1.54202 + 1.63646i$
Time reqd ($t = 0.2$)	0.994 sec	2.02 sec	4.25 sec
nan at	$t = 0.238$	$t = 0.237$	$t = 0.237$

Table 13.2: Simulation results for 3D when $\nu = 0$.

	Euler	RK2	RK4
$E^u(t=0)$	1326	1326	1326
$E^u(t=0.05)$	1329.69	1326.06	1326.06
$V_x(1,1,1)(t=0.05)$	$3.01192 + 7.88901i$	$3.01111 + 7.8743i$	$3.01113 + 7.8743i$
$V_x(2,2,1)(t=0.05)$	$0.627639 - 3.46226i$	$0.627463 - 3.45859i$	$0.627462 - 3.45859$
$E^u(t=0.065)$	1405.19	2053.14	1718.55
$V_x(1,1,1)(t=0.065)$	$1.55511 + 6.67214i$	$1.43555 + 6.52277i$	$1.50197 + 6.59544i$
$V_x(2,2,1)(t=0.065)$	$0.0371095 - 4.42679i$	$0.131824 - 4.445i$	$0.0786792 - 4.43102i$
Time reqd ($t = 0.2$)	22.57 sec	45.26 sec	94.73 sec
nan at	$t = 0.0715$	$t = 0.0685$	$t = 0.0685$

Table 13.3: Simulation results for dissipative fluid simulation in 2D

	Euler	RK2	RK4
$E^u(t=0)$	65	65	65
$E^u(t=1)$	0.002951	same	same
$V_x(2,1)(t=1)$	$0.0134759 + 0.0202138i$	same	same

Table 13.4: Simulation results for dissipative fluid simulation in 3D

	Euler	RK2	RK4
$E^u(t=0)$	234	234	234
$E^u(t=0.05)$	95.1373	same	same
$D^u(t=0.05)$	1727.47	same	same
$V_x(2,2,1)(t=0.05)$	1.27526 1.91288	same	same

13.3.2 Test pure dissipation by taking a single mode thus turning off nonlinearity

- Initial conditions
  - 2D: $\mathbf{V}(2,1) = (2+3i, -4-6i)$.
  - 3D: $\mathbf{V}(2,2,1) = (2+3i, 2+3i, -8-12i)$
- Parameter values
  - 2D: $N = 32 \times 32$, $\nu = 1.0$, $dt = 10^{-3}$, `basis_type = FOUR`
  - 3D: $N = 32 \times 32 \times 32$, $\nu = 0.0$, $dt = 10^{-3}$, `basis_type = FOUR`
- Results:

$$\begin{aligned} \mathbf{V}(\mathbf{k},t) &= \exp(-\nu K^2 t)\mathbf{V}(\mathbf{k},0), \\ E^u(t) &= \exp(-2\nu K^2 t)E^u(0), \\ D^u(t) &= 2K^2 \exp(-2\nu K^2 t)E^u(0). \end{aligned}$$

Chapter 14

Simulation of passive scalar

This solver invokes the library functions and solves the equations for free slip boundary condition. The solver is a function `RB_slip_main(..)` that simulates the equations from time `Tinit` to `Tfinal`. It uses the SinCosFour (SCFT) basis. We will describe this function is some detail here.

14.1 Variables of the main program for passive scalar

- **Connected to the field variables and integration**
  - `int* N[D+1]:` Array size of the field variables, `N[1],..,N[D]`.
  - `int no_of_fields = 2:` (the velocity field $\mathbf{V}$ and the temperature field ζ).
  - `DP diss_coefficients[2]:` The dissipation coefficient for $\mathbf{V}$ (ν), and the diffusion coefficient for the scalar ζ (κ).
  - `string basis_type: FOUR`
  - `string integ_scheme:` The integrating scheme for time-stepping: `EULER, RK2, RK4`.
  - `Array<DP, 1> time_para(5): Tinit, Tfinal, Tdt, Tdiagnostics_init` in index 1..4 while reading from the file `field_para_file`.
  - `ifstream field_para_file:` File from where field parameters are read. It is in `data_dir_name/in`.
- **Connected to the energy transfer**
  - `int ETnospheres:` No of wavenumber spheres for the energy flux calculations
  - `int ETnoshells:` No of shells for the shell-to-shell energy transfer calculations

- `int ET_shell_input_scheme:` 0 if shell radii are computed according to the scheme of Sec...; 1 if it is read from the `field_para_file`.
- `Array<DP, 1> ET_Rshell(MAXSIZE_R_shell):` Contains the shell radii that are read from the file `field_para_file` if `ET_shell_input_scheme = 1`.
- `int real_imag_switch:` Switch for computing the contributions from real and imaginary parts to the energy flux and shell-to-shell transfers.

- **Connected to the input of initial conditions**
  - `int field_input_proc:` The field input procedure for reading initial condition.
  - `int* N_in_reduced[D+1]:` Reduced array size for reading the fields in a smaller grid. Index:1..D.

- **Connected to the output functions**
  - `int* N_out_reduced[D+1]:` Reduced array size for outputting the fields in a smaller grid.
  - `Array<int, 2> out_k_array(MAXSIZE_out_k_array,4):` Contains the set wavenumbers that are read from the file `field_para_file`.
  - `int N_output_waveno:` Number of wavenumbers that are read from the file `field_para_file`.
  - `string nos_output_mode:` The output mode of the field variables: `ASCII, BINARY`.
  - `Array<int,1> time_save(10): Tglobal_save, Tfield_save, Trealfield_save, Tfield_reduced_save, Tfield_k_save, Tspectrum_save, Tflux_save, Tshell_to_shell_save, Tcout_save` in index 1..9 while reading from the file `field_para_file`.

14.2 Main program for passive scalar simulation

The steps in the solver are

- Read_field_para(): Reads parameters of the field from the file `field_para_file` which resides in `data_dir_name/in` directory. It reads `N[], diss_coefficients[]`, etc. See Sec. src_main for details.
- Constructor IncFluid U for the velocity field.
- Constructor IncSF T for the temperature field ζ.
- Create fftw_plans.
- Open input files

- Initialize the field configurations using `U.Init_cond(...)`. The choice of the function depends on `field_input_proc`.
- Close input files.
- Open output files.
- Start the Computation:
- For (`U.Tnow` = `U.Tinit`, `U.Tnow <= U.Tfinal+0.0000001; U.Tnow +=` `U.Tdt`) {
  - `U.Compute_force(T);`
  - `U.Compute_nlin(T);`
  - `U.Add_force(T);`
  - `U.Compute_pressure();`
  - `U.Time_advance(T);`
- Close output files.
- Output results inside the loop.

14.3 Basic tests of the solver

14.3.1 Test the conservation of energy when dissipation and forcing are turned off

- Initial conditions
  - 2D: $\mathbf{V}(2,1) = (2+3i, -4-6i)$, $\mathbf{V}(1,1) = (5+5i, -5-5i)$, $\mathbf{V}(3,2) = (6+6i, -9-9i)$; $\zeta(2,1) = 2+3i$, $\zeta(1,1) = 5+5i$, $\zeta(3,2) = 6+6i$.
  - 3D: $\mathbf{V}(2,2,1) = (2+3i, 2+3i, -8-12i)$, $\mathbf{V}(1,1,1) = (5+5i, 5+5i, -10-10i)$, $\mathbf{V}(3,3,2) = (6+6i, 6+6i, -18-18i)$, $\zeta(2,2,1) = 2+3i$, $\zeta(1,1,1) = 5+5i$, $\zeta(3,3,2) = 6+6i$.
- Parameter values
  - 2D: $N = 32 \times 32$, $\nu = 0.0$, $dt = 10^{-3}$, `basis_type = FOUR`
  - 3D: $N = 32 \times 32 \times 32$, $\nu = 0.0$, $dt = 10^{-3}$, `basis_type = FOUR`
- Results:
- Note:
  - If we keep only one mode with $\nu = 0$, the field configuration does not change, and the energy is conserved.
  - The evolution of the velocity field is the same as that in fluid flow because the scalar does not react back on the velocity field.

Table 14.1: Simulation results for 2D when $\nu = 0$

	Euler	RK2	RK4
$E^u(t=0)$	399	399	399
$E^u(t=0)$	135	135	135
$E^u(t=0.1)$	399.508	399	399
$E^u(t=0.1)$	135.272	135.096	135.096
$V_x(1,1)(t=0.1)$	$2.90075 + 7.62898i$	$2.89982 + 7.62239i$	$2.89982 + 7.62238i$
$V_x(3,2)(t=0.1)$	$4.16542 + 3.80809i$	$4.166 + 3.8077i$	$4.166 + 3.8077i$
$\zeta(1,1)(t=0.1)$	$2.93991 + 6.52197i$	$2.93944 + 6.51603i$	$2.93944 + 6.51603i$
$\zeta(3,2)(t=0.1)$	$3.57215 + 3.29405i$	$3.57379 + 3.29496i$	$3.5738 + 3.29497i$
$E^u(t=0.2)$	405.365	402.855	402.854
$E^u(t=0.2)$	116125	119722	119694
$V_x(1,1)(t=0.2)$	$-2.48577 + 0.631538i$	$-2.49237 + 0.608368i$	$-2.49237 + 0.608345i$
$V_x(3,2)(t=0.2)$	$1.54597 + 1.63347i$	$1.54203 + 1.63646i$	$1.54202 + 1.63646i$
$\zeta(1,1)(t=0.2)$	$-0.402805 + 1.14681i$	$-0.424682 + 1.18352i$	$-0.424737 + 1.18339i$
$\zeta(3,2)(t=0.2)$	$0.264168 - 3.12637i$	$0.456391 - 3.13251i$	$0.456678 - 3.13175i$
Time reqd $(t=0.2)$	1.60 sec	3.12 sec	6.45 sec
nan at	$t = 0.238$	$t = 0.237$	$t = 0.237$

Table 14.2: Simulation results for 3D when $\nu = 0$.

	Euler	RK2	RK4
$E^u(t=0)$	1326	1326	1326
$E^\zeta(t=0)$	135	135	135
$E^u(t=0.05)$	1329.69	1326.06	1326.06
$E^\zeta(t=0.05)$	135.723	135.399	135.399
$V_x(1,1,1)(t=0.05)$	$3.01192 + 7.88901i$	$3.01111 + 7.8743i$	$3.01113 + 7.8743i$
$V_x(2,2,1)(t=0.05)$	$0.627639 - 3.46226i$	$0.627463 - 3.45859i$	$0.627462 - 3.45859i$
$\zeta(1,1,1)(t=0.05)$	$3.09279 + 6.52184i$	$3.09234 + 6.50955i$	$3.09235 + 6.50954i$
$\zeta(2,2,1)(t=0.05)$	$0.710119 - 5.27703i$	$0.710582 - 5.27223i$	$0.710583 - 5.27224i$
$E^u(t=0.065)$	1405.19	2053.14	1718.55
$E^\zeta(t=0.065)$	599.639	1138.5	880.435
$V_x(1,1,1)(t=0.065)$	$1.55511 + 6.67214i$	$1.43555 + 6.52277i$	$1.50197 + 6.59544i$
$V_x(2,2,1)(t=0.065)$	$0.0371095 - 4.42679i$	$0.131824 - 4.445i$	$0.0786792 - 4.43102i$
$\zeta(1,1,1)(t=0.065)$	$1.78776 + 4.97492i$	$1.6421 + 4.80179i$	$1.72364 + 4.88901i$
$\zeta(2,2,1)(t=0.065)$	$0.133171 - 6.21204i$	$0.402219 - 6.32723i$	$0.250709 - 6.2584i$
Time reqd $(t=0.065)$	30.74 sec	85.26 sec	125.1 sec
nan at	$t = 0.0715$	$t = 0.0685$	$t = 0.0685$

Table 14.3: Simulation results for dissipative fluid simulation in 2D

	Euler	RK2	RK4
$E^u(t=0)$	65	65	65
$E^\zeta(t=0)$	13	13	13
$E^u(t=1)$	0.002951	same	same
$E^\zeta(t=1)$	5.90199×10^{-4}	same	same
$D^u(t=1)$	0.02951	same	same
$D^\zeta(t=1)$	0.00590199	same	same
$V_x(2,1)(t=1)$	$0.0134759 + 0.0202138i$	same	same
$\zeta(2,1)(t=1)$	$0.0134759 + 0.0202138i$	same	same

14.3.2 Test pure dissipation by taking a single mode thus turning off nonlinearity

- Initial conditions
  - 2D: $\mathbf{V}(2,1) = (2+3i, -4-6i)$, $\zeta(2,1) = 2+3i$.
  - 3D: $\mathbf{V}(2,2,1) = (2+3i, 2+3i, -8-12i)$, $\zeta(2,2,1) = 2+3i$.
- Parameter values
  - 2D: $N = 32 \times 32$, $\nu = \kappa = 1.0$, $dt = 10^{-3}$, `basis_type = FOUR`
  - 3D: $N = 32 \times 32 \times 32$, $\nu = \kappa = 0.0$, $dt = 10^{-3}$, `basis_type = FOUR`
- Results:

$$
\begin{aligned}
\mathbf{V}(\mathbf{k},t) &= \exp(-\nu K^2 t)\mathbf{V}(\mathbf{k},0), \\
\zeta(\mathbf{k},t) &= \exp(-\kappa K^2 t)\zeta(\mathbf{k},0), \\
E^u(t) &= \exp(-2\nu K^2 t)E^u(0), \\
E^\zeta(t) &= \exp(-2\kappa K^2 t)E^\zeta(0), \\
D^u(t) &= 2K^2 \exp(-2\nu K^2 t)E^u(0) \\
D^\zeta(t) &= 2K^2 \exp(-2\kappa K^2 t)E^\zeta(0)
\end{aligned}
$$

Table 14.4: Simulation results for dissipative fluid simulation in 3D

	Euler	RK2	RK4
$E^u(t=0)$	234	234	234
$E^\zeta(t=0)$	13	13	13
$E^u(t=0.05)$	95.1373	same	same
$E^\zeta(t=0.05)$	5.28541	same	same
$D^u(t=0.05)$	1727.47	same	same
$D^\zeta(t=0.05)$	95.1373	same	same
$V_x(2,2,1)(t=0.05)$	$1.27526 + 1.91288i$	same	same
$\zeta(2,2,1)(t=0.05)$	$1.27526 + 1.91288i$	same	same

Chapter 15

Magnetohydrodynamic flows

This solver invokes the library functions and solves the equations for magneto-hydrodynamic flows. The solver is a function `IMHD_main(..)` that simulates the equations from time `Tinit` to `Tfinal`. It uses the FOUR basis. We will describe this function is some detail here.

15.1 Variables of the main program for MHD flow

- **Connected to the field variables and integration**
  - `int* N[D+1]:` Array size of the field variables, `N[1],..,N[D]`.
  - `int no_of_fields = 2:` (the velocity field $\mathbf{V}$ and the magnetic field $\mathbf{W}$).
  - `DP diss_coefficients[2]:` The dissipation coefficient for $\mathbf{V}$ (ν), and the dissipation coefficient for $\mathbf{W}$ (η).
  - `string basis_type: FOUR`
  - `string integ_scheme:` The integrating scheme for time-stepping: `EULER, RK2, RK4`.
  - `Array<DP, 1> time_para(5): Tinit, Tfinal, Tdt, Tdiagnostics_init` in index 1..4 while reading from the file `field_para_file`.
  - `ifstream field_para_file:` File from where field parameters are read. It is in `data_dir_name/in`.
- **Connected to the energy transfer**
  - `int ETnospheres:` No of wavenumber spheres for the energy flux calculations
  - `int ETnoshells:` No of shells for the shell-to-shell energy transfer calculations

- `int ET_shell_input_scheme:` 0 if shell radii are computed according to the scheme of Sec...; 1 if it is read from the `field_para_file`.
- `Array<DP, 1> ET_Rshell(MAXSIZE_R_shell):` Contains the shell radii that are read from the file `field_para_file` if `ET_shell_input_scheme = 1`.
- `int real_imag_switch:` Switch for computing the contributions from real and imaginary parts to the energy flux and shell-to-shell transfers.

- **Connected to the input of initial conditions**
  - `int field_input_proc:` The field input procedure for reading initial condition.
  - `int* N_in_reduced[D+1]:` Reduced array size for reading the fields in a smaller grid. Index:1..D.

- **Connected to the output functions**
  - `int* N_out_reduced[D+1]:` Reduced array size for outputting the fields in a smaller grid.
  - `Array<int, 2> out_k_array(MAXSIZE_out_k_array,4):` Contains the set wavenumbers that are read from the file `field_para_file`.
  - `int N_output_waveno:` Number of wavenumbers that are read from the file `field_para_file`.
  - `string nos_output_mode:` The output mode of the field variables: `ASCII, BINARY`.
  - `Array<int,1> time_save(10): Tglobal_save, Tfield_save, Trealfield_save, Tfield_reduced_save, Tfield_k_save, Tspectrum_save, Tflux_save, Tshell_to_shell_save, Tcout_save` in index 1..9 while reading from the file `field_para_file`.

15.2 Main program for RB convection

The steps in the solver are

- Read_field_para(): Reads parameters of the field from the file `field_para_file` which resides in `data_dir_name/in` directory. It reads `N[], diss_coefficients[]`, etc. See Sec. src_main for details.
- Constructor IncFluid `U` for the velocity field.
- Constructor IncVF `W` for the magnetic field **W**.
- Create fftw_plans.
- Open input files

- Initialize the field configurations using `U.Init_cond(...)`. The choice of the function depends on `field_input_proc`.
- Close input files.
- Open output files.
- Start the Computation:
- For (`U.Tnow` = `U.Tinit`, `U.Tnow <= U.Tfinal+0.0000001; U.Tnow +=` `U.Tdt`) {
  - `U.Compute_force(W);`
  - `U.Compute_nlin(W);`
  - `U.Add_force(W);`
  - `U.Compute_pressure();`
  - `U.Time_advance(W);`
- Close output files.
- Output results inside the loop.

15.3 Basic tests of the solver

15.3.1 Test the conservation of energy when dissipation and forcing are turned off

15.3.1.1 $\mathbf{B} = 0$

We recover the fluid limit. We find the exact solution described in fluid simulation chapter.

15.3.1.2 $\mathbf{U} = \mathbf{B}$

In this case the nonlinear terms and the pressure gradient is zero. Hence it is a linear equation. The energy is identically conserved.

15.3.1.3 General situation

- Initial conditions
  - 2D: $\mathbf{V}(2,1) = (2+3i, -4-6i)$, $\mathbf{V}(1,1) = (5+5i, -5-5i)$, $\mathbf{V}(3,2) = (6+6i, -9-9i)$; $\mathbf{B}(2,1) = (1+2i, -2-4i)$, $\mathbf{V}(1,1) = (2+3i, -2-3i)$, $\mathbf{V}(3,2) = (2+3i, -3-4.5i)$.
  - 3D: $\mathbf{V}(2,2,1) = (2+3i, 2+3i, -8-12i)$, $\mathbf{V}(1,1,1) = (5+5i, 5+5i, -10-10i)$, $\mathbf{V}(3,3,2) = (6+6i, 6+6i, -18-18i)$, $\mathbf{B}(2,2,1) = (1+2i, 1+2i, -4-8i)$, $\mathbf{B}(1,1,1) = (1+3i, 1+2i, -2-5i)$, $\mathbf{B}(3,3,2) = (3+6i, 6+3i, -13.5-13.5i)$.

Table 15.1: Simulation results for 2D when $\nu = 0$

	Euler	RK2	RK4
$E^u(t=0)$	399	399	399
$E^b(t=0)$	93.25	93.25	93.25
$H_c(t=0)$	187.5	187.5	187.5
$E^u(t=0.05)$	372.911	372.774	372.774
$E^b(t=0.05)$	119.476	119.476	119.476
$E^u(t=0.05)$	187.501	187.5	187.5
$V_x(1,1)(t=0.05)$	$4.85234 + 6.33906i$	$4.8514 + 6.33695i$	$4.8514 + 6.33695i$
$V_x(3,2)(t=0.05)$	$5.89601 + 5.58382i$	$5.8953 + 5.58323i$	$5.8953 + 5.58323i$
$B_x(1,1)(t=0.05)$	$1.74497 + 2.64333i$	$1.74509 + 2.64315i$	$1.74509 + 2.64315i$
$B_x(3,2)(t=0.05)$	$2.89188 + 3.54185i$	$2.89228 + 3.54222i$	$2.89228 + 3.54221i$
$E^u(t=0.1)$	317.695	317.643	317.643
$E^b(t=0.1)$	181.166	181.248	181.248
$H_c(t=0.1)$	189.055	189.075	189.075
$V_x(1,1)(t=0.1)$	$3.65843 + 5.99974i$	$3.65806 + 5.9959i$	$3.65806 + 5.99596i$
$V_x(3,2)(t=0.1)$	$4.86681 + 4.4334i$	$4.86699 + 4.43393i$	$4.86699 + 4.43393i$
$B_x(1,1)(t=0.1)$	$1.43317 + 2.01415i$	$1.43391 + 2.01482i$	$1.43391 + 2.01482i$
$B_x(3,2)(t=0.1)$	$3.94149 + 4.26307i$	$3.93984 + 4.26178i$	$3.93984 + 4.26178i$
Time reqd $(t=0.1)$	1.07 sec	1.97 sec	4.00 sec
nan at	$t = 0.118$	$t = 0.1165$	$t = 0.237$

- Parameter values
  - 2D: $N = 32 \times 32$, $\nu = \eta = 0.0$, $dt = 10^{-3}$, `basis_type = FOUR`
  - 3D: $N = 32 \times 32 \times 32$, $\nu = \eta = 0.0$, $dt = 10^{-3}$, `basis_type = FOUR`
- Results:
- Note:
  - If we keep only one mode with $\nu = 0$, the field configuration does not change, and the energy is conserved.

15.3.2 Test pure dissipation by taking a single mode thus turning off nonlinearity

- Initial conditions
  - 2D: $\mathbf{V}(2,1) = (2+3i, -4-6i)$, $\mathbf{B}(2,1) = (2+3i, -4-6i)$.
  - 3D: $\mathbf{V}(2,2,1) = (2+3i, 2+3i, -8-12i)$, $\mathbf{B}(2,2,1) = (2+3i, 2+3i, -8-12i)$.
- Parameter values

Table 15.2: Simulation results for 3D when $\nu = 0$.

	Euler	RK2	RK4
$E^u(t=0)$	1326	1326	1326
$E^b(t=0)$	588.5	588.5	588.5
$H_c(t=0)$	843	843	843
$E^u(t=0.02)$	1271.65	1270.95	1270.95
$E^b(t=0.02)$	643.656	643.549	643.549
$H_c(t=0.02)$	843.241	843	843
$V_x(1,1,1)(t=0.02)$	$4.75337 + 5.88545i$	$4.75166 + 5.88285i$	$4.75166 + 5.88285i$
$V_x(2,2,1)(t=0.02)$	$2.05655 + 1.14229i$	$2.05632 + 1.14265i$	$2.05632 + 1.14265i$
$B_x(1,1,1)(t=0.02)$	$0.583614 + 3.00627i$	$0.583836 + 3.00562i$	$0.583838 + 3.00562i$
$B_x(2,2,1)(t=0.02)$	$1.58379 + 1.47647i$	$1.58298 + 1.47639i$	$1.58298 + 1.47639i$
$E^u(t=0.04)$	1366.53	1439.99	1440.67
$E^b(t=0.04)$	945.442	1019.05	1019.73
$H_c(t=0.04)$	653.833	580.087	579.41
$V_x(1,1,1)(t=0.04)$	$3.76422 + 5.92274i$	$3.76385 + 5.91896i$	$3.76384 + 5.91897i$
$V_x(2,2,1)(t=0.04)$	$2.02816 - 0.57983i$	$2.02985 - 0.578084i$	$2.02986 - 0.578094i$
$B_x(1,1,1)(t=0.04)$	$0.162143 + 2.78284i$	$0.164271 + 2.7815i$	$0.16428 + 2.78149i$
$B_x(2,2,1)(t=0.04)$	$1.82165 + 0.918483i$	$1.81889 + 0.918598i$	$1.81888 + 0.918609i$
Time reqd $(t=0.04)$	31 sec	57.8 sec	116.3 sec
nan at	$t = 0.049$	$t = 0.0475$	$t = 0.047$

  - 2D: $N = 32 \times 32$, $\nu = \eta = 1.0$, $dt = 10^{-3}$, `basis_type = FOUR`
  - 3D: $N = 32 \times 32 \times 32$, $\nu = \eta = 0.0$, $dt = 10^{-3}$, `basis_type = FOUR`

- Results:

$$
\begin{aligned}
\mathbf{V}(\mathbf{k},t) &= \exp(-\nu K^2 t)\mathbf{V}(\mathbf{k},0), \\
\mathbf{B}(\mathbf{k},t) &= \exp(-\eta K^2 t)\mathbf{B}(\mathbf{k},0), \\
E^u(t) &= \exp(-2\nu K^2 t)E^u(0), \\
E^b(t) &= \exp(-2\eta K^2 t)E^b(0), \\
D^u(t) &= 2K^2 \exp(-2\nu K^2 t)E^u(0) \\
D^b(t) &= 2K^2 \exp(-2\eta K^2 t)E^b(0)
\end{aligned}
$$

Table 15.3: Simulation results for dissipative fluid simulation in 2D

	Euler	RK2	RK4
$E^u(t=0)$	65	65	65
$E^b(t=0)$	65	65	65
$E^u(t=1)$	0.002951	same	same
$E^b(t=1)$	0.002951	same	same
$D^u(t=1)$	0.02951	same	same
$D^b(t=1)$	0.02951	same	same
$V_x(2,1)(t=1)$	$0.0134759 + 0.0202138i$	same	same
$B_x(2,1)(t=1)$	$0.0134759 + 0.0202138i$	same	same

Table 15.4: Simulation results for dissipative fluid simulation in 3D

	Euler	RK2	RK4
$E^u(t=0)$	234	234	234
$E^b(t=0)$	234	234	234
$E^u(t=0.05)$	95.1373	same	same
$E^b(t=0.05)$	95.1373	same	same
$D^u(t=0.05)$	1727.47	same	same
$D^b(t=0.05)$	1727.47	same	same
$V_x(2,2,1)(t=0.05)$	$1.27526 + 1.91288i$	same	same
$B_x(2,2,1)(t=0.05)$	$1.27526 + 1.91288i$	same	same

Chapter 16

Raylegh Benard convection for free slip boundary condition

This solver invokes the library functions and solves the equations for free slip boundary condition. The solver is a function `RB_slip_main(..)` that simulates the equations from time `Tinit` to `Tfinal`. It uses the SinCosFour (SCFT) basis. We will describe this function is some detail here.

16.1 Variables of the main program for RB convection

- **Connected to the field variables and integration**
  - `int* N[D+1]:` Array size of the field variables, `N[1],..,N[D]`.
  - `int no_of_fields = 2:` (the velocity field $\mathbf{V}$ and the temperature field ζ).
  - `DP diss_coefficients[2]:` The dissipation coefficient for $\mathbf{V}$ (ν), and the diffusion coefficient for the scalar ζ (κ).
  - `string basis_type:` basis type either `FOUR` or `SCFT`
  - `string integ_scheme:` The integrating scheme for time-stepping: `EULER, RK2, RK4`.
  - `Array<DP, 1> time_para(5):` `Tinit, Tfinal, Tdt, Tdiagnostics_init` in index 1..4 while reading from the file `field_para_file`.
  - `ifstream field_para_file:` File from where field parameters are read. It is in `data_dir_name/in`.
- **Connected to the energy transfer**

- `int ETnospheres:` No of wavenumber spheres for the energy flux calculations
- `int ETnoshells:` No of shells for the shell-to-shell energy transfer calculations
- `int ET_shell_input_scheme:` 0 if shell radii are computed according to the scheme of Sec...; 1 if it is read from the `field_para_file`.
- `Array<DP, 1> ET_Rshell(MAXSIZE_R_shell):` Contains the shell radii that are read from the file `field_para_file` if `ET_shell_input_scheme = 1`.
- `int real_imag_switch:` Switch for computing the contributions from real and imaginary parts to the energy flux and shell-to-shell transfers.

- **Connected to the input of initial conditions**
  - `int field_input_proc:` The field input procedure for reading initial condition.
  - `int* N_in_reduced[D+1]:` Reduced array size for reading the fields in a smaller grid. Index:1..D.

- **Connected to the output functions**
  - `int* N_out_reduced[D+1]:` Reduced array size for outputting the fields in a smaller grid.
  - `Array<int, 2> out_k_array(MAXSIZE_out_k_array,4):` Contains the set wavenumbers that are read from the file `field_para_file`.
  - `int N_output_waveno:` Number of wavenumbers that are read from the file `field_para_file`.
  - `string nos_output_mode:` The output mode of the field variables: `ASCII, BINARY`.
  - `Array<int,1> time_save(10): Tglobal_save, Tfield_save, Trealfield_save, Tfield_reduced_save, Tfield_k_save, Tspectrum_save, Tflux_save, Tshell_to_shell_save, Tcout_save` in index 1..9 while reading from the file `field_para_file`.

- **Parameters connected to the RB convection**
  - `ifstream RB_slip_para_file:` The file from which the parameters of RB convection are read. It resides in `data_dir_name/in`.
  - `DP Pr:` Thermal Prandtl number
  - `DP Ra:` Rayleigh number
  - `DP r:` $r = Ra/Ra_c$
  - `DP k0:` `kfactor[2];` $k0 = \pi/\sqrt{2}$

- DP q: kfactor[3].
- DP qbyk0: $q/k0$.
- string Pr_switch: Takes on the values PRLARGE, PRSMALL, PRZERO.
- RB_Uscaling: Takes one of the values USMALL, ULARGE.
- DP w101, th101, th200: Lorenz variables.

16.2 Main program for RB convection

The steps in the solver are

- Read_field_para(): Reads parameters of the field from the file field_para_file which resides in data_dir_name/in directory. It reads N[], diss_coefficients[], etc. See Sec. src_main for details.
- Read_RB_para(): Reads parameters for RB from RB_para_file that is resident in data_dir_name/in. The variables read are Pr, r, qbyk0, Pr_scaling, RB_Uscaling. If field_input_proc = 0, then Lorenz variables w101, th101, th200 are also read..
- Set up $Ra = r \times Ra_c$ and $kfactor[i]$.
- Set up the coefficients of $\nabla^2\mathbf{u}$ and $\nabla^2\theta$ (see Appendix and Sec.).
- Constructor IncFluid U for the velocity field.
- Constructor IncSF T for the temperature field ζ.
- Create fftw_plans.
- Open input files
- Initialize the field configurations using U.Init_cond(...). The choice of the function depends on field_input_proc.
- Close input files.
- Open output files.
- Start the Computation:
- For (U.Tnow = U.Tinit, U.Tnow <= U.Tfinal+0.0000001; U.Tnow += U.Tdt) {
  - U.Compute_force(T, Ra, Pr, Pr_switch, RB_Uscaling);
  - U.Compute_nlin(T, Pr_switch);
  - U.Add_force(T, Pr_switch);
  - U.Compute_pressure();

Table 16.1: Simulation results for 2D at $t = 1$.

	RK4-Init-1	RK4-Init2
$E^u(t=0)$	0.06	1634
$E^\zeta(t=0)$	0.135	326
E^u	673.011	296.424
E^ζ	0.0371593	0.0291604
Nu	4.24353	3.71141
$V_x(1,1)$	-10.4612	$0.669826 - 0.665225i$
$V_y(2,0)$	0	3.66145e-06
$V_x(2,1)$	0	$-0.041591 + 0.0473371i$
$V_x(3,2)$	0	$-0.012952 - 0.214353i$
$\zeta(1,1)$	-0.0706091	$0.00424838 - 0.00422111i$
$\zeta(2,0)$	-0.147603	-0.13185
$\zeta(2,1)$	0	$-0.00185042 + 0.00210602i$
$\zeta(3,2)$	0	$-0.00140017 - 0.0231737i$
time reqd	68.22 sec	744 sec

  - `U.Time_advance(T, Ra, Pr, Pr_switch, RB_Uscaling);`

- Close output files.
- Output results inside the loop.

16.3 Basic tests of the solver

16.3.1 Two dimensional simulation ($Pr = 6.8$)

We perform test for $r = Ra/Ra_c = 10$ and $Pr = 6.8$.

- Initial conditions
  - Init-1: Lorenz condition: $w_{11} = 0.1$, $\theta_{11} = 0.15$, $\theta_{20} = 0.3$.
  - Init-2: $\mathbf{V}(2,1) = (2{+}3i, -4{-}6i)$, $\mathbf{V}(1,1) = (5{+}5i, -5{-}5i)$, $\mathbf{V}(3,2) = (6+8i, -9-12i)$, $\zeta(2,1) = 2+3i$, $\zeta(1,1) = 5+5i$, $\zeta(3,2) = 6+8i$.
- Parameter values
  - $N = 32 \times 32$, $r = 10$, $Pr = 6.8$, $dt = 10^{-4}$, `basis_type = SCFT`
- Results:

Table 16.2: Simulation results for 3D. We obtain a steady state. We report the final state for Init-1 at $t = 0.65$, and for Init-2 at $t = 0.5$. In the final state the energy is fluctuating a bit, but the values of the Fourier modes are constant.

	RK4-Init-1	RK4-Init2
$E^u(t=0)$	0.06	1634
$E^\zeta(t=0)$	0.135	326
E^u	673.01	673.215
E^ζ	0.0371589	0.0370734
Nu	4.24358	4.24347
$V_x(1,0,1)$	-10.4612	$0.660366 - 0.758998i$
$V_y(2,0,0)$	0	-4.25617×10^{-16}
$V_x(2,1,1)$	0	$0.0877739 + 0.0944594i$
$V_x(1,1,2)$	-	$-0.0537467 + 0.0017584i$
$V_x(3,2,3)$	-	$(-1.18153 - 1.19117i) \times 10^{-5}$
$\zeta(1,0,1)$	-0.0706107	$0.00459038 - 0.00527599i$
$\zeta(2,0,0)$	-0.147603	-0.147206
$\zeta(2,1,1)$	0	$0.000181435 + 0.00019527i$
$\zeta(1,1,2)$	0	$-0.000437023 + 1.42974 \times 10^{-5}i$
$\zeta(3,2,3)$	-	$(-3.13714 - 3.16216i) \times 10^{-6}$
time reqd	364 min	276 min

16.3.2 Three dimensional simulation ($Pr = 6.8$)

- Initial conditions
  - Init-1: Lorenz like condition: $\mathbf{V}(1,0,1) = (0.42, 0.1, -0.42\sqrt{2}i)$, $\zeta(1,0,1) = 0.07$, $\zeta(2,0,0) = 0.7$.
  - Init-2: $\mathbf{V}(2,1,1) = (2+3i, 2+3i, -8-12i)$, $\mathbf{V}(1,1,2) = (5+5i, 5+5i, -10-10i)$, $\mathbf{V}(3,2,3) = (6+6i, 6+6i, -18-18i)$, $\zeta(2,1,1) = 2+3i$, $\zeta(1,1,2) = 5+5i$, $\zeta(3,2,3) = 6+6i$.
- Parameter values
  - 3D: $N = 32 \times 32 \times 32$, $Pr = 6.8$, $r = 10$, $dt = 10^{-5}$, `basis_type = SCFT`
- Results:

Chapter 17

Magnetoconvection for free slip boundary condition

This solver invokes the library functions and solves the equations for free slip boundary condition. The solver is a function `RB_slip_main(..)` that simulates the equations from time `Tinit` to `Tfinal`. It uses the SinCosFour (SCFT) basis. We will describe this function is some detail here.

17.1 Variables of the main program for RB convection

- **Connected to the field variables and integration**
  - `int* N[D+1]:` Array size of the field variables, `N[1],..,N[D]`.
  - `int no_of_fields = 3:` (the velocity field $\mathbf{V}$, the magnetic field $\mathbf{W}$, and the temperature field ζ).
  - `DP diss_coefficients[2]:` The dissipation coefficient for $\mathbf{V}$ (ν), for $\mathbf{W}$ (η), and the diffusion coefficient for the scalar ζ (κ).
  - `string basis_type: SCFT`
  - `string integ_scheme:` The integrating scheme for time-stepping: `EULER, RK2, RK4`.
  - `Array<DP, 1> time_para(5): Tinit, Tfinal, Tdt, Tdiagnostics_init` in index 1..4 while reading from the file `field_para_file`.
  - `ifstream field_para_file:` File from where field parameters are read. It is in `data_dir_name/in`.
- **Connected to the energy transfer**
  - `int ETnospheres:` No of wavenumber spheres for the energy flux calculations

- `int ETnoshells:` No of shells for the shell-to-shell energy transfer calculations
- `int ET_shell_input_scheme:` 0 if shell radii are computed according to the scheme of Sec...; 1 if it is read from the `field_para_file`.
- `Array<DP, 1> ET_Rshell(MAXSIZE_R_shell):` Contains the shell radii that are read from the file `field_para_file` if `ET_shell_input_scheme = 1`.
- `int real_imag_switch:` Switch for computing the contributions from real and imaginary parts to the energy flux and shell-to-shell transfers.

- **Connected to the input of initial conditions**
  - `int field_input_proc:` The field input procedure for reading initial condition.
  - `int* N_in_reduced[D+1]:` Reduced array size for reading the fields in a smaller grid. Index:1..D.

- **Connected to the output functions**
  - `int* N_out_reduced[D+1]:` Reduced array size for outputting the fields in a smaller grid.
  - `Array<int, 2> out_k_array(MAXSIZE_out_k_array,4):` Contains the set wavenumbers that are read from the file `field_para_file`.
  - `int N_output_waveno:` Number of wavenumbers that are read from the file `field_para_file`.
  - `string nos_output_mode:` The output mode of the field variables: `ASCII, BINARY`.
  - `Array<int,1> time_save(10): Tglobal_save, Tfield_save, Trealfield_save, Tfield_reduced_save, Tfield_k_save, Tspectrum_save, Tflux_save, Tshell_to_shell_save, Tcout_save` in index 1..9 while reading from the file `field_para_file`.

- **Parameters connected to the RB convection**
  - `ifstream RB_slip_para_file:` The file from which the parameters of RB convection are read. It resides in `data_dir_name/in`.
  - `DP Pr:` Thermal Prandtl number
  - `DP Ra:` Rayleigh number
  - `DP r:` $r = Ra/Ra_c$
  - `DP k0:` `kfactor[2];` $k0 = \pi/\sqrt{2}$
  - `DP q:` `kfactor[3]`.

- DP qbyk0: $q/k0$.
- DP eta: Magnetic diffusivity.
- string Pr_switch: Takes on the values PRLARGE, PRSMALL, PRZERO.
- RB_Uscaling: Takes one of the values USMALL, ULARGE.
- DP w101, th101, th200: Lorenz variables read as initial condition if field_input_proc = 0.

17.2 Main program for magnetoconvection

The steps in the solver are

- Read_field_para(): Reads parameters of the field from the file field_para_file which resides in data_dir_name/in directory. It reads N[], diss_coefficients[], etc. See Sec. src_main for details.
- Read_RB_para(): Reads parameters for RB from RB_para_file that is resident in data_dir_name/in. The variables read are Pr, r, qbyk0, Pr_scaling, RB_Uscaling. If field_input_proc = 0, then Lorenz variables w101, th101, th200 are also read.
- Set up $Ra = r \times Ra_c$ and $kfactor[i]$.
- Set up the coefficients of $\nabla^2\mathbf{U}$, $\nabla^2\mathbf{B}$, and $\nabla^2\theta$ (see Appendix and Sec.).
- Constructor IncFluid U for the velocity field.
- Constructor IncVF W for the magnetic field $\mathbf{W}$.
- Constructor IncSF T for the temperature field ζ.
- Create fftw_plans.
- Open input files
- Initialize the field configurations using U.Init_cond(...). The choice of the function depends on field_input_proc.
- Close input files.
- Open output files.
- Start the Computation:
- For (U.Tnow = U.Tinit, U.Tnow <= U.Tfinal+0.0000001; U.Tnow += U.Tdt) {
  - U.Compute_force(W, T, Ra, Pr, Pr_switch, RB_Uscaling);
  - U.Compute_nlin(W, T, Pr_switch);

Table 17.1: ff

	Euler	RK2	RK4
$E^u(t=0)$			
$E^\zeta(t=0)$			

  - `U.Add_force(W, T, Pr_switch);`
  - `U.Compute_pressure();`
  - `U.Time_advance(W, T, Ra, Pr, Pr_switch, RB_Uscaling);`

- Close output files.
- Output results inside the loop.

17.3 Basic tests of the solver

17.3.1 Test the conservation of energy when dissipation and forcing are turned off.

- Initial conditions
- Parameter values
- Results:

17.3.2 Test pure dissipation by taking a single mode thus turning off nonlinearity

Appendix A

Integration schemes

In spectral method, the equations to be integrated are of the form:

$$\frac{\partial \zeta}{\partial t} + \kappa k^2 \zeta = R(\zeta(t), t). \tag{A.1}$$

We make a change of variable from ζ to $\bar{\zeta}$ defined as

$$\bar{\zeta}(t) = \exp(\kappa k^2 t)\zeta(t).$$

If we rewrite Eq. (A.1) in terms of $\bar{\zeta}$, it changes to the following equation:

$$\frac{\partial \bar{\zeta}(t)}{\partial t} = \exp(\kappa k^2 t) R(\bar{\zeta}(t), t). \tag{A.2}$$

We solve the above equation using different integrating schemes.

A.1 Euler's scheme

In Euler's scheme

$$\bar{\zeta}(t + \Delta t) = \bar{\zeta}(t) + \Delta t \times \exp(\kappa k^2 t) R(\bar{\zeta}(t), t)$$

or

$$\zeta(t + \Delta t) = [\zeta(t) + \Delta t \times R(\zeta(t), t)] \exp(-\kappa k^2 \Delta t).$$

A.2 Runge-Kutta second order (RK2)

The RK2 scheme time advances in two steps:

1. We step to the mid-point using Euler's scheme:

$$\bar{\zeta}_{mid}(t + \Delta t/2) = \bar{\zeta}(t) + \frac{\Delta t}{2} \times \exp(\kappa k^2 t) R(\bar{\zeta}(t), t)$$

or

$$\zeta_{mid}(t + \Delta t/2) = [\zeta(t) + \frac{\Delta t}{2} \times R(\zeta(t), t)] \exp(-\kappa k^2 \Delta t/2).$$

2. We compute the function $R(\zeta(t_{mid}), t_{mid})$ at the mid-point. The function ζ is computed at time $t+\Delta t$ using the slope at the mid-point. The formulas are as follows:

$$\bar{\zeta}(t+\Delta t) = \bar{\zeta}(t) + \Delta t \times \exp(\kappa k^2 t_{mid}) R(\bar{\zeta}(t_{mid}), t_{mid})$$

or

$$\zeta(t+\Delta t) = \zeta(t)\exp(-\kappa k^2 \Delta t) + \Delta t \times R(\zeta(t_{mid}), t_{mid})\exp(-\kappa k^2 \Delta t/2).$$

This is the final $\zeta(t+\Delta t)$ in the RK2 scheme.

A.3 Runge-Kutta fourth order (RK4)

In this scheme the time advance is done in four steps outlined below:

1. we step to the mid-point using Euler's scheme:

$$\bar{\zeta}_{mid1}(t+\Delta t/2) = \zeta(t) + \frac{\Delta t}{2} \times \exp(\kappa k^2 t) R(\bar{\zeta}(t), t)$$

or

$$\zeta_{mid1}(t+\Delta t/2) = [\zeta(t) + \frac{\Delta t}{2} \times R(\zeta(t), t)]\exp(-\kappa k^2 \Delta t/2).$$

We also compute C_1 that would added finally in the computation of the final $\zeta(t+\Delta t)$.

$$C_1 = \Delta t \times R(\zeta(t), t)\exp(-\kappa k^2 t).$$

2. We compute the rhs at the midpoint with $\zeta_{mid1}(t+\Delta t/2)$ as the value for the funcion. Using the new value of rhs, we compute $\zeta_{mid2}(t_{mid})$:

$$\bar{\zeta}_{mid2}(t+\Delta t/2) = \zeta(t) + \frac{\Delta t}{2} \times \exp[\kappa k^2 (t+\Delta t/2)] R(\bar{\zeta}_{mid1}(t+\Delta t/2), t+\Delta t/2)$$

or

$$\zeta_{mid2}(t+\Delta t/2) = \zeta(t)\exp(-\kappa k^2 \Delta t/2) + \frac{\Delta t}{2} \times R(\zeta_{mid1}(t+\Delta t/2), t+\Delta t/2).$$

We also compute C_2

$$C_2 = \Delta t \times R(\zeta_{mid1}(t+\Delta t/2), t+\Delta t/2)\exp[\kappa k^2 (t+\Delta t/2)].$$

3. We compute the rhs at the midpoint with $\zeta_{mid2}(t+\Delta t/2)$ as the value for the funcion. Using the new value of rhs, we compute $\zeta_3(t+\Delta t)$:

$$\bar{\zeta}_3(t+\Delta t) = \zeta(t) + \Delta t \times \exp[\kappa k^2 (t+\Delta t/2)] R(\bar{\zeta}_{mid2}(t+\Delta t/2), t+\Delta t/2)$$

or

$$\zeta_3(t+\Delta t) = \zeta(t)\exp(-\kappa k^2 \Delta t) + \Delta t \times R(\zeta_{mid2}(t+\Delta t/2), t+\Delta t/2)\exp(-\kappa k^2 \Delta t/2).$$

We also compute C_4

$$C_3 = \Delta t \times R(\zeta_{mid2}(t+\Delta t/2), t+\Delta t/2)\exp[\kappa k^2 (t+\Delta t/2)].$$

4. In the final step we compute the rhs at $t = t + \Delta t$ with $\zeta_3(t + \Delta t)$ as the function. After this we compute C_4 as

$$C_4 = \Delta t \times R(\zeta_3(t + \Delta t), t + \Delta t) \exp[\kappa k^2 (t + \Delta t)].$$

The final value of the function at $t + \Delta t$ is

$$\bar{\zeta}_4(t + \Delta t) = \overline{\zeta}(t) + \frac{1}{6}(C_1 + 2C_2 + 2C_3 + C_4)$$

or

$$\begin{aligned} \zeta_4(t + \Delta t) \quad = \quad & \zeta(t) \exp(-\kappa k^2 \Delta t) + \frac{\Delta t}{6} \times R(\zeta(t), t) \exp(-\kappa k^2 \Delta t). \\ & \frac{\Delta t}{3} \times R(\zeta_{mid1}(t + \Delta t/2), t + \Delta t/2) \exp(-\kappa k^2 \Delta t/2) \\ & \frac{\Delta t}{3} \times R(\zeta_{mid2}(t + \Delta t/2), t + \Delta t/2) \exp[-\kappa k^2 \Delta t/2] \\ & \frac{\Delta t}{6} \times R(\zeta_3(t + \Delta t), t + \Delta t). \end{aligned}$$

$\zeta_4(t+\Delta t)$ is the desired value of the function at time $t+\Delta t$ in RK4 scheme.

Exercise: Solve the equation

$$\frac{\partial \zeta}{\partial t} + \kappa k^2 \zeta = \zeta$$

using Euler and RK2 scheme. Compare the result with the exact solution.

Appendix B

Rayleigh Benard Convection With Free Slip Boundary Condition

B.1 Equations

B.1.1 Finite Prandtl number

We choose length scale as d, time scale as d^2/κ, velocity scale as κ/d, and temperature scale as $(\Delta T)_0$, then the equations are

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla\sigma + RP\theta + P\nabla^2\mathbf{u} \tag{B.1}$$

$$\frac{\partial \theta}{\partial t} + (\mathbf{u}\cdot\nabla)\theta = u_1 + \nabla^2\theta. \tag{B.2}$$

We can argue using dimensional analysis that the large-scale velocity $u_L \sim \sqrt{RP}$ and $\theta_L \sim 1$. This equation is not similar to nondimensionaled NS equation where we normalize the large-scale velocity to be 1. Note that dimensionful $u_L \sim (\kappa/d)\sqrt{RP}$ and $\delta T \sim (\Delta T)_0$.

If we use the convective velocity scale $\sqrt{\alpha(\Delta T)_0 g d}$ as the velocity scale, and d as the length scale, we obtain an equation similar to nondimensionalized NS equation used in turbulence. The equations are

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla\sigma + \theta + \sqrt{\frac{P}{R}}\nabla^2\mathbf{u} \tag{B.3}$$

$$\frac{\partial \theta}{\partial t} + (\mathbf{u}\cdot\nabla)\theta = u_1 + \frac{1}{\sqrt{PR}}\nabla^2\theta. \tag{B.4}$$

For large R and finite P, both the viscous terms could be ignored. Hence

$u_L \sim \theta_L \sim 1$ as expected. With this again, dimensionful large-scale velocity is $u_L \sim \sqrt{\alpha(\Delta T)_0 g d} \sim (\kappa/d)\sqrt{RP}$.

Eqs. (B.1, B.13) are equivalant to Eqs. (B.3, B.4). However the latter set is more useful for larger R (turbulence) because dt required for the latter set is $\sqrt{RP}$ times larger compared to the former set (CFL condition). Hence the latter set is more accurate. For smaller R, the earlier equations are as good, probably better (??). We implement both the schemes in our code.

B.1.2 Small Prandtl number

The above two sets of equations are valid for finite P. For small P, the diffusive term of Eq. (B.4) becomes comparable to u_1 and cannot be ignored. In fact, for very small P

$$u_L \sim \frac{\theta_L}{L^2\sqrt{PR}}. \tag{B.5}$$

The thermal energy supplied at the large-scale drives the velocity field through energy cascade. Note that thermal energy is supplied only at the large-scale because the spectrum of temperature fluctuations is very steep [preprint]. The equation for energy cascade yields

$$\epsilon_u \sim \frac{u_L^3}{L} \sim \theta_L u_L \sim \sqrt{PR} u_L^2 L^2.$$

Therefore,

$$u_L \sim \sqrt{PR} L^3,$$

which yields the dimensionful $u_L \sim R(\nu/d)$. Substitution of this expression in Eq. (B.5) yields $\theta_L \sim PRL^5$. This expression is consistent with the earlier statement that the thermal energy spectrum is very steep.

Note that Peclet number $Pe = u_L d/\kappa \sim \sqrt{RP}$ for large P, and $Pe \sim RP$ for small P. Also, temperature fluctuations is very small because of high thermal diffusivity.

For small P, it is convenient to use $\theta' = \theta/P$ that is finite. In terms of θ' the RB equations (B.3, B.4) become

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla\sigma + P\theta' + \sqrt{\frac{P}{R}}\nabla^2\mathbf{u} \tag{B.6}$$

$$P\left[\frac{\partial \theta'}{\partial t} + (\mathbf{u}\cdot\nabla)\theta'\right] = u_1 + \sqrt{\frac{P}{R}}\nabla^2\theta'. \tag{B.7}$$

These equations use large-scale velocity $\sqrt{\alpha(\Delta T)_0 g d}$ as velocity scale. However if we use small-scale velocity ν/d as velocity scale, then we obtain

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla\sigma + R\theta' + \nabla^2\mathbf{u} \tag{B.8}$$

$$\frac{\partial \theta'}{\partial t} + (\mathbf{u}\cdot\nabla)\theta' = P^{-1}\left[u_1 + \nabla^2\theta'\right]. \tag{B.9}$$

If we set $P = 0$, the Eq. (B.8,B.9) becomes

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla\sigma + R\theta' + \nabla^2\mathbf{u} \quad \text{(B.10)}$$

$$0 = u_1 + \nabla^2\theta'. \quad \text{(B.11)}$$

B.2 Implementation (2D)

The expansion of velocity and temperature fields due to free slip boundary condition:

$$u^{(1)}_{\mathbf{j}} = \sum_{m,k_y} \hat{u}^{(1)}_{m,k_y} 2\sin\left(\pi\frac{mj_x}{N_x}\right)\exp\left(2\pi i\frac{j_y k_y}{N_y}\right),$$

$$u^{(2)}_{\mathbf{j}} = \sum_{k_y} \hat{u}^{(2)}_{0,k_y}\exp\left(2\pi i\frac{j_y k_y}{N_y}\right) + \sum_{m,k_y} \hat{u}^{(2)}_{m,k_y} 2\cos\left(\pi\frac{mj_x}{N_x}\right)\exp\left(2\pi i\frac{j_y k_y}{N_y}\right),$$

$$\theta_{\mathbf{j}} = \sum_{m,k_y} \hat{\theta}_{m,k_y} 2\sin\left(\pi\frac{mj_x}{N_x}\right)\exp\left(2\pi i\frac{j_y k_y}{N_y}\right),$$

Arrays$(N[1], N[2])$ of size $N[1]\times(N[2]/2+1)$.
The equations in Fourier space are

$$\frac{\partial u^{(1)}}{\partial t} + SFT[\partial_j(u^{(j)}u^{(1)})] = -\sigma + RP\theta(m,k_y) + PK^2u^{(1)} \quad \text{(B.12)}$$

$$\frac{\partial \theta}{\partial t} + (\mathbf{u}\cdot\nabla)\theta = u_1 + \nabla^2\theta. \quad \text{(B.13)}$$

Note that the derivative operators in SCFT are

$$\begin{aligned} SFT(\partial_x f) &= -K_x CFT(f) \\ CFT(\partial_x f) &= K_x SFT(f) \\ SFT(\partial_{y,z} f) &= iK_{y,z} SFT(f) \\ CFT(\partial_{y,z} f) &= iK_{y,z} CFT(f). \end{aligned}$$

Note that $K_x = m\pi$, $K_y = k_y k_0$, $K_z = k_z q$.
The nonlinear terms are written as

$$\begin{aligned} SFT[\partial_j(u^{(j)}u^{(1)})] &= -m\pi CFT[u^{(1)}u^{(1)}] + ik_y k_0 SFT[u^{(2)}u^{(1)}] \\ CFT[\partial_j(u^{(j)}u^{(2)})] &= m\pi SFT[u^{(1)}u^{(2)}] + ik_y k_0 CFT[u^{(2)}u^{(2)}] \\ SFT[\partial_j(u^{(j)}\theta)] &= -m\pi CFT[u^{(1)}\theta] + ik_y k_0 SFT[u^{(2)}\theta] \end{aligned}$$

B.3 Implementation (3D)

$$
\begin{aligned}
u_{\mathbf{j}}^{(1)} &= \sum_{m,\mathbf{k}} \hat{u}_{m,\mathbf{k}}^{(1)} 2\sin\left(\pi\frac{mj_x}{N_x}\right)\exp\left(2\pi i\sum_{s>1}\frac{j_s k_s}{N_s}\right),\\
u_{\mathbf{j}}^{(2)} &= \sum_{\mathbf{k}} \hat{u}_{0,\mathbf{k}}^{(2)} \exp\left(2\pi i\sum_{s>1}\frac{j_s k_s}{N_s}\right)+\sum_{m,\mathbf{k}} \hat{u}_{m,\mathbf{k}}^{(2)} 2\cos\left(\pi\frac{mj_x}{N_x}\right)\exp\left(2\pi i\sum_{s>1}\frac{j_s k_s}{N_s}\right),\\
u_{\mathbf{j}}^{(3)} &= \sum_{\mathbf{k}} \hat{u}_{0,\mathbf{k}}^{(3)} \exp\left(2\pi i\sum_{s>1}\frac{j_s k_s}{N_s}\right)+\sum_{m,\mathbf{k}} \hat{u}_{m,\mathbf{k}}^{(3)} 2\cos\left(\pi\frac{mj_x}{N_x}\right)\exp\left(2\pi i\sum_{s>1}\frac{j_s k_s}{N_s}\right),\\
\theta_{\mathbf{j}} &= \sum_{m,\mathbf{k}} \hat{\theta}_{m,\mathbf{k}} 2\sin\left(\pi\frac{mj_x}{N_x}\right)\exp\left(2\pi i\sum_{s>1}\frac{j_s k_s}{N_s}\right),
\end{aligned}
$$

$$
\begin{aligned}
SFT[\partial_j(u^{(j)}u^{(1)})] &= -m\pi CFT[u^{(1)}u^{(1)}]+ik_yk_0SFT[u^{(2)}u^{(1)}]+ik_zqSFT[u^{(3)}u^{(1)}]\\
CFT[\partial_j(u^{(j)}u^{(2)})] &= m\pi SFT[u^{(1)}u^{(2)}]+ik_yk_0CFT[u^{(2)}u^{(2)}]+ik_zk_0CFT[u^{(3)}u^{(2)}]\\
CFT[\partial_j(u^{(j)}u^{(3)})] &= m\pi SFT[u^{(1)}u^{(3)}]+ik_yk_0CFT[u^{(2)}u^{(3)}]+ik_zk_0CFT[u^{(3)}u^{(3)}]\\
SFT[\partial_j(u^{(j)}\theta)] &= -m\pi CFT[u^{(1)}\theta]+ik_yk_0SFT[u^{(2)}\theta]+ik_zk_0SFT[u^{(3)}\theta]
\end{aligned}
$$

B.4 Magnetoconvection

For magnetic field we use the same boundary condition as the velocity field. So the computation of nonlinear term for magnetic field will be on the same lines as the velocity field.

$$
\begin{aligned}
SFT[\partial_j(Zm^{(j)}Zp^{(1)})] &= -m\pi CFT[Zm^{(1)}Zp^{(1)}]+ik_yk_0SFT[Zm^{(2)}Zp^{(1)}]+ik_zqSFT[Zm^{(3)}Zp^{(1)}]\\
CFT[\partial_j(Zm^{(j)}Zp^{(2)})] &= m\pi SFT[Zm^{(1)}Zp^{(2)}]+ik_yk_0CFT[Zm^{(2)}Zp^{(2)}]+ik_zk_0CFT[Zm^{(3)}Zp^{(2)}]\\
CFT[\partial_j(Zm^{(j)}Zp^{(3)})] &= m\pi SFT[Zm^{(1)}Zp^{(3)}]+ik_yk_0CFT[Zm^{(2)}Zp^{(3)}]+ik_zk_0CFT[Zm^{(3)}Zp^{(3)}]\\
SFT[\partial_j(u^{(j)}\theta)] &= -m\pi CFT[u^{(1)}\theta]+ik_yk_0SFT[u^{(2)}\theta]+ik_zk_0SFT[u^{(3)}\theta]
\end{aligned}
$$

Appendix C

Design Issues

- Why do we choose dimensions using compiler directive?
- Why the first direction was chosen for Sin/Cos transform?

Appendix D

Memory and Time requirements

D.1 Fluid

- $\mathbf{V}(\mathbf{k})$, $\mathbf{V}(\mathbf{r})$ (D components each): The velocity field
- $\mathbf{nlin}(\mathbf{k})$ (D components): The nonlinear terms $(\mathbf{U} \cdot \nabla)\mathbf{U}$
- $p(\mathbf{k})$ (1 component): The pressure field
- $VF\_\text{temp}(\mathbf{k})$ (1 component): Temporary field for velocity field operations
- $\mathbf{Force}(\mathbf{k})$ (D components): Force field for Temporary field for velocity field operations.
- $\mathbf{Vfrom}(\mathbf{k})$ (D components): Energy Giver (useful for energy transfer function)
- temoET($\mathbf{k}$) (1 component): Temporary field for energy transfer functions.
- Miscelleneous

Therefore total memory requirement is

$$mem = (4D + 3)\Pi(N(i)).$$

For 512^3 simulation, the requirement in 3D is 15 GB for double precision. In 2D for 1024^2 runs, the requriment is $11 \times 8 = 88$ MB, rather small.

D.2 Passive Scalar and RB Convection

- $\mathbf{V}(\mathbf{k})$, $\mathbf{V}(\mathbf{r})$ (D components each): The velocity field

- **nlin**(**k**) (D components): The nonlinear terms $(\mathbf{U}\cdot\nabla)\mathbf{U}$
- p(**k**) (1 component): The pressure field
- $VF_$temp(**k**) (1 component): Temporary field for velocity field operations
- **Force**(**k**) (D components): Force field for the velocity field
- **Vfrom**(**k**) (D components): Energy Giver (useful for energy transfer function)
- temoET(**k**) (1 component): Temporary field for energy transfer functions.
- F(**k**) (1 component): Scalar field
- $SF_$temp(**k**) (1 component): Temporary field for the scalar field operations
- $Force$(**k**) (1 component): Force field for the scalar field.
- Miscelleneous

Therefore total memory requirement is

$$mem = (4D + 6)\Pi(N(i)).$$

For 512^3 simulation, the requirement in 3D is 18 GB for double precision. In 2D for 1024^2 runs, the requriment is $11 \times 8 = 88$ MB, rather small.

D.3 MHD

The memorey requirement for MHD is twice of fluid because we create two IncVF in this simulation. For 512^3, the requirement will be 30 GB.

D.4 Magnetoconvection

Here we have 2 IncVF and 1 IncSF. For 512^3, the requirement will be 33 GB.

Appendix E

Caution

1. Compute $\nabla.\mathbf{V}$ in the beginning of the loop. It uses F field that is reserved for the pressure field. So the divergence computation must be done before the pressure computation starts.
2. The output functions calls should not be moved. They require the state of the variables at that point. Specifically, output_global needs the current state of *nlin*.