Variable-Density Mixing Layer Entrainment

Paul Bartholomew

February 25, 2019

1 Introduction

The aim of this study is to investigate the entrainment effects of a non-Boussinesq jet. For negligible Richardson number and infinite Reynolds number, the only non-dimensional parameter governing the jet is the density ratio ρ_j/ρ_a where subscripts j, a represent jet and ambient conditions respectively. We would like to know how does this ratio affect the entrainment coefficient $\alpha = \dot{V}_e/\mathcal{U}_j$ where $\dot{V}_e$ is the flow rate per unit surface of ambient fluid entrained within the jet and $\mathcal{U}_j$ is a velocity scale of the jet. According to experimental data of RICOU&SPALDING1961 the entrainment coefficient scales with density ratio as

$$\alpha = \alpha_0 \sqrt{\frac{\rho_j}{\rho_a}} \tag{1}$$

where $\alpha_o = 0.08$ is the entrainment parameter for an iso-density jet, however a theoretical basis for this is lacking and developing this is the basis for the present study.

2 Setup

Rather than simulate a full jet with the wide range of length scales that entails, a temporal mixing layer will be used as an analogue, replacing the jet axis with time.

Table 1: Overview of cases

Case	Description
mx01	Testing initial setup
v1	Validation study

2.1 Case mx01

This was initially testing the setup based on the 3D case of GOLANSKI2005 - it would appear to be working superficially. For our purposes however, the low Reynolds number ($Re = 400$) may be too low.

2.2 Case v1

This is a validation study to confirm the solver is working for high-Reynolds, variable-density low-speed flows. It is based on the simulations performed by ALMAGRO2017 for temporally evolving mixing layers with $1 \leq \rho_2/\rho_2 \leq 8$ and $4200 \lesssim Re_w \lesssim 7000$ with

$$Re_w = \frac{\Delta u \delta_w}{\nu} \tag{2}$$

where $\Delta u = u_1 - u_2 = 1$ is the initial free-stream velocity difference and δ_w is the initial vorticity thickness.

The Reynolds number based on the momentum thickness δ_m, used as the reference length scale, is $Re = 160$ for all cases.

2.2.1 Domain and Mesh

The domain is of dimensions $461 \times 368 \times 173$ and is meshed with $1536 \times 851 \times 576$ mesh nodes. The reference vertical resolution in the region $|y| \leq 20$ is $\Delta y = 0.2$ and is stretched at 1% out to $|y| < 150$ where $\Delta y = 0.85$, the resolution is then increased again towards the walls with stretching 3% to give $\Delta y = 0.3$ at the walls. Incompact3D cannot do this stretching/unstretching as far as I know - therefore best approach is probably to try and have the average resolutions match in the inner ($|y| \leq 20$) and outer layers. The resolution in x and z is constant.

```
461.0    #xlx  # Lx (Size of the box in x−direction)
368.0    #yly  # Ly (Size of the box in y−direction)
173.0    #zlz  # Lz (Size of the box in z−direction)
```

Listing 1: Code to specify the domain

For mesh convergence, several meshes will be considered as given in table 2

Table 2: Mesh sizes

Mesh	N_x	N_y	N_z	N_{tot}
m1	32	33	16	16896
m2	64	65	32	133120
m3	128	129	64	1056768
m4	256	257	128	8421376
m5	512	513	256	67239936
m6	1024	1025	512	537395200

The mesh can be stretched by the following code:

```
0      #istret # y mesh refinement (0:no, 1:center, 2:both sides, 3:bottom)
0.3    #beta   # Refinement parameter (beta)
```

Listing 2: Mesh stretching

The code to specify the mesh for each case is given as

1. Mesh m1

```
integer,parameter :: nx=32, ny=33, nz=16
```

Listing 3: Case m1 mesh specification

2. Mesh m2

```
integer,parameter :: nx=64, ny=65, nz=32
```

Listing 4: Case m2 mesh specification

3. Mesh m3

```
integer,parameter :: nx=128, ny=129, nz=64
```

Listing 5: Case m3 mesh specification

4. Mesh m4

```
integer,parameter :: nx=256, ny=257, nz=128
```

Listing 6: Case m4 mesh specification

5. Mesh m5

```
integer,parameter :: nx=512, ny=513, nz=256
```

Listing 7: Case m5 mesh specification

6. Mesh m6

```
integer,parameter :: nx=1024, ny=1025, nz=512
```

Listing 8: Case m6 mesh specification

2.2.2 Initial conditions

The flow is a mixing layer (`itype=4`) and we want to use a function to specify the initial fluctuations/initial conditions as specified by the following code:

```
4       #itype  # Type of flow   (1: Constant flowfield, 4: Mixing layer, 6: Taylor-Green)
1       #iin    # Inflow condition (1: classic, 2: turbinit)
```

Listing 9: Setup a mixing layer with specified fluctuations/initialisation

The mean velocity field is specified according to the hyperbolic-tangent profile:

$$u(y) = \frac{\Delta u}{2} \tanh\left(-\frac{y}{2\delta_m}\right) \tag{3}$$

and the density field as

$$\rho(y) = \rho_0 \left(1 + \lambda(s) \tanh\left(-\frac{y}{2\delta_m}\right)\right) \tag{4}$$

where $\lambda(s) = (s-1)/(s+1)$ and $s = \rho_2/\rho_1$ is the density ratio.

These (mean) initial conditions are set by the following `Fortran90` code:

```
DO i = 1, xsize(1)
   DO j = 1, xsize(2)
      y = (j + xstart(2) - 2) * dy - 0.5_mytype * yly
      DO k = 1, xsize(3)
         ux1(i, j, k) = ux1(i, j, k) + 0.5_mytype * (u1 - u2) * TANH(-0.5_mytype * y)
         rho1(i, j, k) = (1._mytype + ((dens2 / dens1 - 1._mytype) &
                              / (dens2 / dens1 + 1._mytype)) &
                              * TANH(-0.5_mytype * y))
      ENDDO
   ENDDO
ENDDO
```

Listing 10: Code to set mean initial velocity and density profiles

The velocities `u1` and `u2` are set by the following code:

```
-0.5     #u1     # u1 (max velocity) (for inflow condition)
0.5      #u2     # u2 (min velocity) (for inflow condition)
```

Listing 11: Code to set mean velocities

whilst there is a case-specific code to set the densities with a case identifier according to table 3 with associated code in §1-4.

Table 3: Case specifications

Case	s	Re	Pr
s1	1	160	0.7
s2	2	160	0.7
s4	4	160	0.7
s8	8	160	0.7

In addition to the mean initial conditions, an initial perturbation is added to the velocity field. The simplest perturbation is a random fluctuation that decays as a function of $|y|$. This fluctuation is setup by first setting the velocity field to a field of random numbers $0 \leq u_x, u_y, u_z \leq 1$ which is then shifted and scaled to $-u' \leq u_x, u_y, u_z \leq u'$ where u' is set at runtime by the `noise` keyword.

```
! Create the random field
call system_clock(count=code)
call random_seed(size = ii)
call random_seed(put = code+63946*nrank*(/ (i - 1, i = 1, ii) /)) !

call random_number(ux1)
call random_number(uy1)
call random_number(uz1)

! Shift and scale the random field
do k=1, xsize(3)
  do j=1, xsize(2)
    do i=1, xsize(1)
      ux1(i, j, k) = noise * (1._mytype - 2._mytype * ux1(i, j, k))
      uy1(i, j, k) = noise * (1._mytype - 2._mytype * uy1(i, j, k))
      uz1(i, j, k) = noise * (1._mytype - 2._mytype * uz1(i, j, k))
    enddo
  enddo
enddo

!modulation of the random noise
do k=1,xsize(3)
   z = float(k + xstart(3) - 2) * dz - zlz / 2._mytype
   do j=1,xsize(2)
      if (istret.eq.0) then
         y=(j+xstart(2)-2)*dy-yly/2._mytype
      else
         y=yp(j+xstart(2)-1)-yly/2._mytype
      endif

      do i=1,xsize(1)
         x = (i + xstart(1) - 2) * dx

         um = noise * exp(-y**2)

         ux1(i,j,k)=um*ux1(i,j,k)
         uy1(i,j,k)=um*uy1(i,j,k)
         uz1(i,j,k)=um*uz1(i,j,k)
      enddo
   enddo
enddo
```

Listing 12: Fortran90 code to generate a random initial velocity field about zero, decaying with $|y|$

```
0.1    #noise# Turbulence intensity (1=100%) !! Initial condition
0.05    #noise1# Turbulence intensity (1=100%) !! Inflow condition
```

Listing 13: Runtime code to set fluctuation scale (noise1 can be ignored)

Alternatively, we could read an initial flow field

```
0           #ilit     # Read initial flow field ?
```

Listing 14: Initialise from restart file

1. Case s1

   ```
   1.0     #dens1 # dens1
   1.0     #dens2 # dens2
   ```

   Listing 15: Case s1 density specification

2. Case s2

   ```
   1.0     #dens1 # dens1
   2.0     #dens2 # dens2
   ```

   Listing 16: Case s2 density specification

3. Case s4

```
1.0     #dens1 # dens1
4.0     #dens2 # dens2
```

Listing 17: Case s4 density specification

4. Case s8

```
1.0     #dens1 # dens1
8.0     #dens2 # dens2
```

Listing 18: Case s8 density specification

2.2.3 Boundary conditions

The domain is periodic in x and z and free-slip/zero-gradient boundary conditions are applied at top and bottom boundaries. The boundary-condition section of `incompact3d.prm` follows as

```
0       #nclx   # nclx  (BC)
1       #ncly   # ncly  (BC)
0       #nclz   # nclz  (BC)
```

Listing 19: Boundary condition specification

2.2.4 Gravity

We can specify a gravity field by setting the Froude number in each direction, defined as

$$Fr = \frac{\Delta u}{\sqrt{gl}} \tag{5}$$

where Δu is the velocity scale, g the magnitude of the gravity vector and l the length scale.

In QuasIncompact3D there is a Froude vector to allow gravity vectors in any (combination of) coordinate direction(s), with the default value 0 for this case without gravity

```
0.      #frx   # Froude number in x direction (frx = 0 gives no gravity)
0.      #fry   # Froude number in y direction
0.      #frz   # Froude number in z direction
```

2.2.5 Time stepping

In ALMAGRO2017 they run at $CFL = 0.5$ which for $\Delta u = 1 \Rightarrow u_1 = -u_2 = -0.5$ gives on the finest grid `m6` a timestep of $\Delta t = 0.225$ (basing CFL on Δu) for 3106.72 timesteps to reach $t = 700$. For a round number of steps 3200, this yields a timestep of $\Delta t = 0.21875$. The code to specify the timestep, number of timesteps and time integrator follows:

```
0.21875  #dt    # Time step
```

Listing 20: Code to specify timestep

```
1     #ifirst # First iteration
3200  #ilast  # Last iteration
```

Listing 21: Code to specify number of steps

```
1       #nscheme# Temporal scheme (1:AB2, 2: RK3, 3:RK4, 4:AB3)
```

Listing 22: Code to specify time integrator

The period between writing data and saving checkpoints can be specified by changing `imodulo` and `isave` respectively.

```
640       #isave    # Frequency for writing backup file
32        #imodulo  # Frequency for visualization for VISU_INSTA
```

2.2.6 Fluid properties

Apart from the densities, the Reynolds and Prandtl numbers are constant as given in table 3, set by the following code

```
160. #re    # Reynolds number
0.7  #pr    # Prandtl number
1.   #sc    # Schmidt number (if passive scalar)
```

Listing 23: Common fluid properties specification

We can also control whether the properties are variable in space or not:

```
1       #iprops # (0: constant properties, 1: variable properties)
```

Listing 24: Code to enable/disable variable properties

2.2.7 Low Mach Number

To control the LMN behaviour of the solver, we can first enable or disable it:

```
0       #ilmn   # (0:incompressible, 1: Low Mach Number)
```

Listing 25: Disable LMN for case s1

```
1       #ilmn   # (0:incompressible, 1: Low Mach Number)
```

Listing 26: Enable LMN for all other (variable-density) cases

We can then decide to solve for temperature (instead of density) and whether ours is a multi-component flow - for all cases we want to transport density and our flow is single component

```
0       #isolvetemp# (0: solve for density, 1: solve for temperature)
0       #imulticomponent
```

Listing 27: Disable temperature-based solver and multicomponent flow

If we are using the constant-coefficient solver (which we will choose later) we have to select an approximation for $\left.\frac{\partial \rho}{\partial t}\right|^{n+1}$ which we use Golanski's approach for (`nrhoscheme=3`):

```
0       #nrhoscheme# Density scheme (how to approximate drhodt^{k+1})
```

Listing 28: Select approximation of $\frac{\partial\rho}{\partial t}\Big|^{n+1}$

Finally we can choose whether to use constant- or variable-coefficient Poisson solver. For cases s2 and s4 we should be able to use the constant-coefficient Poisson solver (similarly for s1):

```
0       #ivarcoeff# (0:constant−coefficient Poisson equation, 1:variable−coefficient Poisson equati
```

Listing 29: Enable constant-coefficient Poisson solver

whereas for case s8 we will need to solve the variable-coefficient Poisson equation

```
1       #ivarcoeff# (0:constant−coefficient Poisson equation, 1:variable−coefficient Poisson equati
```

Listing 30: Enable variable-coefficient Poisson solver

When using the variable-coefficient Poisson solver, we need to choose how to compute $\widetilde{\rho}$ and the tolerance for the solver (this can be set the same for all cases, affecting only s8):

```
1       #npoisscheme# How to compute rho0 in var−coeff poisson equation? (0: rho0 = MIN(rho), Anyth
1.0e−14 #tol     # Tolerance for Poisson equation
```

2.2.8 Numerics

We can solve either the rotational or (quasi) skew-symmetric form of the advection terms in momentum equation:

```
1       #iskew  # (0:urotu, 1:skew, for the convective terms)
```

Listing 31: Choice of momentum advection

2.2.9 Scalar

There are no scalars to solve:

```
0       #iscalar# (0: no scalar, 1:scalar)
```

Listing 32: Disable passive scalar solver

2.2.10 Online postprocessing

Due to the size of the largest problem (`m6`) much of the postprocessing must be performed online. As will be seen, this introduces several new variables to the code: $\overline{\rho}$, $\boldsymbol{U}$, $\boldsymbol{R}$ and $\overline{\epsilon}$ - the Reynolds-averaged density, the Favré-averaged velocity, the Reynolds stress tensor and the Reynolds-averaged dissipation rate of fluctuating kinetic energy.

1. Setup

   The variables for post-processing must all be defined:

```
real(mytype), save, allocatable, dimension(:,:,:) :: rhomean
real(mytype), save, allocatable, dimension(:,:,:) :: ufmean, vfmean, wfmean
real(mytype), save, allocatable, dimension(:,:,:) :: Rxx, Rxy, Rxz, Ryy, Ryz, Rzz, rdiss
```

and allocated:

```
if (ilmn.ne.0) then
   allocate(rhomean(xstS(1):xenS(1), xstS(2):xenS(2), xstS(3):xenS(3)))
   allocate(ufmean(xstS(1):xenS(1), xstS(2):xenS(2), xstS(3):xenS(3)))
   allocate(vfmean(xstS(1):xenS(1), xstS(2):xenS(2), xstS(3):xenS(3)))
   allocate(wfmean(xstS(1):xenS(1), xstS(2):xenS(2), xstS(3):xenS(3)))
else
   allocate(rhomean(1, 1, 1))
   allocate(ufmean(1, 1, 1))
   allocate(vfmean(1, 1, 1))
   allocate(wfmean(1, 1, 1))
endif
allocate(Rxx(xstS(1):xenS(1), xstS(2):xenS(2), xstS(3):xenS(3)))
allocate(Rxy(xstS(1):xenS(1), xstS(2):xenS(2), xstS(3):xenS(3)))
allocate(Rxz(xstS(1):xenS(1), xstS(2):xenS(2), xstS(3):xenS(3)))
allocate(Ryy(xstS(1):xenS(1), xstS(2):xenS(2), xstS(3):xenS(3)))
allocate(Ryz(xstS(1):xenS(1), xstS(2):xenS(2), xstS(3):xenS(3)))
allocate(Rzz(xstS(1):xenS(1), xstS(2):xenS(2), xstS(3):xenS(3)))
allocate(rdiss(xstS(1):xenS(1), xstS(2):xenS(2), xstS(3):xenS(3)))
```

note that to save space we have exploited symmetry of the Reynolds stress tensor.

2. Reynolds-averaged density

   The Reynolds-averaged density, defined as:

$$\overline{\rho} = \frac{1}{T}\int_0^T \rho dt \tag{6}$$

   is computed by the following code:

```
!! rhomean=rho1
call fine_to_coarseS(1,rho1,tmean)
rhomean(:,:,:)=rhomean(:,:,:)+tmean(:,:,:)
```

3. Favré-averaged velocity

   The Favré-averaged velocity, defined as:

$$U = \frac{\overline{\rho u}}{\overline{\rho}} = \frac{\overline{\rho}\ \overline{u} + \overline{\rho' u'}}{\overline{\rho}} \tag{7}$$

   is computed by the following code:

```
!! Favre averages
ta1(:,:,:) = rho1(:,:,:) * ux1(:,:,:)
call fine_to_coarseS(1, ta1, tmean)
ufmean(:,:,:) = (ufmean(:,:,:) + tmean(:,:,:)) / rhomean(:,:,:)

ta1(:,:,:) = rho1(:,:,:) * uy1(:,:,:)
```

```
call fine_to_coarseS(1, ta1, tmean)
vfmean(:,:,:) = (vfmean(:,:,:) + tmean(:,:,:)) / rhomean(:,:,:)

ta1(:,:,:) = rho1(:,:,:) * uz1(:,:,:)
call fine_to_coarseS(1, ta1, tmean)
vfmean(:,:,:) = (vfmean(:,:,:) + tmean(:,:,:)) / rhomean(:,:,:)
```

To compute $\overline{\rho \boldsymbol{u}}$ as part of the computation of the Favré-averaged velocity we first do $\boldsymbol{U} \leftarrow \overline{\rho}\boldsymbol{U}$ as follows:

```
!! Get Favre averages to just AVG(rho u)
ufmean(:,:,:) = rhomean(:,:,:) * ufmean(:,:,:)
vfmean(:,:,:) = rhomean(:,:,:) * vfmean(:,:,:)
wfmean(:,:,:) = rhomean(:,:,:) * wfmean(:,:,:)
```

so that the entire computation of $\overline{\rho}$ and $\boldsymbol{U}$ is as

```
if (ilmn.ne.0) then
   <<src:ufmean-prep.f90>>
   <<src:rhomean.f90>>
   <<src:ufmean.f90>>
else
   rhomean(:,:,:) = 1._mytype
endif
```

4. Reynolds stresses

$$R_{ij} = \frac{\overline{\rho {u_i}'' {u_j}''}}{\overline{\rho}} \tag{8}$$

where $\boldsymbol{u}'' = \boldsymbol{u} - \boldsymbol{U}$ is the Favré-perturbation of velocity.

As with the Favré-averaged velocities, we first do $\boldsymbol{R} \leftarrow \overline{\rho}\boldsymbol{R}$:

```
Rxx(:,:,:) = rhomean(:,:,:) * Rxx(:,:,:)
Rxy(:,:,:) = rhomean(:,:,:) * Rxy(:,:,:)
Rxz(:,:,:) = rhomean(:,:,:) * Rxz(:,:,:)

Ryy(:,:,:) = rhomean(:,:,:) * Ryy(:,:,:)
Ryz(:,:,:) = rhomean(:,:,:) * Ryz(:,:,:)

Rzz(:,:,:) = rhomean(:,:,:) * Rzz(:,:,:)
```

N.B. we are exploiting the symmetry of the Reynolds stress tensor.

We can then update the mean $\overline{\rho {u_i}'' {u_j}''}$ and divide through by $\overline{\rho}$

```
!! Rxx, Rxy, Rxz
ta1(:,:,:) = rho1(:,:,:) * (ux1(:,:,:) - ufmean(:,:,:)) * (ux1(:,:,:) - ufmean(:,:,:))
call fine_to_coarseS(1, ta1, tmean)
Rxx(:,:,:) = (Rxx(:,:,:) + tmean(:,:,:)) / rhomean(:,:,:)

ta1(:,:,:) = rho1(:,:,:) * (ux1(:,:,:) - ufmean(:,:,:)) * (uy1(:,:,:) - vfmean(:,:,:))
call fine_to_coarseS(1, ta1, tmean)
Rxy(:,:,:) = (Rxy(:,:,:) + tmean(:,:,:)) / rhomean(:,:,:)
```

```
ta1(:,:,:) = rho1(:,:,:) * (ux1(:,:,:) - ufmean(:,:,:)) * (uz1(:,:,:) - wfmean(:,:,:))
call fine_to_coarseS(1, ta1, tmean)
Rxz(:,:,:) = (Rxz(:,:,:) + tmean(:,:,:)) / rhomean(:,:,:)

!! Ryy, Ryz
ta1(:,:,:) = rho1(:,:,:) * (uy1(:,:,:) - vfmean(:,:,:)) * (uy1(:,:,:) - vfmean(:,:,:))
call fine_to_coarseS(1, ta1, tmean)
Ryy(:,:,:) = (Ryy(:,:,:) + tmean(:,:,:)) / rhomean(:,:,:)

ta1(:,:,:) = rho1(:,:,:) * (uy1(:,:,:) - vfmean(:,:,:)) * (uz1(:,:,:) - wfmean(:,:,:))
call fine_to_coarseS(1, ta1, tmean)
Ryz(:,:,:) = (Ryz(:,:,:) + tmean(:,:,:)) / rhomean(:,:,:)

!! Rzz
ta1(:,:,:) = rho1(:,:,:) * (uz1(:,:,:) - wfmean(:,:,:)) * (uz1(:,:,:) - wfmean(:,:,:))
call fine_to_coarseS(1, ta1, tmean)
Rzz(:,:,:) = (Rzz(:,:,:) + tmean(:,:,:)) / rhomean(:,:,:)
```

Currently there is no function to compute the Reynolds stresses so we define one[1]

```
SUBROUTINE calc_rstress(rho1, ux1, uy1, uz1, rhomean, ufmean, vfmean, wfmean, &
     Rxx, Rxy, Rxz, Ryy, Ryz, Rzz, ta1, tmean)

  USE param
  USE variables
  USE decomp_2d
  USE decomp_2d_io

  IMPLICIT NONE

  REAL(mytype), DIMENSION(xsize(1), xsize(2), xsize(3)), INTENT(IN) :: rho1, ux1, uy1, uz1
  REAL(mytype), DIMENSION(xszS(1), xszS(2), xszS(3)), INTENT(IN) :: rhomean, ufmean, vfmean,
  REAL(mytype), DIMENSION(xszS(1), xszS(2), xszS(3)) :: Rxx, Rxy, Rxz, Ryy, Ryz, Rzz, ta1, tm

  <<src:rstress-prep.f90>>
  <<src:rstress.f90>>
  <<src:rstress-write.f90>>
ENDSUBROUTINE calc_rstress
```

Like the STATISTIC routine, this and other postprocessing routines defined here output every isave timesteps:

```
if (mod(itime,isave)==0) then
   call decomp_2d_write_one(1, Rxx, "Rxx.dat", 1)
   call decomp_2d_write_one(1, Rxy, "Rxy.dat", 1)
   call decomp_2d_write_one(1, Rxz, "Rxz.dat", 1)

   call decomp_2d_write_one(1, Ryy, "Ryy.dat", 1)
   call decomp_2d_write_one(1, Ryz, "Ryz.dat", 1)

   call decomp_2d_write_one(1, Rzz, "Rzz.dat", 1)
endif
```

[1] A call needs to be added to incompact3d.f90

5. Energy spectra

   Obtained in x and z directions at $y = 0$ for velocity and temperature.

   Somehow Almagro has temperature spectra for case `s1` which doesn't seem to make sense - the temperature should be constant.

6. Energy dissipation rate

   The dissipation rate of turbulent kinetic energy for variable-density flows is given as

$$\begin{aligned}\overline{\rho}\,\overline{\epsilon} &= \overline{\boldsymbol{\tau}' : \boldsymbol{\nabla} \boldsymbol{u}'} \\ &= \mu \left(\frac{4}{3} \overline{\left(\boldsymbol{\nabla} \cdot \boldsymbol{u}' \right)^2} + \overline{\left(\boldsymbol{\nabla} \times \boldsymbol{u}' \right)^2} + 2 \left(\frac{\partial^2 \overline{u_i' u_j'}}{\partial x_i \partial x_j} - 2 \boldsymbol{\nabla} \cdot \overline{\boldsymbol{u}' \boldsymbol{\nabla} \cdot \boldsymbol{u}'} \right) \right)\end{aligned} \tag{9}$$

   where $\boldsymbol{u}' = \boldsymbol{u} - \overline{\boldsymbol{u}}$ is the *Reynolds*-fluctuation of velocity. **N.B.** this is the *Reynolds* dissipation rate according to Chassaing - it may also be of interest to record the *Favré* dissipation rate

$$\overline{\rho}\,\overline{\epsilon}^F = \overline{\boldsymbol{\tau} : \boldsymbol{\nabla} \boldsymbol{u}''} = 2\mu \left(\overline{\boldsymbol{s} : \boldsymbol{s}''} - \frac{1}{3} \overline{\left(\boldsymbol{\nabla} \cdot \boldsymbol{u} \right) \left(\boldsymbol{\nabla} \cdot \boldsymbol{u}'' \right)} \right) \tag{10}$$

   To compute the *Reynolds* dissipation rate, first we apply $\overline{\epsilon} \leftarrow \overline{\rho}\,\overline{\epsilon}$ and compute the *Reynolds* fluctuating velocities

```
rdiss(:,:,:) = rhomean(:,:,:) * rdiss(:,:,:)
ta1(:,:,:) = ux1(:,:,:) - umean(:,:,:)
tb1(:,:,:) = uy1(:,:,:) - vmean(:,:,:)
tc1(:,:,:) = uz1(:,:,:) - wmean(:,:,:)
```

   We can then begin computing derivatives for the *Reynolds*-fluctuating viscous stress tensor and velocity gradient

```
!! X-derivatives
CALL derx (td1,ta1,di1,sx,ffx,fsx,fwx,xsize(1),xsize(2),xsize(3),0)
CALL derx (te1,tb1,di1,sx,ffxp,fsxp,fwxp,xsize(1),xsize(2),xsize(3),1)
CALL derx (tf1,tc1,di1,sx,ffxp,fsxp,fwxp,xsize(1),xsize(2),xsize(3),1)

CALL transpose_x_to_y(ta1, ta2)
CALL transpose_x_to_y(tb1, tb2)
CALL transpose_x_to_y(tc1, tc2)

!! Y-derivatives
CALL dery (td2,ta2,di2,sy,ffyp,fsyp,fwyp,ppy,ysize(1),ysize(2),ysize(3),1)
CALL dery (te2,tb2,di2,sy,ffy,fsy,fwy,ppy,ysize(1),ysize(2),ysize(3),0)
CALL dery (tf2,tc2,di2,sy,ffyp,fsyp,fwyp,ppy,ysize(1),ysize(2),ysize(3),1)

CALL transpose_y_to_z(ta2, ta3)
CALL transpose_y_to_z(tb2, tb3)
CALL transpose_y_to_z(tc2, tc3)

!! Z-derivatives
CALL derz (td3,ta3,di3,sz,ffzp,fszp,fwzp,zsize(1),zsize(2),zsize(3),1)
CALL derz (te3,tb3,di3,sz,ffzp,fszp,fwzp,zsize(1),zsize(2),zsize(3),1)
CALL derz (tf3,tc3,di3,sz,ffz,fsz,fwz,zsize(1),zsize(2),zsize(3),0)
```

As we are now in the Z-pencil it is most efficient to then build the double-inner product $\boldsymbol{\tau}' : \boldsymbol{\nabla u}'$ backwards:

```
!! Z-pencils
!    (dudz + dwdx) * dudz
! + (dvdz + dwdy) * dvdz
! + (dwdz + dwdz) * dwdz

<<src:wder-to-z.f90>>

tc3(:,:,:) = (td3(:,:,:) + ta3(:,:,:)) * td3(:,:,:) &
     + (te3(:,:,:) + tb3(:,:,:)) * te3(:,:,:) &
     + (tf3(:,:,:) + tf3(:,:,:)) * tf3(:,:,:)
CALL transpose_z_to_y(tc3, tc2)

!! Y-pencils
!    (dudy + dvdx) * dudy
! + (dvdy + dvdy) * dvdy
! + (dwdy + dvdz) * dwdy

<<src:vder-to-y.f90>>

tb2(:,:,:) = (td2(:,:,:) + ta2(:,:,:)) * td2(:,:,:) &
     + (te2(:,:,:) + te2(:,:,:)) * te2(:,:,:) &
     + (tf2(:,:,:) + tb2(:,:,:)) * tf2(:,:,:)

tb2(:,:,:) = tb2(:,:,:) + tc2(:,:,:)
CALL transpose_y_to_x(tb2, ta1)

!! X-pencils
!    (dudx + dudx) * dudx
! + (dvdx + dudy) * dvdx
! + (dwdx + dudz) * dwdx

<<src:uder-to-x.f90>>

ta1(:,:,:) = ta1(:,:,:) &
     + (td1(:,:,:) + td1(:,:,:)) * td1(:,:,:) &
     + (te1(:,:,:) + tb1(:,:,:)) * te1(:,:,:) &
     + (tf1(:,:,:) + tc1(:,:,:)) * tf1(:,:,:)
```

Where we transport already computed derivatives up/down the pencils as required:

```
CALL transpose_x_to_y(tf1, tc2) ! dwdx, x->y

CALL transpose_y_to_z(tc2, ta3) ! dwdx, y->z
CALL transpose_y_to_z(tf2, tb3) ! dwdy, y->z
```

```
CALL transpose_x_to_y(te1, ta2) ! dvdx, x->y

CALL transpose_z_to_y(te3, tb2) ! dvdz, z->y
```

```
CALL transpose_z_to_y(td3, ta2) ! dudz, z->y

CALL transpose_y_to_x(td2, tb1) ! dudy, y->x
CALL transpose_y_to_x(ta2, tc1) ! dudz, y->x
```

All that remains is to subtract the divergence of fluctuations term, noting that $(\nabla \cdot \boldsymbol{u}')\,\boldsymbol{I} : \nabla \boldsymbol{u}' = (\nabla \cdot \boldsymbol{u}')^2$

```fortran
CALL transpose_z_to_y(tf3, tf2)

CALL transpose_y_to_x(te2, te1)
CALL transpose_y_to_x(tf2, tf1)

ta1(:,:,:) = ta1(:,:,:) &
     - (2._mytype / 3._mytype) * (td1(:,:,:) + te1(:,:,:) + tf1(:,:,:))**2
```

This can all then be wrapped in a subroutine

```fortran
!! -*- mode: f90 -*-
SUBROUTINE calc_rdiss(ux1, uy1, uz1, rhomean, umean, vmean, wmean, rdiss, &
     ta1, tb1, tc1, td1, te1, tf1, di1, &
     ta2, tb2, tc2, td2, te2, tf2, di2, &
     ta3, tb3, tc3, td3, te3, tf3, di3)

  USE param
  USE variables
  USE decomp_2d
  USE decomp_2d_io

  IMPLICIT NONE

  REAL(mytype), DIMENSION(xsize(1), xsize(2), xsize(3)), INTENT(IN) :: ux1, uy1, uz1, &
       rhomean, umean, vmean, wmean
  REAL(mytype), DIMENSION(xsize(1), xsize(2), xsize(3)) :: ta1, tb1, tc1, td1, te1, tf1, di1
  REAL(mytype), DIMENSION(ysize(1), ysize(2), ysize(3)) :: ta2, tb2, tc2, td2, te2, tf2, di2
  REAL(mytype), DIMENSION(zsize(1), zsize(2), zsize(3)) :: ta3, tb3, tc3, td3, te3, tf3, di3
  REAL(mytype), DIMENSION(xsize(1), xsize(2), xsize(3)) :: rdiss

  INTEGER :: i, j, k
  REAL(mytype) :: rint
  LOGICAL :: file_exists

  <<src:rdiss-prep.f90>>
  <<src:rdiss-grad.f90>>
  <<src:rdiss-2d:gradu.f90>>
  <<src:rdiss-divu2.f90>>

  !! Update dissipation
  rdiss(:,:,:) = (rdiss(:,:,:) + xnu * ta1(:,:,:)) / rhomean(:,:,:)

  !! Save field
  CALL decomp_2d_write_one(1, rdiss, "dissipation.dat", 1)

  <<src:rdiss-avg.f90>>
ENDSUBROUTINE calc_rdiss
```

Apart from the 3D field, we also want to know the vertical integral of the dissipation rate:

```fortran
  !! Average in x
  ta1(:,:,:) = rdiss(:,:,:)
  do i = 2, xsize(1)
```

```
      ta1(1,:,:) = ta1(1,:,:) + ta1(i,:,:)
   enddo
   ta1(1,:,:) = ta1(1,:,:) / real(xsize(1), mytype)
   do i = 2, xsize(1)
      ta1(i,:,:) = ta1(1,:,:)
   enddo

   call transpose_x_to_y(ta1, ta2)
   call transpose_y_to_z(ta2, ta3)

   do k = 2, zsize(3)
      ta3(:,:,1) = ta3(:,:,1) + ta3(:,:,k)
   enddo
   ta3(:,:,1) = ta3(:,:,1) / real(zsize(3), mytype)
   do k = 2, zsize(3)
      ta3(:,:,k) = ta3(:,:,1)
   enddo

   call transpose_z_to_y(ta3, ta2)

   rint = 0._mytype
   do j = 1, ysize(2)
      rint = rint + ta2(1,j,1)
   enddo

   if (nrank.eq.0) then
      inquire(FILE="dissipation.log", EXIST=file_exists)
      if (file_exists) then
         open(1234, FILE="dissipation.log", STATUS="old", ACTION="write", POSITION="append")
      else
         open(1234, FILE="dissipation.log", STATUS="new", ACTION="write")
         write(1234, *) "ITIME dissipation"
      endif
      write(1234, 9123) itime, rint
9123  format(" ", I8, E15.7)
      close(1234)
   endif
```

7. Mixing layer thickness

   The momentum thickness, defined as

$$\delta_m = \frac{1}{\rho_0 \Delta u^2} \int_{-\infty}^{\infty} \overline{\rho} \left(\frac{1}{2}\Delta u - U \right) \left(\frac{1}{2}\Delta u + U \right) dy \tag{11}$$

   and the vorticity thickness

$$\delta_\omega = \frac{\Delta u}{|\partial U / \partial y|_{max}} \tag{12}$$

   are recorded as functions of time.

   To compute the momentum thickness we first average in the stream and spanwise directions

```
!! First compute the 3D field
if (ilmn.ne.0) then
   ta1(:,:,:) = rhomean(:,:,:) * (0.5_mytype * du - ufmean(:,:,:) / REAL(itime, mytype)) &
```

```
       * (0.5_mytype * du + ufmean(:,:,:) / REAL(itime, mytype))
else
   ta1(:,:,:) = (0.5_mytype * du - ufmean(:,:,:) / REAL(itime, mytype)) &
       * (0.5_mytype * du + ufmean(:,:,:) / REAL(itime, mytype))
endif

!! Now average in X
DO i = 2, xsize(1)
   ta1(1,:,:) = ta1(1,:,:) + ta1(i,:,:)
ENDDO
ta1(1,:,:) = ta1(1,:,:) / REAL(xsize(1), mytype)
DO i = 2, xsize(1)
   ta1(i,:,:) = ta1(1,:,:)
ENDDO

!! Get to Z
CALL transpose_x_to_y(ta1, ta2)
CALL transpose_y_to_z(ta2, ta3)

!! Now average in Z
DO k = 2, zsize(3)
   ta3(:,:,1) = ta3(:,:,1) + ta3(:,:,k)
ENDDO
ta3(:,:,1) = ta3(:,:,1) / REAL(zsize(3), mytype)
DO k = 2, zsize(3)
   ta3(:,:,k) = ta3(:,:,1)
ENDDO
```

ending up in the z-pencils. We then integrate in y and divide by $\rho_0 \Delta u^2$

```
!! First, get from Z to Y
CALL transpose_z_to_y(ta3, ta2)

!! Now integrate
deltam = 0._mytype

DO j = 1, ysize(2) - 1
   deltam = deltam + (ta2(1, j, 1) + ta2(1, j + 1, 1)) * (dy / 2._mytype)
ENDDO
deltam = deltam / (((dens1 + dens2) / 2._mytype) * du**2)
```

Computation of the vorticity thickness is relatively straight-forward - first compute the gradient of U wrt y:

```
!! Get Favre-avg u to Y (averaged in X and Z)
ta1(1,:,:) = ufmean(1,:,:)
DO i = 2, xsize(1)
   ta1(1,:,:) = ta1(1,:,:) + ufmean(i,:,:)
ENDDO
ta1(1,:,:) = ta1(1,:,:) / REAL(xsize(1), mytype)
DO i = 2, xsize(1)
   ta1(i,:,:) = ta1(1,:,:)
ENDDO

CALL transpose_x_to_y(ta1, tb2)
CALL transpose_y_to_z(tb2, ta3)
```

```
DO k = 2, zsize(3)
   ta3(:,:,1) = ta3(:,:,1) + ta3(:,:,k)
ENDDO
ta3(1,:,:) = ta3(1,:,:) / REAL(zsize(3), mytype)
DO k = 2, zsize(3)
ta3(:,:,k) = ta3(:,:,1)
ENDDO

CALL transpose_z_to_y(ta3, tb2)

!! Compute dUdy
call dery (ta2,tb2,di2,sy,ffyp,fsyp,fwyp,ppy,ysize(1),ysize(2),ysize(3),1)
```

followed by identification of the maximum and definition of the vorticity thickness

```
!! Find (local) maximum of |dUdy|
maxloc = MAX(MAXVAL(ta2), -MINVAL(ta2)) / REAL(itime, mytype)

!! Find the (global) maximum of |dUdy|
CALL MPI_ALLREDUCE(maxloc, maxglob, 1, real_type, MPI_MAX, MPI_COMM_WORLD, ierr)

!! Compute deltaw
deltaw = du / maxglob
```

We can then build this into a routine to calculate (and print) the mixing layer thickness

```
SUBROUTINE calc_thickness(rhomean, ufmean, ta1, ta2, tb2, di2, ta3)

  USE MPI
  USE param
  USE variables
  USE decomp_2d

  IMPLICIT NONE

  REAL(mytype), DIMENSION(xsize(1), xsize(2), xsize(3)), INTENT(IN) :: rhomean, ufmean
  REAL(mytype), DIMENSION(xsize(1), xsize(2), xsize(3)) :: ta1
  REAL(mytype), DIMENSION(ysize(1), ysize(2), ysize(3)) :: ta2, tb2, di2
  REAL(mytype), DIMENSION(zsize(1), zsize(2), zsize(3)) :: ta3

  REAL(mytype) :: du
  REAL(mytype) :: maxloc, maxglob
  REAL(mytype) :: deltam, deltaw

  INTEGER :: i, j, k
  INTEGER :: ierr
  LOGICAL :: file_exists

  du = ABS(u1 - u2)

  <<src:momthick-homoavg.f90>>
  <<src:momthick-yint.f90>>
  <<src:vortthick-dudy.f90>>
  <<src:vortthick.f90>>
  !! Write out
  IF (nrank.EQ.0) THEN
     INQUIRE(FILE="thickness.log", EXIST=file_exists)
     IF (file_exists) THEN
        OPEN(30, FILE="thickness.log", STATUS="old", ACTION="write", POSITION="append")
     ELSE
        OPEN(30, FILE="thickness.log", STATUS="new", ACTION="write")
        WRITE(30, *) "ITIME deltam deltaw"
     ENDIF
     WRITE(30, 9000) itime, deltam, deltaw
9000 FORMAT(" ", I8, E15.7, E15.7)
     CLOSE(30)
  ENDIF

ENDSUBROUTINE calc_thickness
```

8. Calling postprocessing

   Finally, we must call these postprocessing routines from the main program:

```
if (ilmn.ne.0) then
   call calc_rstress(rho1, ux1, uy1, uz1, rhomean, ufmean, vfmean, wfmean, &
        Rxx, Rxy, Rxz, Ryy, Ryz, Rzz, ta1, tmean)
else
   call calc_rstress(rho1, ux1, uy1, uz1, rhomean, umean, vmean, wmean, &
        Rxx, Rxy, Rxz, Ryy, Ryz, Rzz, ta1, tmean)
endif
call calc_rdiss(ux1, uy1, uz1, rhomean, umean, vmean, wmean, rdiss, &
     ta1, tb1, tc1, td1, te1, tf1, di1, &
     ta2, tb2, tc2, td2, te2, tf2, di2, &
     ta3, tb3, tc3, td3, te3, tf3, di3)
if (ilmn.ne.0) then
   call calc_thickness(rhomean, ufmean, ta1, ta2, tb2, di2, ta3)
else
   call calc_thickness(rhomean, umean, ta1, ta2, tb2, di2, ta3)
endif
```

2.2.11 incompact3d.prm

An `incompact3d.prm-s*` is generated automatically for each case by tangling this file. For reference, that for case `s1` is shown.

```
#
# INCOMPACT 3D Flow parameters
#
461.0    #xlx  # Lx (Size of the box in x-direction)
368.0    #yly  # Ly (Size of the box in y-direction)
173.0    #zlz  # Lz (Size of the box in z-direction)
160. #re     # Reynolds number
0.7  #pr     # Prandtl number
1.   #sc     # Schmidt number (if passive scalar)
0.     #frx    # Froude number in x direction (frx = 0 gives no gravity)
0.     #fry    # Froude number in y direction
0.     #frz    # Froude number in z direction
-0.5     #u1     # u1 (max velocity) (for inflow condition)
```

```
0.5      #u2    # u2 (min velocity) (for inflow condition)
1.0     #dens1 # dens1
1.0     #dens2 # dens2
0.1    #noise# Turbulence intensity (1=100%) !! Initial condition
0.05    #noise1# Turbulence intensity (1=100%) !! Inflow condition
0.21875  #dt    # Time step
#
# INCOMPACT3D Flow configuration
#
0       #nclx    # nclx (BC)
1       #ncly    # ncly (BC)
0       #nclz    # nclz (BC)
4       #itype   # Type of flow  (1: Constant flowfield, 4: Mixing layer, 6: Taylor-Green)
1       #iin     # Inflow condition (1: classic, 2: turbinit)
1      #ifirst # First iteration
3200  #ilast  # Last iteration
1       #nscheme# Temporal scheme (1:AB2, 2: RK3, 3:RK4, 4:AB3)
0       #istret # y mesh refinement (0:no, 1:center, 2:both sides, 3:bottom)
0.3     #beta    # Refinement parameter (beta)
0       #ilmn    # (0:incompressible, 1: Low Mach Number)
0       #isolvetemp# (0: solve for density, 1: solve for temperature)
0       #imulticomponent
0       #nrhoscheme# Density scheme (how to approximate drhodt^{k+1})
0       #ivarcoeff# (0:constant-coefficient Poisson equation, 1:variable-coefficient Poisson equati
1       #npoisscheme# How to compute rho0 in var-coeff poisson equation? (0: rho0 = MIN(rho), Anyth
1.0e-14 #tol     # Tolerance for Poisson equation
1       #iskew   # (0:urotu, 1:skew, for the convective terms)
1       #iprops # (0: constant properties, 1: variable properties)
0       #iscalar# (0: no scalar, 1:scalar)
#
# INCOMPACT 3D File parameters
#
0            #ilit      # Read initial flow field ?
640        #isave     # Frequency for writing backup file
32         #imodulo  # Frequency for visualization for VISU_INSTA
#
# INCOMPACT 3D Body  old school
#
0      #ivirt# IBM? (1: old school, 2: Lagrangian Poly)
5.     #cex   # X-centre position of the solid body
6.     #cey   # Y-centre position of the solid body
0.     #cez   # Z-centre position of the solid body
0.5    #re    # Radius of the solid body
#
```

Listing 33: Case s1 incompact3d.prm

3 Validation

3.1 Of the ∇q solver

4 Results

5 Log

5.1 2018-12-12 Wednesday

Added code to collect Reynolds-averaged density and Favré-averaged velocity