DNS Code User Guide

Ryan Kelly

Last Updated: October 6, 2023

This code is a pseudo-spectral direct numerical simulation code to solve the full 3-D incompressible Navier-Stokes equations. The solver is based on **Kim, J., Moin, P., & Moser, R. (1987). Turbulence statistics in fully developed channel flow at low Reynolds number. Journal of Fluid Mechanics, 177, 133-166.**

The external force field is based on **Goldstein, D., Handler, R., Sirovich, L. (1993). Modeling a No-Slip Flow Boundary with an External Force Field. Journal of Computational Physics, Volume 105, Issue 2, Pages 354-366.**

The polymer coupling uses a passive scalar advection-diffusion equation with a source, with polymer effects added in as a body force term. The polymer conformation tensor is evolved based on the following equation:

$$\frac{\mathrm{D}C_{ij}}{\mathrm{D}t} = C_{ik}\frac{\partial u_j}{\partial x_k} + C_{kj}\frac{\partial u_i}{\partial x_k} - \frac{1}{\lambda}[f(C_{kk})C_{ij} - \delta_{ij}] + \alpha_p \nabla^2 C_{ij}$$

where λ is the polymer relaxation time, $f(C_{kk})$ is the Peterlin function, and α_p is a numerical diffusion constant. The conformation tensor is then used to calculate the polymer stress, and the resulting body force is subsequently computed as the divergence of polymer stress. The code implementation is very similar to that in **R. Sureshkumar, A.N. Beris, and R.A. Handler, "Direct numerical simulation of the turbulent channel flow of a polymer solution" (1997)**. For the targeted polymer, however, we use an exponential model based on experimental results for PEO in **B. Nsom and N. Latrache, "Measurement of Drag Reduction in Dilute Polymer Solution Using Triboelectric Effect" (2018)**.

The particle tracking method is a point-particle approximation using a modified form of the Maxey-Riley equations **M. R. Maxey and J. J. Riley, "Equations of motion for a small rigid sphere in a nonuniform flow," The Physics of Fluids 26, 883–889 (1983)**

1 Downloading the Code

First thing's first: you'll need to download the code in order to use it. The code is available on Github at `https://github.com/ryankellybp11/Polymer_DNS`. You can navigate to the website and download the code locally or you can clone the repository to a local directory via the command

```
git clone https://github.com/ryankellybp11/Polymer_DNS.git
```

If you want to work on your own version in Github, you can fork the repository (copy it to your own account) or create a branch to work within the same project. The latter allows you to create pull requests and flag issues with the code as you find them. Inside the repository, you'll find several folders, each containing the main code, but configured a little differently. `Polymer_DNS-main` contains the main code with configurations from the last run I did before uploading the file. The other directories contain the same code but prepared for the example cases in Sections 6-9.

2 Code Architecture

2.1 Directories

All files (binary executables as well as the source code) are contained in a master folder, with the only optional dependency being Tecplot (see Section 5 for information on how to install Tecplot). Visually, it is helpful to see how all the codes are organized, as shown in the figure below.

```
.
├── code                          # Contains source code, compiled code, binariy files, and geometry file
│   ├── bin                       # Contains all binary files
│   │   ├── dns                   # Main executable - runs DNS
│   │   ├── geometry              # Contains geometry generator and geometry file
│   │   │   ├── Geom              # Compiled geometry generator
│   │   │   └── grid_size.mod     # Binary module for grid size data
│   │   ├── grid_size.mod         # Binary module for grid size data
│   └── src                       # Contains all the source files
│       ├── dns                   # Contains DNS source files
│       │   ├── dns.f90           # Main DNS source file (with polymer)
│       │   ├── dns_no_poly.f90   # Main DNS source file (without polymer)
│       │   ├── ffts.f90          # Useful fft functions for dns.f90
│       │   ├── grid_size.f90     # Module with all grid size data
│       │   ├── init_flow.f90     # Defines flow type (Channel, Couette, BL, etc.)
│       │   ├── part_track.f90    # Particle tracking integrator
│       │   └── polymer_routines.f90 # Additional subroutines for polymer effects
│       ├── geometry              # Contains geometry generator source code
│       │   └── Geom.f90          # Geometry generator source code
│       └── preconfigure.py       # preconfigure tool: compiles all the necessary files. Run with Python >=2.7.13
├── setup                         # Contains all the setup files that are inputs to the DNS.
│   ├── dns.config                # Main DNS input file
│   ├── vort.config               # Contains input information for generating vortices (soon to be vortex ring as well, but as of now, that's in dns.config)
│   ├── restart                   # Restart file (outputs/last-restart file of previous simulation)
│   ├── c-restart                 # Scalar/polymer restart file
│   └── particles                 # Particle setup files
│       ├── generator.f90         # Generates random particles in flow domain
│       ├── particles.dat         # Contains particle trajectory info (ASCII)
│       └── pgen                  # Binary particle generator file
├── outputs                       # Contails all the outputs produced by the DNS
│   ├── flowfield                 # Contains tecplot-readable binary data files of the flow field
│   │   ├── grid.plt              # Grid file
│   │   ├── time-00000.plt        # Flow field at time step 0 (in Tecplot binary format)
│   │   └── ...                   # etc.
│   ├── particles                 # Contains particle data
│   │   ├── part-t-00000.dat      # Particles at time step 0 (in Tecplot ASCII format)
│   │   └── ...                   # etc.
│   ├── last-restart              # Latest restart file. Do `mv outputs/last-restart setup/restart` if you want to restart latest simulation.
│   ├── c-last-restart            # Latest polymer restart file. `mv outputs/c-last-restart setup/c-restart` if you want to restart latest simulation.
│   ├── err                       # Runtime errors will appear here
│   ├── log                       # Runtime outputs
│   └── KE,enstrophy,...          # Channel flow output files for comparing different flow variables
├── jobcode                       # File for submitting a job to TACC. Usage: `sbatch jobcode`
├── clean                         # File for cleaning outputs directory. Usage: `. clean'
├── run                           # File for running the code locally (or on a work node). Usage: `. run'
├── hushrun                       # Same as `run' but runs the code in the background
└── err_comp.txt                  # Output file for any compilation errors (empty if no errors)
```

2.2 Communication

The directory names are intuitive, so the way the code operates shouldn't be surprising. The source code is stored in the `./code/src/` directory, and these are the files written to actually do the work. For normal runs, these files are not often changed, and the code is designed to operate primarily by interfacing with the setup files inside the `./setup/` directory. The files inside this directory are quickly changed and provide a host of options for different cases you can run. Once the appropriate setup parameters are chosen, the code compiles into executables inside the `./code/bin/` directory. When you run the main executable, any outputs you have will be placed in the `./outputs/` directory, appropriately categorized by type.

3 Setup Files

All setup parameters are contained inside the `dns.config` file and `vort.config`. Currently, `vort.config` is only half-used for the Gaussian vortex cases, but it will eventually be used for vortex ring parameters as well. Inside `dns.config`, the parameters are divided into different sections within the file for ease of use. Note that if you add, remove, or rearrange any of the lines in this file, you will need to appropriately update `preconfigure.py`, `Geom.f90`, `generator.f90`, and `dns.f90` in the subroutine `setstuf`. As a note, the code is set up to have dimensional variables, and the dimensions are centimeters and seconds. You have quite a bit of control over the what the code calculates solely from this file. There is a brief description of each variable inside the file, but for thoroughness, I will list the variables here with a few helpful comments:

3.1 Main Solver/Grid/Time Parameters

This is a configuration file where most flow conditions are specified. From this file, you can change the number of time steps, when to print, the domain size, etc.

- `NSTEPS` – The number of time steps the code will run
- `IPRNFRQ` – The printing frequency: how many time steps between each time the flowfield is printed
- `DT` – The time step. We have a CFL restriction based on the grid size and time step, so the finer the grid is, the smaller the time step will need to be
- `gain` – Gain for the integral term in the immersed boundary control scheme
- `ugain` – Gain for the integral term in the immersed boundary control scheme. Neither this nor gain should need to be changed
- `THETA` – Implicitness factor used in the solver. You probably won't ever change this
- `NY1` – Number of grid cells in the y-direction
- `NZ1` – Number of grid cells in the z-direction
- `NX1` – Number of grid cells in the x-direction
- `BFTAIL` – Number of grid cells in x-direction for the buffer zone. This should be 0 unless you're using a buffer region to recycle flow
- `YL` – Length of computational domain in the wall-normal direction
- `ZL` – Length of computational domain in the span-wise direction
- `XL` – Length of computational domain in the stream-wise direction
- `RE` – "Code Reynolds number" - This is actually $1/\nu$ where ν is the kinematic viscosity of the fluid (solvent if there is polymer)

- `XSTART` – x-location along a flat plate that corresponds to the boundary layer profile created by the buffer zone (only relevant for BL flow)
- `Uinf` – Free-stream velocity (only relevant for Couette and BL flow)
- `KWALL` – y-index corresponding to the nominal wall location (only relevant for BL flow)
- `Perturbstayingtime` – The cell type 4 acts like a solid for this many time steps, and afterwards, it is turned off
- `R_tau` – This is Re_τ for channel flow, and it is ignored unless `flow_select` is 1.
- `Omearth` – Rotation rate of the Earth (rad/s) used in Coriolis force calculation. Typically set to 0
- `rotang` – Rotation angle of the Earth (degrees) used in Coriolis force calculation. Typically set to 90°
- `Grav` – Gravitational acceleration (cm/s^2) used for buoyancy calculation of scalar. This should probably be merged with the gravity used for particles, but since we typically set it to 0, it's not a high-priority right now
- `virtual` – Background virtual temperature used for buoyancy calculation of scalar. Typically set to 4831

3.2 Flags and Switches

- `IRSTRT` – A switch for restarting a run. 0 means the flow is initialized in the code, 1 means the initial flow state is read from the `restart` file inside the `setup/` directory, and 2 allows you to restart but with a different grid resolution. (I have never used 2, so I'm not sure how well that works)
- `CRSTRT` – A switch for restarting a run with polymer. 0 means the polymer is initialized in the code, 1 means the initial polymer (conformation tensor) state is read from the `c-restart` file inside the `setup/` directory. Currently, there's no option to restart with different grid resolution.
- `print3d` – A switch for how the flowfield is printed. 0 means it is *not* printed, 1 means the output file is ASCII, 2 means the code writes the ASCII file then converts to Tecplot binary, and 3 means the code writes directly to Tecplot binary. 3 is the best option if you're using Tecplot to visualize as it's much faster, but it requires you to install Tecplot and its dependencies. For GMU's post-processing, I made it where you can set this variable to -1, and it will create a directory called `GMU_out/` and print the typical outputs there. For simplicity, I made all printing frequencies equal to `IPRNFRQ`.
- `geomtype` – This sets the geometry situation of the flow. 0 means there are no objects in the domain, 2 is a set of 2D bricks, and 3 is a 3D brick in the flow (to trip the flow to turbulence). Earlier versions of the code have used this for generating roughness elements on a flat plate, but I don't have that code. Other shapes and functionality can be added with different numbers for the switch.
- `readvdes` – Switch for whether or not to add suction along the top boundary. This is necessary to satisfy continuity in a straight domain for a boundary layer flow. 1 is on, 0 is off
- `particle_flag` – Switch for the type of particle. At the moment, 0 is a tracer particle and 1 is a passive particle. In future applications, there may be reason to include particles with 2- or 4-way coupling. For a moving scalar source, the scalar is emitted from particle locations, so a value of -1 defines a stationary particle for a fixed scalar source.
- `flow_select` – Switch for selecting the type of flow. Each case currently listed has appropriate boundary conditions automatically applied inside the code. By default, any new flow configurations will have shear-free BCs on top and bottom (and of course, periodic BCs in x and z). If you need to change this, see section **??**. Here are the current options:

0: **Still fluid**. ($\boldsymbol{u} = 0$ everywhere)

1: **Channel flow**. The velocity is set by Re_τ, which is used to compute a constant pressure gradient term. In the code, this constant pressure gradient is treated like a body force and added to the $\boldsymbol{u} \times \boldsymbol{\omega}$ term inside the subroutine `vcw3d`. Works the same in 2D and 3D.

2: **Couette flow**. The upper-wall velocity is set by `Uinf`. The code automatically handles the appropriate boundary conditions. Works the same in 2D and 3D.

3: **Modified Couette flow**. Superimposed Couette flow in the x- and z-directions, so the result is a diagonal flow in x and z, linearly increasing in y. The code automatically sets the appropriate boundary conditions based on `Uinf`. There is no option to set different x and z velocities.

4: **Blasius boundary layer**. Uses a solver to generate the solution to the Blasius equation for a given `Uinf`. Note that this only sets a Blasius profile (extruded in the z-direction) as the initial condition, and there is no way to sustain it except with the buffer region. *It is a known issue at the time of this writing that the initial Blasius solver has a bug causing it to run exceptionally slowly (or perhaps not ever converge on a solution at all).*

5: **Vortex Ring**. Generated by an impulsive body force, characterized by the parameters in section 3.5.

1X: **Gaussian Vortex**. A vortex with Gaussian vorticity (ω_x) is set as an initial condition. The vortex decays with time as there are no body forces sustaining it. It may be a single Gaussian vortex or a counter-rotating vortex pair, characterized by parameters in section 3.6. The Gaussian vortex can be superimposed with any other flow field listed above (theoretically). Currently, it's only programmed to be superimposed on still fluid (10), channel flow (11), or Couette flow (12). the first 1 denotes a Gaussian vortex, and the X denotes which flow is superimposed.

- `itarget` – Flag for 'targeting'. Targeting means that the polymer only acts where the scalar is present, so 1 means polymer effects are calculated based on the scalar variable, and 0 means polymer effects are calculated everywhere (polymer ocean). However, if `ipolyflag` = 0, no polymer effects are calculated.
- `ipeter` – Flag for polymer model. 0 means the polymer forcing is set to 1, 1 means the Peterlin function is used to calculate forcing. It is recommended to use the Peterlin function since it is generally more accurate.
- `ipolyflag` – Flag to include polymer effects. 1 = on, 0 = off
- `scl_flag` – Flag that controls whether or not the scalar evolution is computed and printed. 0 is no scalar, 1 is an initial Gaussian distribution of scalar, and 2 is a moving scalar source. If particles are simulated, the scalar source is attached to each particle, but this may be changed in a future update.

3.3 Particle Parameters

- `npart` – Number of particles. If this number is 0, the particle tracking routine is skipped altogether.
- `new_part` – Used only by the configuration code, `preconfigure.py`. Use this to specify whether/how you want to generate new random particles. The following are calculated based on the program coded in `setup/particles/generator.f90`:

  0: No new particles are generated. Any previously saved particle initial conditions are left untouched for repeated use.

  1: Generate `npart` particles distributed uniformly randomly throughout the domain.

  2: Generates `npart` particles distributed uniformly randomly in a spherical volume. Typically used with vortex ring case.

  3: Generates `npart` particles distributed uniformly around a circle in the y-z plane. Typically used with vortex ring case.

4: Generates `npart` particles distributed uniformly randomly on a particular y-z plane. Typically used with vortex ring case.

5: Generates `npart` particles distributed uniformly randomly near the walls. How near the walls is hard-coded.

6: Generates `npart` particles distributed uniformly randomly in a streamwise cylinder. Typically used with vortex ring case.

7: Generates `npart` particles in horizontal lines on a given y-z plane. Typically used with Gaussian vortex case.

10: Generates particles to be released from the same spot. Currently does not work, but would be used to steadily release tracers/bubbles to recreate the wire bubble experiments.

11: Generate `npart` particles distributed on a uniform grid throughout the domain. Currently, this only works for `npart`= 2^{16}, but it may be generalized in a later update.

- `ratio` – The density ratio of the particle to the fluid (solvent if polymer is active)
- `a` – Particle radius (cm)
- `C_mu` – This is a viscosity coefficient related to surfactants. For most drag models, this is just 1
- `CD_switch` – Chooses how the particle drag is modeled. 0 = no drag, 1 = Bubble with surfactant (uses C_μ), 2 = Sphere correlation from Magnaudet et al. 1995, 3 = Curve fit (I used a curve-fitting tool to fit data from Magnaudet et al. 1995 using my own formulation. This was more of a curiosity and, while not too different from 2, probably shouldn't be used)

3.4 Scalar/Polymer Parameters

- `zlmax` – Maximum polymer length
- `tpoly` – Polymer relaxation time
- `alpha_poly` – In the new polymer model, this is a value equivalent to $\frac{\partial \nu_s}{\partial \gamma}$ where ν_s is the solution viscosity, and γ is the polymer concentration. This is a physical parameter which varies for each particular polymer species. For PEO in water, use $\alpha_p = 0.002418$.
- `DIFF` – Schmidt number of scalar
- `DIFFPOLY` – Schmidt number of artificial polymer diffusion
- `DELTAT` – Maximum concentration of the scalar, used when `scl_flag` is 1 to set the distribution.
- `C11` – Initial condition for C_{11}, which represents the stretching of the polymer chains in the x-direction. Typically unstretched, which is 1.
- `C22` – Initial condition for C_{22}, which represents the stretching of the polymer chains in the y-direction. Typically unstretched, which is 1.
- `C33` – Initial condition for C_{33}, which represents the stretching of the polymer chains in the z-direction. Typically unstretched, which is 1.
- `qbeta` – Used only for the polymer ocean case (ipolyflag = 1, itarget = 0), to set $\beta = \nu_0/\nu_s$ as a constant. Keep this ≥ 0.9.
- `xshift,yshift,zshift` – Directional shifts of the scalar centroid. The scalar is defined as a 3D Gaussian 'blob' with its center at the center of the domain, shifted by these variables
- `sigmax,sigmay,sigmaz` – Standard deviation of the 3D Gaussian in each direction

- `polyrate` – The emission rate of scalar source. It is programmed to have units of PPM/s
- `mpoly` – The total mass of polymer allowed to enter the domain from the source. Only applies if `scl_flag` is 2. For best results, set `src_stop` to -1.
- `src_start` – Initial time step number when the polymer source starts, as well as when the polymer force is activated. Therefore, if running a polymer ocean, the effects will only take place if `it` > `src_start`
- `src_stop` – Final time step number when the polymer source stops emitting more scalar. Does not affect polymer forces. If `src_stop` < `src_start`, it is automatically made equal to `nsteps` to denote that it releases for the entire simulation (not recommended for long simulations).

3.5 Body Force Parameters

- `ampbdy1,ampbdy2,c11amp` – Body force terms for generating a vortex ring. `ampbdy1` should be on the order of 1000 to generate a nice vortex ring. Note: if using the immersed boundary vortex ring (`flow_select` = 5), only ampbdy1 will be read, and it will be stored in the code as `bdyfx`
- `tfrac` – Duration of the body force as a fraction of the total run time. 0.1 means the body force acts for 10% of the run time and then turns off. This may be changed in a future update.
- `forbeta` – An extra term used to compute the appropriate forcing to generate a vortex ring
- `xc,yc,zc` – x-, y-, and z-location of force centroid
- `L,rad` – Length and radius of cylindrical force region

3.6 Vortex Parameters

These are found inside the `vort.config` file. Eventually the body force parameters will also be defined in that file, but they are ignored for now. For now, it's only the following:

- `vortGamma,vortSigma` – Circulation of the vortex, and standard deviation of vorticity profile
- `vortY,vortZ` – Physical y and z-coordinates of the vortex center (it is uniform in x)
- `vNum` – Number of vortices (currently only 1 or 2 supported but it could technically be extended to an arbitrary array of vortices later)
- `vortSpace` – Spacing between vortices (ignored if `vNum` = 1)

3.7 `Particles/`

This is a directory that contains code to generate random particles (`generator.f90`), the binary file to execute (`pgen`), and the initial particle locations (`particles.dat`). In some cases, it is convenient to manually specify a few particle locations. To do this, set `new_part` equal to 0 and edit `particle.dat` directly. There is a header line to identify what the numbers are interpreted as in the code, and it should not be removed.

4 Source Code Files

The following codes (barring `grid_size.f90`) are only separated for organization. When compiled, all the files are included together, but I find it easier as a user to modulize the code based on functionality. This makes it easier to work on one aspect of the code with less of a chance to mess something up elsewhere[1].

[1] but it can still happen!

4.1 Main Code: `dns.f90`

This file is the largest by far, and it contains the main program and many subroutines that are used in the solver. It is far too complicated to go into much detail, so I'm just going to highlight some of the key features. Alternatively, there is a separate version of this code in `dns_no_poly.f90` which is exactly the same except scalar and polymer terms are removed, leaving only the fluid solver and particle tracking. This has to be set inside `preconfigure.py` and compiled separately. See section 5.2 for more details.

4.1.1 Program: `threed`

This is what you should look at if you want a general feel for how the code works. This solves the 3D incompressible Navier-Stokes equations for a given number of time steps. The details are in the subroutine calls, but generally it works as follows.

1. Read in input variables using subroutine `setstuf`
2. Initialize some solver stuff (Green's function solutions)
3. Initialize forcing terms for immersed boundary method. This is also where the initial flow parameters are computed from `init_flow.f90`.
4. Initialize the flow. This runs through pretty much all the same subroutines as the main loop to get the initial conditions of the flow. If the run is a restart, then this simply reads in the restart file(s). This is a special routine to get the explicit time integrator started.
5. Enter the main loop. This is a loop which explicitly steps through time and computes the flowfield at each step
6. Compute linear terms of the conformation tensor and scalar equations (in spectral space)
7. Compute linear terms of velocity and vorticity equations (in spectral space)
8. Calculate conformation tensor derivatives and velocity gradients which will be used to compute nonlinear conformation tensor and scalar terms (and used in particle tracking)
9. Transform all data into y-physical space
10. Calculate nonlinear terms in physical space using `vcw3d`. This is where immersed boundary forces are applied, and where the particle tracking integration happens.
11. Transform data back into y-spectral space.
12. Add polymer, Coriolis, and thermal forces to the momentum equation as body forces
13. Use spectral smoothing to dampen Gibbs phenomenon
14. Calculate final solution values
15. Write restart files every `IPRNFRQ` time steps
16. (After main loop is finished) Compute statistics and end program

4.1.2 Subroutine: `vcw3d`

This subroutine is where the code spends most of its time, so it's worth looking a little at what's going on in here. At the moment, this subroutine is nominally parallelized with OpenMP, although not very efficiently. The subroutine computes the nonlinear terms plane-by-plane in the y-direction, and OpenMP assigns one thread per plane to give a modest speed-up. The y-physical variables are read into new variables within each y-plane, and these new variables are transformed to physical space in x and z to do the nonlinear calculations and then transformed back to spectral space and stored in the nonlinear terms to go back into the main code. The general process is as follows.

1. Loop over y-planes, and for each plane, store the x-z data in a new variable (complex –> real).
2. Transform into 3/2 grid physical space. At this point, the domain is now $\frac{3}{2}n_x \times n_y \times \frac{3}{2}n_z$ because of how it is transformed. The extra grid points contain spectrally interpolated values.
3. Compute the cross product of velocity and vorticity, as well as the advection term of the scalar, and the nonlinear terms of the conformation tensor.
4. Set polymer forces (based on `ipeter`), add Brownian motion terms, and compute variable polymer concentration for targeting.
5. Apply immersed boundary forces. These forces are based on the cell type, which is stored in a variable called `imatrix`. The important cell types are 0 (no forces), 1 (forces generate a solid object), 4 (forces generate a temporary solid object), 6 (buffer zone forces), and 7 (suction region forces). There are clauses in here based on `flow_select` which apply a uniform pressure gradient (channel flow) or generate a vortex ring. For these (and any other body forces you may add), the forcing term is added directly to the $\boldsymbol{v} \times \boldsymbol{\omega}$ term in the appropriate direction.
6. Apply the force field to a solid surface. This is only relevant for a flat plate simulation, and it probably should be left alone otherwise.
7. Save real-space variables to a 3D variable for printing later.
8. Transform $\boldsymbol{v} \times \boldsymbol{\omega}$ and the nonlinear scalar/polymer terms back to spectral space.
9. Store nonlinear terms in their spectral variables for use in the main code.
10. End loop over y-planes and compute the λ_2-criterion.
11. Write flowfield data using 3D physical variables.
12. Check the interpolation function. This was originally used to test a spectral interpolation subroutine for use with integrating particles, but this appeared to be a dead end. The lines are commented out now, and they may be safely deleted unless you wish to figure out the spectral interpolation.
13. Integrate the particle trajectories. All of this is done inside the file `part_track.f90`, described below. It uses the 3D physical velocity and vorticity as inputs as well as the saved velocity data from the previous time step for the unsteady term.
14. Write mean u data. This is to check the mean velocity profile of a turbulent channel. The data is averaged over x and z and printed as $\bar{u}(y)$ at each time step. I use a supplementary MATLAB code to read this data and print it out as a graph.
15. Compute relevant flow/turbulence data. This is useful for drag reduction calculations for polymer channel flows. There is also some capability to do this spectrally, but I do it in physical space because it's simpler.
16. Check for CFL failure (numerical stability criterion) and end program.

4.2 FFTs: `ffts.f90`

This file contains all the FFT subroutines and functions that are used extensively in the code. Just ignore these.

4.3 Particle Tracking: `part`$_t rack.f90$

This file contains all the relevant subroutines and functions for integrating particle trajectories using flowfield information calculated by the CFD solver as inputs. The particles are computed independently as tracers or passive particles, depending on `particle_flag`. Tracers simply take on the velocity interpolated at its location for each time step and advect using explicit Euler. Passive particles are integrated via RK4. Particle trajectories are printed out to a file. For correlations, the swirl criterion is calculated at the particle location and the percentage is dumped out in files labeled 'swirlX' where X represents the user-defined threshold (hard-coded). There is a description of the code in the file itself. The code runs as follows.

1. Initialize particles – read in particle parameters and fluid data, compute density ratios, and force initial velocity to match local fluid velocity.
2. Check for sufficiently small time step (based on Stokes number of particle). If time step is not small enough for the particle, it will sub-step by breaking the time step into smaller chunks but using the same velocity field for each step. Since the overall time step is based on the fluid time scale, this should not affect the answer significantly.
3. Reset the output directory and read in initial values from the generated particle locations (`setup/particles/particles.dat`).
4. Loop (in parallel) through all particles independently, doing the following:
   (a) Interpolate fluid velocity and vorticity at particle location.
   (b) Compute $\frac{\mathrm{D}\boldsymbol{v}}{\mathrm{D}t}$ at nearest neighbors and interpolate to particle location. Velocity gradient inputs are used here and interpolated as well.
   (c) Compute $\frac{\mathrm{d}}{\mathrm{d}t}(\nabla^2\boldsymbol{v})$ at nearest neighbors and interpolate to particle location.
   (d) Call `RKStage` to compute forces on the particle based on MR equations for each stage of RK4. Any change to the particle equations should be made in this subroutine.
   (e) Update particle position.
5. Calculate (interpolate) swirl at particle position and determine percentage of particles in regions of certain swirl for correlation.
6. End loop and write particle data to a file.
7. Check for particle numerical stability and end subroutine.

4.4 Polymer/Scalar: `polymer`$_r outines.f90$

This file contains all of the additional subroutines used for calculating scalar and polymer evolution. It contains subroutines to implement Coriolis forces and thermal forces (and other body forces), calculate derivatives of the conformation tensor, compute the polymer forces. Generally, these should be left alone, especially the solver routines, unless you need to change something about the polymer model or the body force term. In the no-polymer case, the subroutine `derivscji`, which normally calculates the velocity gradients and conformation tensor derivatives, is incorporated into `dns_no_poly.f90` and only calculates velocity gradients and the Laplacian for use in the particle tracking calculations.

4.5 Flow Setup: `init`$_flow.f90$

This file contains the eponymous subroutine, `init_flow` in which the initial flow is specified based on `flow_select`, and routines to initialize the flow, including a Blasius solver for simulating a Blasius boundary layer. The main purpose of this subroutine is to define `initu`, `initv`, and `initw` which is used in the initialization of the flow field, and the buffer zone, if necessary to sustain the flow. Except for the Blasius BL, these are all analytical expressions of a flowfield.

4.6 Grid: `grid`$_size.f90$

This is a file generated by `preconfigure.py` which contains all the grid parameters. It is used in the other files as a module (where it says `usegrid_size` before the variable declarations). This makes using `include` and the `PARAM` file obsolete.

5 How to Run

Now that you know a little about the code, let's take a look at how to actually use it. It is assumed you will be running this in a Unix-based environment, so the instructions will reflect this. A few prerequisites are that your environment has the latest Intel Fortran compiler (although I believe the code compiles back to at least version 12.0.0), and you should have Python3 to run the configuration file (I'm not sure if it works with Python2).

5.1 Before You Start

5.1.1 Unix/Linux Commands

If you are not very familiar with working in a Unix-based environment, I strongly recommend you take a look at a Linux tutorial[2]. For editing files, you are free to use a basic text editor, but I would recommend either emacs or vim[3].

5.1.2 Adding Tecplot (optional)

This step is only relevant if you have access to Tecplot for flow visualization. If you do not, you may skip this section. To download the latest version of Tecplot, enter the following command in your terminal:

```
wget https://tecplot.azureedge.net/products/360/2021r1/tecplot360ex2021r1_linux64.sh
```

As a note, this is the version I used. Presumably newer versions would replace 2021 with the current year and r1 with the current release (r1 is probably safe though). You can also find Linux installation instructions on Tecplot's website. Once you've downloaded whichever version you want to work with, install Tecplot via

```
chmod +x tecplot360ex2021r1_linux64.sh
./tecplot360ex2021r1_linux64.sh
```

This will take you through an agreement and then it will prompt you for an installation directory. Choose anywhere convenient (a local directory). After the installation finishes, you should see it wherever you specified, but now we need to find the path, which you can find using the command

```
which tec360
```

If this turns up empty, you'll have to manually find the path. If it's not empty, you should see something like `path_to_tecplot/bin/tec360`, but we only need `path_to_tecplot`[4]. In order to use the Tecplot file writer, you will have to include this filepath to your system's `$PATH`.

5.1.3 Editing the `/.bashrc` file

Open up the file `~/.bashrc` using any text editor. If you installed Tecplot, you can add it to your system's `\$PATH` by adding the following lines at the end of the file:

```
export TECPLOT_360_INCLUDE=path_to_tecplot/include/
export TECPLOT_360_LIB=path_to_tecplot/bin/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$TECPLOT_360_LIB:TECPLOT_360_INCLUDE
```

Regardless of whether you install Tecplot, it's convenient to add an alias for fewer keystrokes every time you want to run the code:

```
alias preconfigure-dns='python3 code/src/preconfigure.py'
```

[2]This website is a very handy reference - `https://ryanstutorials.net/linuxtutorial/cheatsheet.php`
[3]Another handy reference - `https://vim.rtorr.com/`
[4]This is just a placeholder, what you'll actually see is a filepath unique to your system

You can also add any other helpful commands (like loading a module you need) or define other aliases while you're in the file. Once you're done, save and close the file and then source it via

```
source ~/.bashrc
```

This updates everything for your current session, but it will also run every time you start up or log into the system, so you only need to do this once.

5.2 Preconfigure DNS

Once you have all the configuration files the way you want them, it's time to compile the code so that it generates a binary executable that the computer can read. By default the code is set to use the Intel Fortran compiler (ifort) with particular flags that give us exactly what we want. To use a different compiler, you'll have to edit `code/src/preconfigure.py`. Thanks to the alias mentioned in the previous section, all you have to do to initiate the compilation is type the `preconfigure-dns` in the command line and hit Enter. You should see a message saying 'Preconfiguring DNS code' followed by some other information. This process handles 3 or 4 tasks: Setting up the grid size module, specifying the geometry, generating particles (optional), and compiling the DNS code. These actions are detailed below.

5.2.1 Grid Generation

This is a short section that reads the configuration files to determine the grid data to generate the file `grid_size.f90`. This file is used when the main code runs for easy access to the grid-related variables. I have also modified it to read the number of particles, as well as a flag to determine whether or not new particles are generated.

5.2.2 Geometry Setup

This section compiles `Geom.f90` and executes the resulting binary file. All this does is define the `imatrix` variable which contains cell type information for the immersed boundary forces.

5.2.3 Particle Setup

This section is skipped if `new_part` = 0. Otherwise, it compiles `generator.f90` and executes the resulting binary file. This fills out the initial conditions for the particles by distributing them in the domain based on the value of `new_part`.

5.2.4 Compile DNS Code

This section compiles the DNS code, which is now possible since `grid_size.f90` has been written. There is a flag here to determine whether or not we should include Tecplot files, but the binary file is generated and then placed in the `code/bin/` directory. Any compilation errors will produce a message which says 'Compilation failed! Check error in err_comp.txt'. The error message is written to the file `err_comp.txt` for readability. Otherwise, you will get a message saying 'DNS code is ready to run!'. Note that only errors in the compilation of the DNS code will be written to a file. Errors in the setup codes will simply be printed to the terminal screen since they are much rarer and less extensive.

5.3 Running the Code

How the code actually runs will depend on the machine(s) you're using. Below I'll mention a few things about running on TACC and a local machine (Linux). Consult general user guides for other systems.

5.3.1 TACC

TACC is an extremely powerful supercomputer cluster with state-of-the-art technology, so it is often prudent to take advantage of their systems if possible. For general etiquette and information about the TACC systems, visit their website: `tacc.utexas.edu`, or find one of their clusters specifically, such as Stampede2: `https://portal.tacc.utexas.edu/user-guides/stampede2`. For short runs and debugging, you should start an interactive development session, which allots you a private worker node (or more if you specify, but this code isn't parallelized well enough to warrant more than one node). Once the node is up and running, you are free to execute the code directly via `./code/bin/dns`. However, to utilize the threads, you have to specify OMP variables. If you're not doing any polymer calculations, you only need 8 threads, but you can still get marginal improvements up to 32 threads. Beyond that, there's virtually no speed-up. With polymer calculations, however, using 32 threads does see a significant speed-up, and you can go up to 64 threads with slightly better performance (the improvement is more significant the higher your grid resolution is, and the longer your runs are). I would recommend a maximum of 48 threads, but beyond this is unnecessary and will result in too much idle time for the threads in certain parts of the code. I usually run with 16 or 32 threads. As an example, if you want to run with 32 threads, execute the following commands:

```
export OMP_NUM_THREADS=32
export OMP_STACKSIZE=2G
ulimit -s unlimited
```

Now when you run the code, it will use 32 threads (or however many you specify). Occasionally, you may get a memory error, which is usually caused by the `OMP_STACKSIZE` variable being too big (not enough memory) or too small (too much data per thread). In general, the more threads you have, the less memory you need on the stack. Note that this method will have the code run in the foreground, which means you'll be able to see the log file printed out on your screen, and you won't be able to do anything else. If you want the code to run in the background, execute as

```
./code/bin/dns > outputs/log 2> outputs/err &
```

If you need to stop the code for whatever reason, one way is to bring it to the foreground with the command `fg %1`[5] and cancel the action with `CTRL+C`.

Alternatively, you can utilize three different files within the home directory, `clean`, `run`, and `hushrun`.

- `clean` wipes the output directory and makes empty flowfield and particle folders. This is useful if you don't need the output data and you want to copy the whole directory for a separate case. Copying the huge output files can take a long time.
- `run` wipes the output directory and executes the code as described above. OMP environment variables should be set here.
- `hushrun` does the same as `run` except it executes the code in the background.

To use these files, simply type `. <filename>`. This executes all of the lines in the file sequentially as command-line inputs, so it's just like typing each of those lines individually and executing one-at-a-time, but streamlined.
Since development sessions are limited to 2 hours, you'll have to submit longer jobs through the slurm scheduler. To do this, edit the `jobcode` file and change whatever you need to, then execute the command `sbatch jobcode`. Once the job is submitted, it will run once at its scheduled time (based on queue) for the specified duration, or until the code finishes or fails. Different partitions have different time limits, so keep this in mind for your jobs. For most partitions, the maximum length is 48 hours, and for particularly long runs, you can either keep restarting every 48 hours or run indefinitely on a local machine.

[5] This number will be shown in brackets, [], when you run the job

5.3.2 Local Machines

The only real difference between running on a local machine vs. TACC is that there's no scheduler or time limit on the local machine (along with whatever performance discrepancies there may be)[6]. The files mentioned above should work exactly the same in the local machine as they do on a TACC work node. Before you can run a code, however, you must have the appropriate dependencies installed. They are already installed on TACC, but they may or may not be installed on your local machine. The code is set up and compiled by executing a python script, so you need to have an updated version of python installed. If you are using an unmodified version of the code I sent along with these instructions, the compiler should be set to `ifort`, which is the Intel Fortran compiler. In order to use this, you must have Intel oneAPI installed and make sure your `$PATH` variables include the appropriate folders. You can also install the GNU Fortran compiler, but if you use this one, you will have to update the compiler option in `preconfigure.py` along with the compiler flags. I haven't used this, so I'm not sure what exactly it should look like, but I'm sure the information can be easily found online somewhere.

5.4 Visualization

As the code runs, it will print out flowfield and/or particle data in the `outputs/` directory or the `GMU_out/` directory depending on the `print3d` flag. These are what we will use to visualize the results of the code. My only flow visualization experience is with Tecplot, but other file output compatibility may be added in a future version of the code. The Tecplot files can be moved (if necessary) to a local directory from which you can upload them directly to Tecplot. For the flowfield, the `grid.plt` file must be first in the list of files uploaded, and it should be loaded instantly. If the data files are in ASCII, Tecplot will need to convert them to binary, which may take quite a while if you have many large files. The particle output files are set to be ASCII data files unless there are more than 100 particles. The ASCII files will be very small and will load into Tecplot quickly, otherwise, it will create a .szplt file for the particles (which is much more convenient when we have several thousand particles to track at once). Once your data are loaded into Tecplot, you can use the plethora of tools to visualize velocity, vorticity, swirl, and derived variables through iso-surfaces, slices, and more. The particles can be shown using 'Scatter', but be sure to turn off the flowfield data so it doesn't turn your display into an incomprehensible mess. From the scatter menu, you can change the color, shape, and size of the particles for the best visual effect.

[6]Of course, TACC has the capability of multiple nodes to handle huge computations, but our code can only use one node

6 Example 1 – 2D Couette Flow

Now that we've gone through all the details of how the code works and how to run it, the next step is to get your hands dirty with an exercise. Let's start simple and small with a 2D laminar flow. For this example, start within the folder called `2D_flow_example` and edit the files only in this directory. In this example, you'll get practice changing the setup variables, compiling, running, and visualizing. Since it's a 2D case, the runs should be very quick and the file sizes are relatively small so you can produce and see the results quickly.

6.1 Couette Flow Setup

First, let's look at the simple steady Couette flow. Remember that Couette flow has a linear velocity profile between the velocity of the top and bottom walls. From the reference frame of the bottom wall, let's say we have U_{top} for the velocity of the top wall and $U_{bot} = 0$ for the velocity of the bottom wall. Then the velocity profile is given by

$$u(y) = U_{top}\frac{y}{L_y}$$

Remember that L_y is the domain length in the y-direction (denoted by `YL` in the code). Take a look at `code/src/dns/init_flow.f90` and verify this flow profile in the 2D Couette flow subroutine.

Now let's set up the parameters inside `setup/dns.config`. I've left variables for you to change as asterisks (*) in place of numbers, so if you try to compile the code as it is, you'll get an error in the first step. There are a total of 9 variables you need to set. Six of these variables are the physical and computational domain sizes. Since we're running a 2D case, we'll shrink down the z-direction so it can be neglected. In reality, the code does not have 2D flow capability, but if we set $n_z = 4$ and $L_z = 0.078125$ (which is 5/64), we can effectively ignore that direction. Next, L_y should naturally be 2.0 because the code normalizes the domain to be $[-1, 1]$. We can let L_x be anything, but it should at least be twice the length of L_y because of the periodic boundary condition. I suggest setting it to 5.0, and let $n_y = n_x = 64$. Leave `dt` set to 10^{-4} s for now, as this is important if you want to replicate the figures in the next example.

Next, set `Uinf` to be any positive real number. However, keep in mind that the faster the fluid velocity, the smaller your time step needs to be (for a given cell size) to achieve numerical stability. Set the `flow_select` flag to the appropriate value for 2D Couette flow. Lastly, set the `print3d` flag to the appropriate value for your visualization. If you have Tecplot (and TecIO installed), this should be 3, otherwise you'll have to make do with ASCII files and set it to 1.

6.2 Couette Flow Run

Once you've replaced all asterisks with the appropriate values in the setup file, you can compile the code using `preconfigure-dns`. Note that the preconfigure file is set up to compile using `dns_no_poly.f90` so there will be no polymer calculations. For all cases where you don't need to calculate scalar concentration or the polymer conformation tensor, I recommend using the no-polymer file as it will perform better when it's not making a bunch of unnecessary computations on unused variables. If everything is set up correctly, you should see Figure 1 printout on the terminal. If there is an error in the source code, you will see a message as in Figure 2. In that case, open `err_comp.txt` to view details on why the compilation failed.

After a successful compilation, you can run the code in the foreground by executing '`. run`' in the command line. *Be sure you're in a development session first if you're on TACC! Never run on the login nodes!* As soon as you run the code, you'll see the log printed out to the terminal. In the beginning, it gives some variables, but after that each time step is the same, giving the current time step and maximum CFL number, along with some progress notes. The code is set to only run for 10 time steps for you to be sure the code is working properly. The code should run in under a second, and you can move the output files to a local directory, if necessary, and load them into Tecplot for visualization.

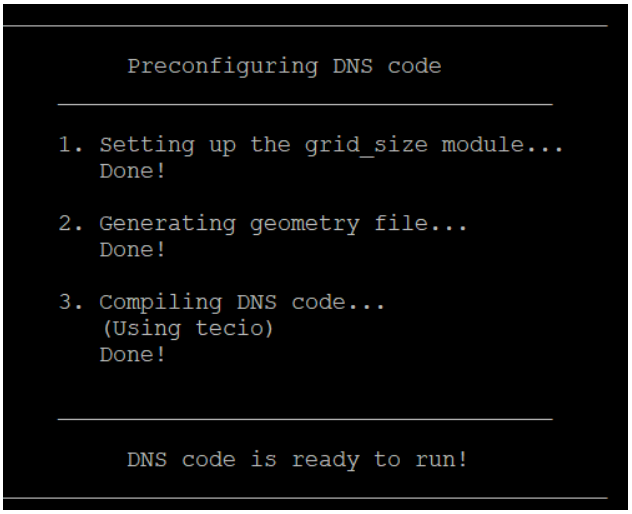

Figure 1: Successful compilation message

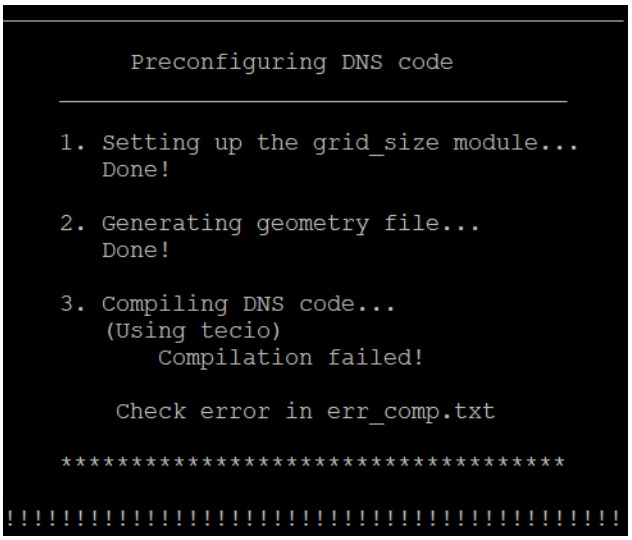

Figure 2: Failed compilation message

6.3 Couette Flow Visualization

The data files are located in `outputs/flowfield`, whose names are based on the time step. If you upload those files to Tecplot, it should automatically detect that it's a 2D x-y plot. After some adjustments, you can display the velocity profile as a color gradient, as in Figure 3. You can easily see that the profile is indeed linear with $U_{top} = 10$ cm/s. Check to be sure the initial time step and the last printed time step are identical.

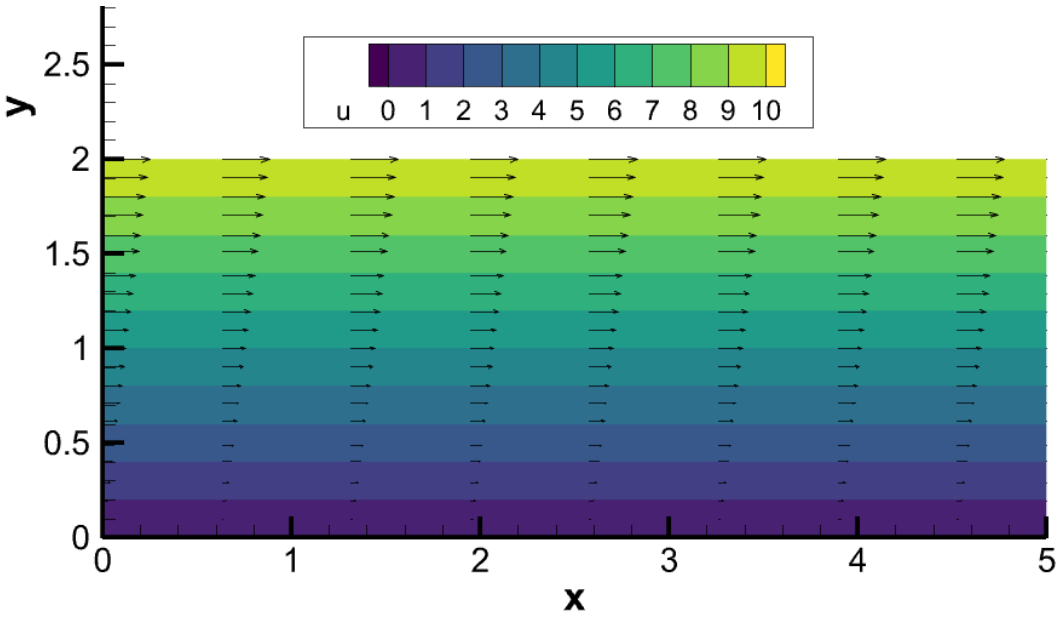

Figure 3: Couette flow visualization

7 Example 2 – 2D 'Turbulent' Channel

Now that you've run your first DNS code, we can step it up to something a little more interesting: a 2D channel. Staying in the same directory, simply switch some variables in `setup/dns.config`. The domain can stay the same, but you'll need to change the `flow_select` flag to 1, and set $Re_\tau = 125$. There's no need to recompile since all you changed were variables to be read at run time, but you can go ahead and run this again and take a look at the flowfield. It should look like Figure 4.

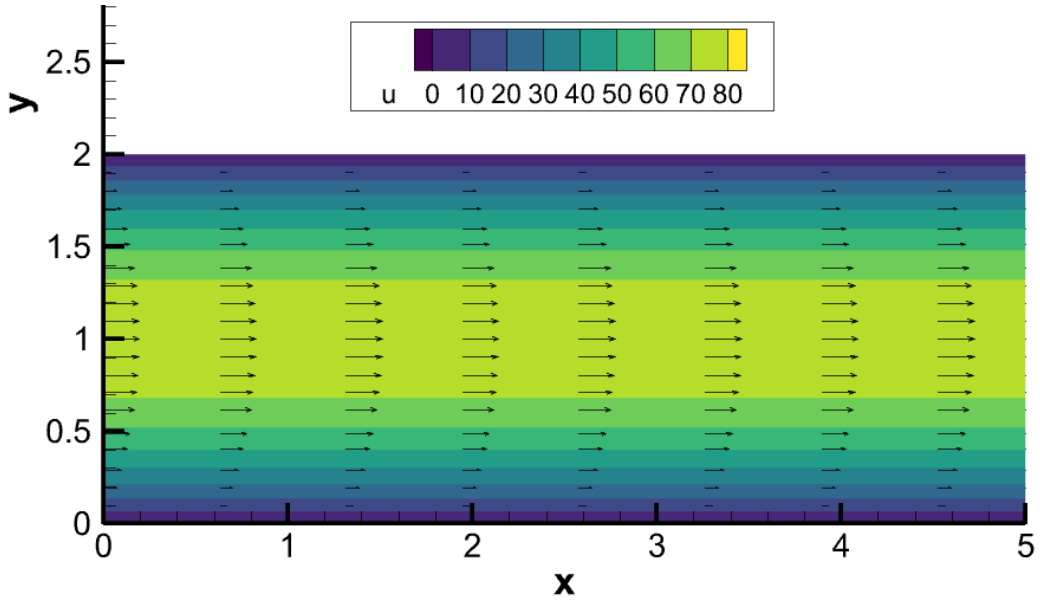

Figure 4: 2D Channel flow visualization

7.1 Transition to Turbulence

Great, we've got a working channel flow, but really what we're interested in is turbulence. Although turbulence isn't well-defined in 2D, we can generate quasi-turbulence to see some interesting unsteady vortex structures. To do this, we add obstructions to the normal flow and the solver takes care of the rest since turbulence is deterministic. In general, we need to be careful because the solver has stability limitations, so we try to avoid large gradients, but there are a couple default cases programmed in the geometry that should work fine. You can set these by changing the `geomtype` flag from 0 to 2 or 3. They are named '2D brick' and '3D brick', respectively, but they both work in either 2D or 3D. Try both of these and note the difference. To see the source code and possibly create your own bricks/obstructions, look at `code/src/geometry/Geom.f90`.

These structures are implemented in the code through immersed boundary forces, and they define effective solid surfaces within the flow. However, this force is only active from $t = 0$ until the code reaches the `perturbstayingtime` time step. Compile and run the code for 40,000 time steps, printing every 1,000 time steps with `perturbstayingtime` = 10,000. This should take a few minutes to run, so get up, stretch, and walk around a little bit. You've been sitting in front of a computer for too long.

Once the code finishes, you'll have 100 output files, but they'll be pretty small. Load these into Tecplot and take a look at what the flow is doing. The end state should look something like Figures 5 and 6. Note that these are both visualizations of the same flow state, but I'm looking at different variables in each case.

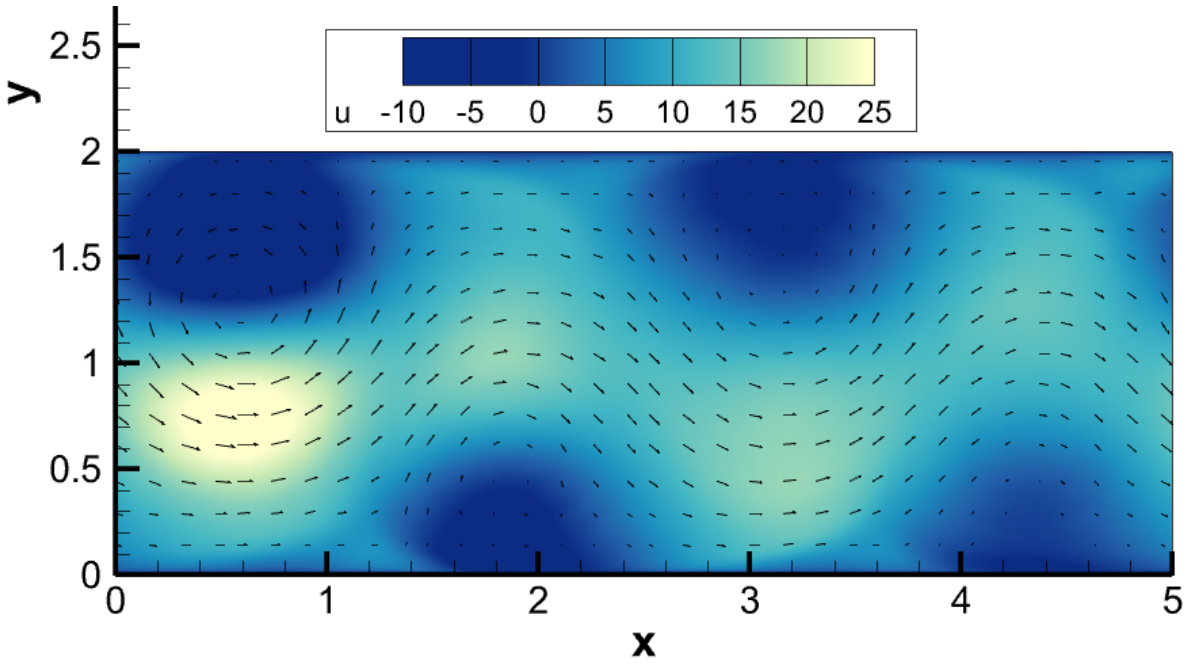

Figure 5: 2D Channel at time step 40,000, visualized velocity field

7.2 Restart and Particles

Now we've set up a 2D channel flow with a few distinct, stable but unsteady vortices. What can we do with that? One of the reasons this version of the code was developed is because we wanted to track particle movements in complex flows. The general strategy is to run the flow solver to develop the desired flow state (turbulence or quasi-turbulence in this case) and then insert particles and see what happens. In order to do this, we need to save the last flow state of the previous run and use it as an input for a new run. The

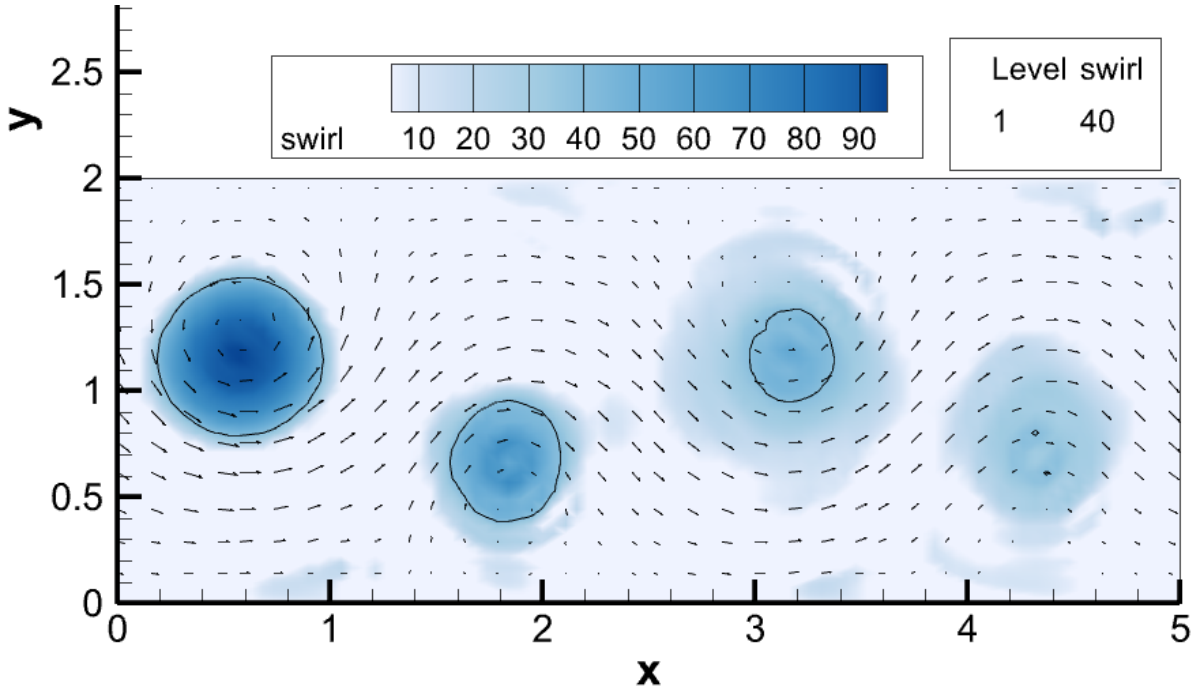

Figure 6: 2D Channel at time step 40,000, visualized swirl criterion

code is designed to automatically save all the relevant flow variables every `iprnfrq` time steps in the file `outputs/last-restart`. When you want to use this file for the next run, you must first transfer this file to the setup folder. E.g., from the main directory, execute `cp outputs/last-restart setup/restart`. Then, in `setup/dns.config` change the `irstrt` flag to 1 and change `geomtype` back to 0. You can verify that this works by recompiling and running a few time steps and printing out the flowfield to verify that it looks like the last printout from the previous run.

Once you've verified that the restart works, you can add some bubbles to your code. Inside `setup/dns.config`, change the number of time steps to 10,000, change the number of particles to 10,000, and set `particle_flag` and `new_part` to 1. Recompile the code, and you should get a slightly different message on the terminal screen that looks like Figure 7 telling you about the particle generation. For subsequent runs where you don't want to change the number or distribution of particles, you will need to reset `new_part` to 0.

Once it's compiled, run the code. Even though the particle tracking routine isn't explicitly parallelized (though not for lack of trying), the compiler flags do a good job of making it more efficient (though it's still noticeably slower than fluid-only runs). The run time should only be a few minutes even with 10,000 extra RK4 integrations each time step. Loading this data into Tecplot, I get Figure 8. Play around with bubble sizes and density ratios and compare the results in Tecplot. What do you notice?

```
      Preconfiguring DNS code
 ____________________________________

1. Setting up the grid_size module...
   Done!

2. Generating geometry file...
   Done!

3. Generating 10000 random particles...
   Done!

4. Compiling DNS code...
   (Using tecio)
   Done!

 ____________________________________

      DNS code is ready to run!
```

Figure 7: Compilation message when generating particles

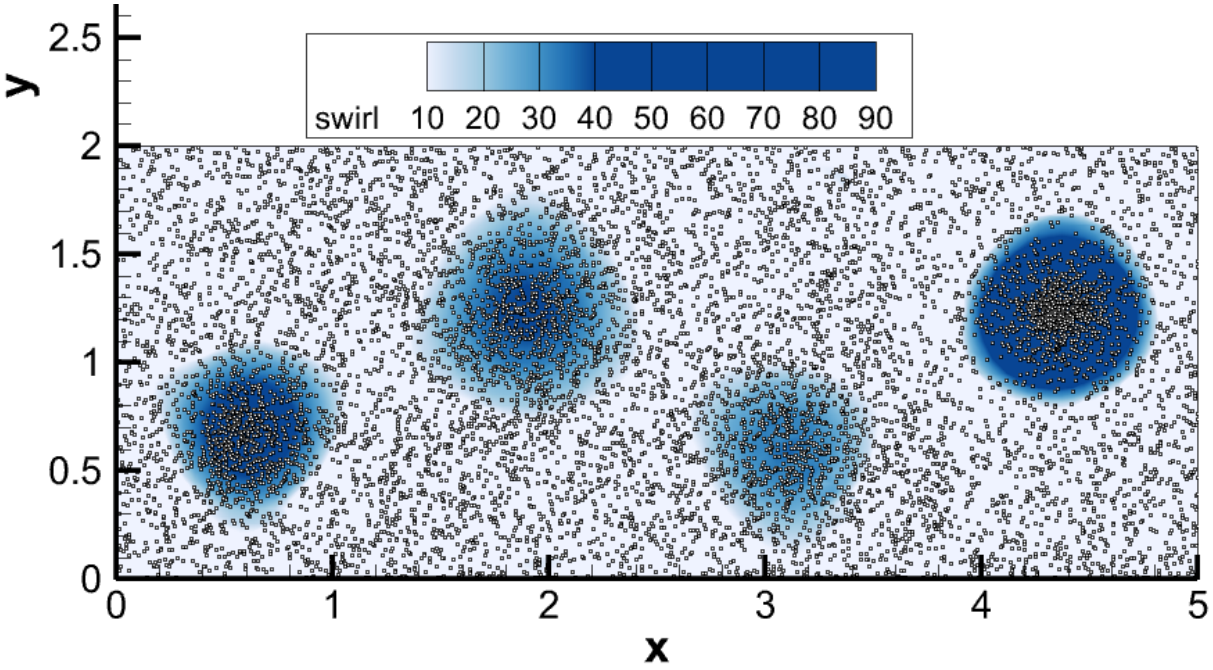

Figure 8: 2D quasi-turbulent channel with bubbles

8 Example 3 – 3D Turbulent Channel

Now, try extending the 2D case to 3D on your own. Here are some things you will need to change:

- Domain size (`XL`, `YL`, `ZL` – the z-direction of course needs to be extended, and the x-direction should be lengthened as well.

- Grid size (`nx`, `ny`, `nz` – You'll need more grid points to resolve the turbulent structures in 3D
- Restart – Don't try to use the 2D restart file for 3D, you will simply get an error that ends the run.

That's really it. I suggest turning the particles off while you run this and then add them once you've got a turbulent 3D channel flow.

8.1 Some Tips and Suggestions

Use the paper "Turbulence statistics in fully developed channel flow at low Reynolds number" by Kim, Moin, and Moser (1987) as a reference for your domain size, grid size, and Re_τ turbulence profiles. Compare what you can with the outputs you have. The code is set up to provide a spatially averaged streamwise velocity profile (`mean_u_data.dat`) that you can examine using Python or MATLAB quite easily. Note that the data needs to be extracted carefully because it's simply a data file as a string of numbers, so you'll have to manually set n_y in the post-processing code and save every n_y values in its own vector/array.

Use the 3D brick `geomtype` with `perturbstayingtime` = 10,000. You may need to adjust the time step because of the higher velocity gradients around the brick, but after a while, you can save the restart file, and do a new run with a larger time step to speed up the development. I usually let it run until the mean profile doesn't change much over time and the velocity gradient is the same as laminar flow ($u_\tau = 1.8$).

There is a file in the Github repository called `mean_U.m` which is a MATLAB code I wrote to look at the mean velocity profile and check to see if the turbulence statistically stationary. You may use that directly or adapt it to suit your needs. You will need to ensure the domain and grid size variables are consistent and that the filename is correct for however you saved the mean u data.

Check the initial condition by plotting a y-z slice and ensuring the laminar case looks identical to 2D. Once the flow is turbulent, you'll see a transient velocity field and moving structures by plotting iso-surfaces of the swirl variable. In Figure 9, I plotted a snapshot of the channel with $Re_\tau = 180$ and iso-surfaces of swirl set to 40.

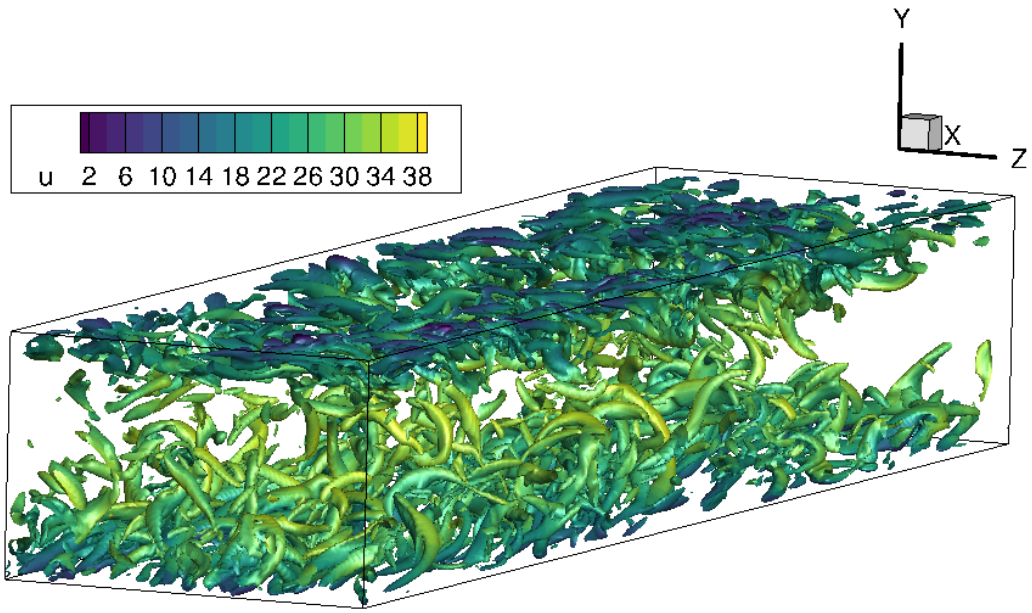

Figure 9: 3D Channel snapshot with iso-surfaces of swirl set to 40

9 Example 4 – Vortex Ring with Polymer

This code was originally designed specifically to calculate channel flows with the highest accuracy possible, and that's just what you've done in the previous example. Now let's move on to some other cool features of the code that were added much later. Let's simulate a vortex ring and then introduce a particle that emits polymer after a short delay.

9.1 Generating the Ring

In this code, the vortex ring is generated via an impulsive cylindrical body force, added to the nonlinear terms in physical space. To set the conditions, all you need to do is change `flow_select` to 5. The boundary conditions will automatically be changed to shear-free on the top and bottom (x and z boundary conditions are always periodic). Other previously relevant variables (such as `Uinf` and `R_tau`) are ignored. Be sure to change the geometry type to 0, and I suggest starting with `RE` = 50.0 to avoid numerical issues for now. You don't need as high spatial resolution as you did for the 3D channel, but you should adjust the grid size such that the outputs look smooth when visualized.

The force is defined in the last section of the main setup file. The first parameter to set is `tfrac`, which determines how long the body force is active, as a fraction of the total time of the simulation. For example, if you run a simulation for 10,000 time steps, and `tfrac` = 0.1, then the force will be active from time step 1 to time step 1,000. For a run of 15,000 time steps, leave it at 0.1 so it will be active for 1,500 time steps. Then you can set the position with `xc`, `yc`, and `zc`, which define the location of the centroid of the force. Finally, `L` and `rad` set the length and radius of the cylindrical force, respectively. I've set all the values in the example file the same as what I show in the figures below, but feel free to play around with various parameters and see how it affects the ring!

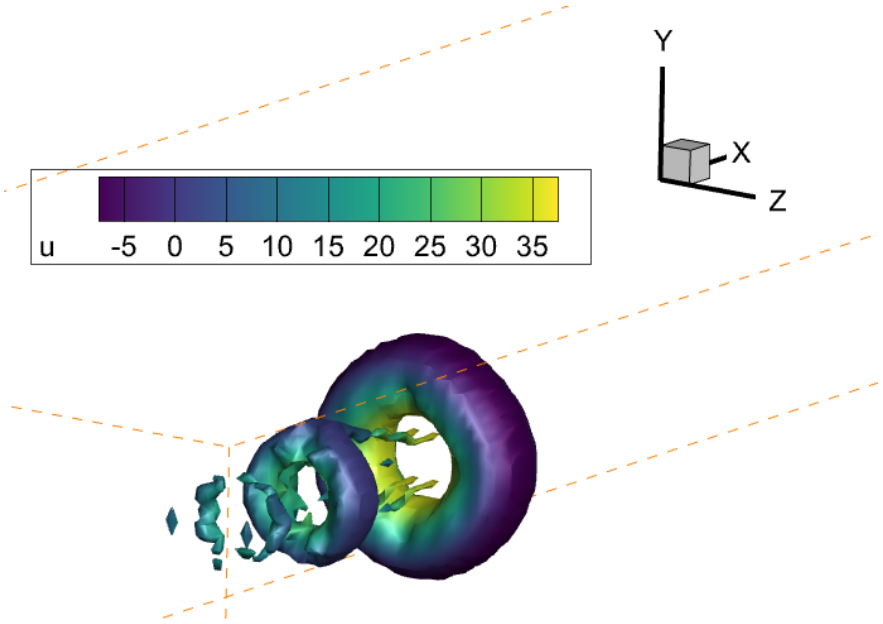

Figure 10: Initial vortex ring structure. Iso-surface of $\lambda_{ci} = 40$ (Swirl).

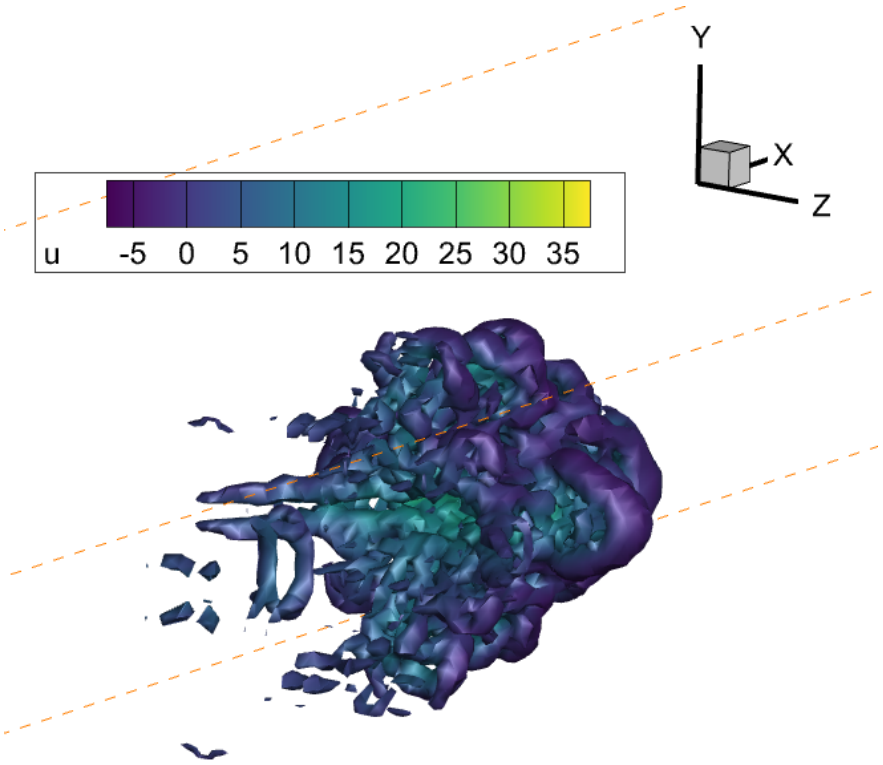

Figure 11: Breakdown of the vortex ring into turbulence. Iso-surface of $\lambda_{ci} = 40$. At lower viscosity (increase `RE`), you will see a more stable ring that slowly dissipates.

Figure 12: Caption

9.2 Adding Scalar

After you have a satisfactory vortex ring, you can start adding some passive scalar for a little pizzazz. We'll go over three different ways you can use the passive scalar, then after that we'll get into how to use the polymer effects.

9.2.1 Initial Fixed Distribution

First is the simplest case: an initial 'blob' of scalar somewhere in the domain. There is no source term, just a given distribution of scalar and evolution according to the advection-diffusion equation. To do this, you simply need to change `scl_flag` to 1 inside `setup/dns.config`. It's currently set up so that the scalar distribution is a Gaussian spheroid just above the center of the channel in front of where the vortex ring is generated. You can change the location of the scalar with the variables `xshift`, `yshift`, and `zshift`. These variables determine how far the center of the distribution is from the absolute channel center. Feel free to change these and move it around to see how it moves with the flow. The other relevant parameters are `sigmax`, `sigmay`, and `sigmaz` which determine the standard deviation of the Gaussian distribution in their respective directions. Play around with these as well to see what the shape looks like.

9.2.2 Fixed Source

Sometimes it's more realistic to show the introduction of a scalar into the channel from a point source. To do this in the code, change `scl_flag` from 1 to 2 and be sure you have at least one particle in your `setup/particles/particles.dat` file. The source will take the location information from this file, so either set it manually or generate a specific group of particles that you want to emit polymer.

For now, we're going to use stationary particles to represent a fixed source. To do this, change the `particle_flag` to -1. In the particle tracking subroutine, this will set the initial position from the input file but then skip subsequent integrations so that the particle position never changes. Try to set the particle position to be the same location as the initial fixed distribution in the previous section. Remember that these are defined differently so think carefully!

You will also need to set a few more parameters which are only relevant for a source. The first is the rate at which scalar is released, `polyrate`. In general, this should be inversely related to the number of particles you have (fewer particles, higher source rate, vice versa). Next is the total mass of polymer (in milligrams) allowed in the domain, `mpoly`. The total mass of polymer is calculated by integrating the scalar concentration throughout the volume at each time step until it reaches the threshold set by `mpoly`. Last are the first and last time steps to release scalar, `src_start` and `src_stop`, respectively. If `src_stop` is less than `src_start`, the code will automatically set it to the number of time steps. In this case, the limiting factor will be `mpoly` which will cut off the source regardless of what `src_stop` is. Therefore, `src_stop` is only relevant if you want to limit the time of release rather than the total amount released.

9.2.3 Moving Source

Making the source move is only a slight change in the setup. Simple change the `particle_flag` from -1 to 0 or 1, depending on how you want the particle to move. Set the particle parameters to whatever you want to simulate (tracer, bubble, heavy, etc.) and let it go. You should not need to recompile, but it never hurts to do it anyway. My example is shown below in Figure 13.

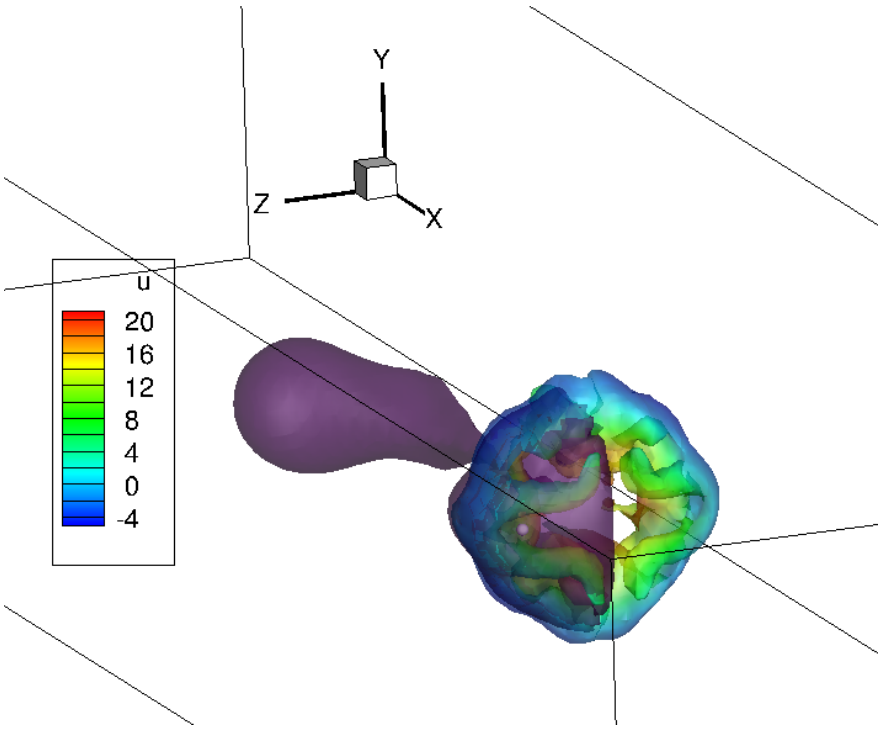

Figure 13: Snapshot of a vortex ring (iso-surface at $\lambda_{ci} = 40$, colored by u velocity) carrying a particle (white sphere) emitting scalar (purple iso-surface at 250 PPM)

9.3 Targeting

Last is doing everything all together. This is the true purpose of this code - to perform polymer calculations in targeted locations based on concentration of a passive scalar. Keep the settings you used in Section 9.2.3, but now change the flag `itarget` from 0 to 1. Run the code again and look at how the polymer disrupts the structure of the vortex ring. In my example figures, compare Figure 14 to Figure 13. Notice how the ring is smoother and how the left side of the surface is "eaten" by the polymer as kinetic energy is stored as elastic potential energy. Later in the flow, you may see a swirl iso-surface inside the polymer region from the rubber band-like effect as the elastic potential energy is converted back into kinetic (with losses).

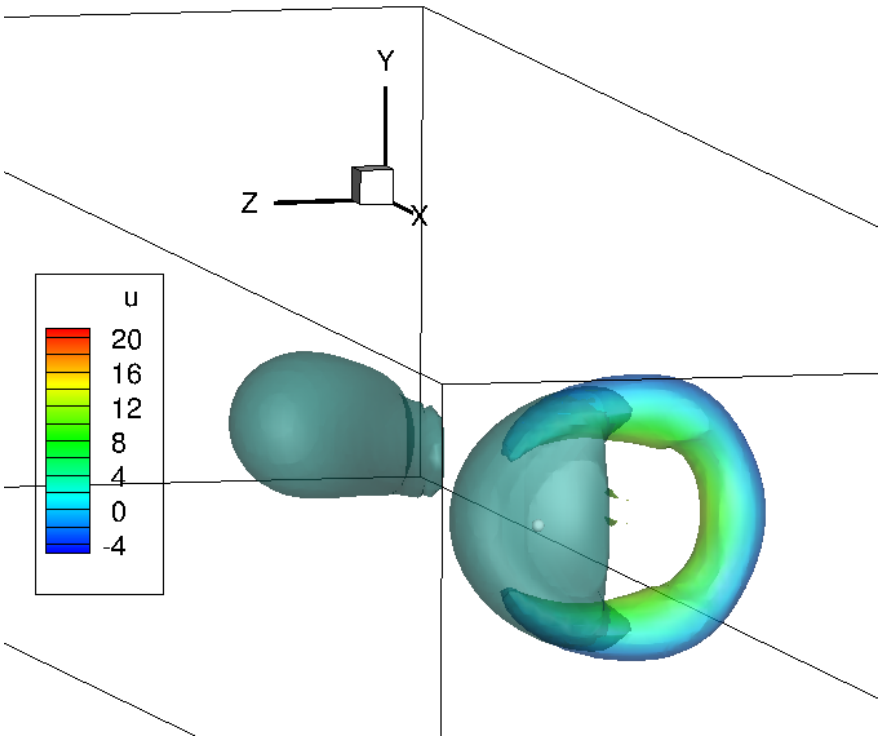

Figure 14: Snapshot of a vortex ring (iso-surface at $\lambda_{ci} = 40$, colored by u velocity) carrying a particle (white sphere) emitting polymer (teal iso-surface at $\beta = 0.6$)