Simulating Fluid Dynamics

Ronan Hanley, Cassie Metzger, & Jeremy Robin

1 Methods

To begin visually depicting the motion of a fluid over time, we needed a file that would be able to quickly calculate the equations necessary to describe the fluid's evolution. Specifically, we aimed to solve the Euler equations (the reduced form of the Navier-Stokes equations) in 2D. This is implemented in the file `fluid_solver_2d.h`. The Euler equations are solved over an area $L_x \times L_y$ that can be broken down into a grid of $N_x \times N_y$ cells. In order to enforce boundary conditions, one cell on each end is reserved as a ghost cell, making the physical cell domain $(N_x - 2) \times (N_y - 2)$. `fluid_solver_2d.h` also takes in the adiabatic index that describes the degrees of freedom for particles within a fluid. In this project, we assume an ideal fluid and use $\gamma = \frac{5}{3}$. `output_dt0` is also entered into the fluid solver so that the equations can be evolved over time.
A fluid is described by the mass, momentum, and energy of each particle within it. Therefore, we implement the *finite volume method* with second-order accuracy.

2 Finding Δt

We can solve for Δt with the following formula:

$$\Delta t = C_{CFL} \times \frac{\min(\Delta x, \Delta y)}{\max_{ij}(\sqrt{\gamma \frac{P_{ij}}{\rho_{ij}}} + \sqrt{v_{x,ij}^2 + v_{y,ij}^2})}$$

It should be noted that $v_{max} = \max_{L,R}(\sqrt{\gamma \frac{P_{ij}}{\rho_{ij}}} + \sqrt{v_{x,ij}^2 + v_{y,ij}^2}) = c + |\mathbf{v}|$. In this case, c represents the speed at which density fluctuations travel through a fluid while $\mathbf{v}$ represents the velocity of the fluid. Therefore, it makes sense that $\Delta t \propto \frac{\Delta}{v_{density}+v_{fluid}}$ where Δ represents the displacement. This concept is implemented in the method `find_dt()` where the numerator remains constant and the denominator changes by an index `idx` which represents the iteration across an array $N_x \times N_y$ using *column major*. Here, `int idx = i + j*Nx`. This function returns the new Δt, which changes according to which point on the 2D grid is being referred to. Overall, we've noted that more stable results can be produced with a lower C_{CFL}, but the run time drastically increases. As a result, we chose a C_{CFL} condition of 0.4.

3 Applying Periodic Boundary Conditions

Something that is incredibly important to note is that we will be applying the periodic boundary condition to each of the relevant quantities, including both conserved and primitive variables. Because this is a condition that will need to be applied in several places several times throughout the code, we wrote a function that takes in a C++ vector and applies the periodic boundary condition to it. In order to enforce this boundary condition, we can set the ghost cells to be equal to the last real cell on the other side of the domain. For example, we can set the values in the first row (which is a ghost row) equal to the values in the second to last row. The implementation is shown below

```
for(int j = 0; j < Ny; j++) {
  f[0 + j * Nx] = f[(Nx - 2) + j * Nx];
  f[(Nx - 1) + j * Nx] = f[1 + j * Nx];
}

for(int i = 0; i < Nx; i++) {
  f[i + 0 * Nx] = f[i + (Ny - 2) * Nx];
  f[i + (Ny - 1) * Nx] = f[i + 1 * Nx];
}
```

4 Primitive to Conserved

The flux calculation only changes the conserved variables. Therefore, in order to properly update all of the variables, we need to put the primitive variables in terms of conservative variables. Therefore, we can rewrite ρ, v_x, v_y, and P using the following equations.

$$\rho = \frac{M}{\Delta x \Delta y} \qquad v_x = \frac{p_x}{\rho \Delta x \Delta y}$$
$$v_y = \frac{p_y}{\rho \Delta x \Delta y} \qquad u = \frac{U}{\rho} - \frac{1}{2}v^2$$
$$P = u(\gamma - 1)\rho$$

Here ρ represents the density of the fluid, v_x and v_y represent the velocity of the fluid in the x and y directions, u represents the internal energy of the fluid while U is the total energy and P is the pressure of the fluid.
Next, we enforce the boundary conditions from Section 3.

5 Conserved to Primitive

We also need to be able to convert the conserved variables to the primitive variables. To do this, we used the following relations:

$$\rho = \frac{M}{\Delta x \Delta y} \qquad v_x = \frac{p_x}{\rho \Delta x \Delta y}$$
$$v_y = \frac{p_y}{\rho \Delta x \Delta y} \qquad u = \frac{U}{\rho \Delta x \Delta y} - \frac{1}{2}(v_x^2 + v_y^2)$$
$$P = (\gamma - 1)\rho u$$

Where M is mass, p_x and p_y are momenta in x and y respectively, v_x and v_y are velocities in x and y respectively, u is the internal energy of the fluid, U is the total energy, and P is pressure. In practice, we used for loops to iterate through every point in the domain and assign primitive values based on these relationships. Finally we enforced periodic boundary conditions on all primitive variables.

6 Primitive Half Step

At every time step, after Δt has been calculated the next step is to take a trial half-step and update the primitive variables accordingly. We use the primitive version of the Euler equations and the FCTS scheme to take this half step. We rearranged the Euler equations to be in the following form:

$$\frac{\partial\rho}{\partial t} = -v_x\frac{\partial\rho}{\partial x} - v_y\frac{\partial\rho}{\partial y} - \rho(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}) \quad (1)$$

$$\frac{\partial v_x}{\partial t} = -\frac{1}{\rho}[\frac{\partial P}{\partial x} + v_x\frac{\partial v_x}{\partial x} + v_y\frac{\partial v_y}{\partial y}] \quad (2)$$

$$\frac{\partial v_y}{\partial t} = -\frac{1}{\rho}[\frac{\partial P}{\partial y} + v_x\frac{\partial v_x}{\partial x} + v_y\frac{\partial v_y}{\partial y}] \quad (3)$$

$$\frac{\partial P}{\partial t} = -\gamma P[\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}] - v_y\frac{\partial P}{\partial y} - v_x\frac{\partial P}{\partial x} \quad (4)$$

As an example, below is how we implemented the equation for density in the actual code.

```
double drho_dx = (rho[idx+1] -
    rho[idx-1])/(2.0*dx);
double drho_dy = (rho[idx + Nx] -
    rho[idx - Nx])/(2.0*dy);
double dvx_dx = (vx[idx+1] -
    vx[idx-1])/(2.0*dx);
double dvy_dy = (vy[idx + Nx] -
    vy[idx - Nx])/(2.0*dy);
rho_tmp[idx] = rho[idx] - dt *
    (vx[idx]*drho_dx +vy[idx]*drho_dy+
    rho[idx]*(dvx_dx + dvy_dy));
```

Finally, we imposed periodic boundary conditions on the temporary vectors and then assigned those temporary vectors to the actual primitive variables.

7 Extrapolate to Interface

After achieving 2nd order accuracy in time, we need to achieve 2nd order accuracy in space. We can do this by taking the spatial derivative of the primitive variables. To get a more visual understanding of this procedure, we've constructed the following diagram.

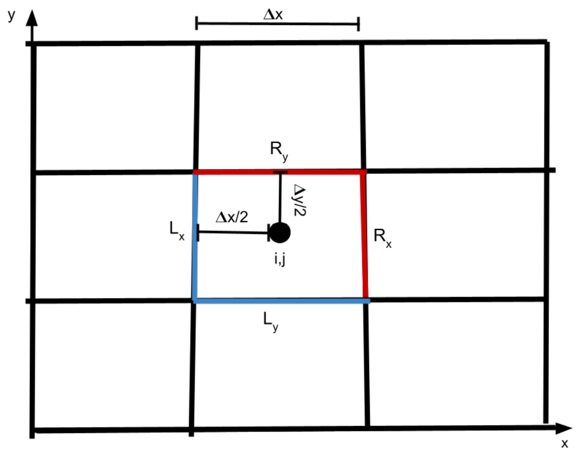


Figure 7.1: A visual depiction of the xy grid our code operates on

Therefore, to find the densities on the left and right boundaries of each cell we can essentially take the derivative or "average" of the cells on either side of cell (i, j) and multiply this by a half. To advance to the right (positive direction), we add this value to our original value at (i, j) and to move to the left, we subtract this value from the original value at (i, j). We implemented equations 25-28 for each primitive variable. Our implementation of v_x can be seen below, this structure is exactly copied for all other primitive variables.

```
  vx_Lx[index] = vx[index] -
  0.25 * (vx[index+1] - vx[index-1]);
  vx_Rx[index] = vx[index] +
  0.25 * (vx[index+1] - vx[index-1]);
vx_Ly[index] = vx[index] -
0.25 * (vx[index+Nx] - vx[index-Nx]);
vx_Ry[index] = vx[index] +
0.25 * (vx[index+Nx] - vx[index-Nx]);
```

8 Computing the Fluxes

In order to calculate the flux at each point along the grid, the flux on the left and right cell boundaries must be determined. This can be done using the variables determined from Section 7. In addition, v_{max} must be calculated and requires the velocity on the left and right cell boundaries. This, again, can be calculated with the extrapolated primitive variables. For each point along the grid, we begin by calculating v_{max} in the x and y directions using Eq. 30. Next, we calculate the flux to the left and right. It's important to note that F_{Rx} is really in regards to the **left** side of the point to the right, while F_{Lx} is the **right** side of the point to the left. The same is true in the y-direction. Therefore, we must write F_L in terms of the right and F_R in terms of variables to the left. To do this, we utilize Eqs. 17-20. We use the $\frac{\partial}{\partial x}$ term to denote the flux in the x-direction and the $\frac{\partial}{\partial y}$ to denote the flux in y-direction. We do this for the mass, the momentum in both the x and y directions and the energy. The code computing this for p_x is shown below:

```
//momentum_x in the x
F_Lx = rho_Rx[idx]*vx_Rx[idx]
*vx_Rx[idx]+ P_Rx[idx];
F_Rx = rho_Lx[idx+1]*vx_Lx[idx+1]
*vx_Lx[idx+1] + P_Lx[idx+1];
//momentum_x in the y
F_Ly = rho_Ry[idx]*vx_Ry[idx]
*vy_Ry[idx];
F_Ry = rho_Ly[idx+Nx]*vx_Ly[idx+Nx]
*vy_Ly[idx+Nx];
```

Next, we find the conserved quantity to the left and to the right (Q_R, Q_L). In the case of the mass flux and the momentum flux, we need to write this quantity in terms of ρ since mass and momentum are not conserved. We can write the conserved quantity of mass as ρ. We do not need to multiply ρ by any area or volume because this is done in Section 9. Similarly, we can express $p_{x,y} = \rho_{x,y} v_{x,y}$.

```
//momentum_x in the x
Q_Lx = rho_Rx[idx]*vx_Rx[idx];
Q_Rx= rho_Lx[idx+1]*vx_Lx[idx+1];
//momentum_x in the y
Q_Ly = rho_Ry[idx]*vx_Ry[idx];
Q_Ry= rho_Ly[idx+Nx]*vx_Ly[idx+Nx];
```

Finally, we use the following code to compute the flux in the x-direction and then repeat the procedure to calculate the flux in the y-direction.

```
//momentum_x in the x
momx_flux_x[idx]= (0.5)*(F_Lx+F_Rx) -
(vmax_x/2.0)*(Q_Rx-Q_Lx);
//momentum_x in the y
momx_flux_y[idx]= (0.5)*(F_Ly+F_Ry) -
(vmax_y/2.0)*(Q_Ry-Q_Ly);
```

After the fluxes in the x and y directions are computed for all variables, the boundary conditions are reinforced.

9 Update Conserved

Once we have calculated the fluxes in each conserved variable it's very simple to use those fluxes to update the conserved variables. Below is the code used, with mass as an example.

```
mass[idx]= mass[idx] -
  (mass_flux_x[idx]-
  mass_flux_x[idx-1])*dy*dt
  - (mass_flux_y[idx]-
  mass_flux_y[idx-Nx])*dx*dt;
```

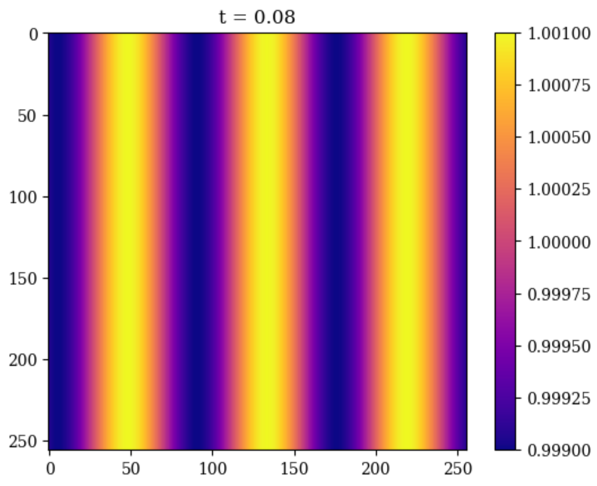
Then we applied periodic boundary conditions to all conserved variables.

10 Sound waves in a fluid

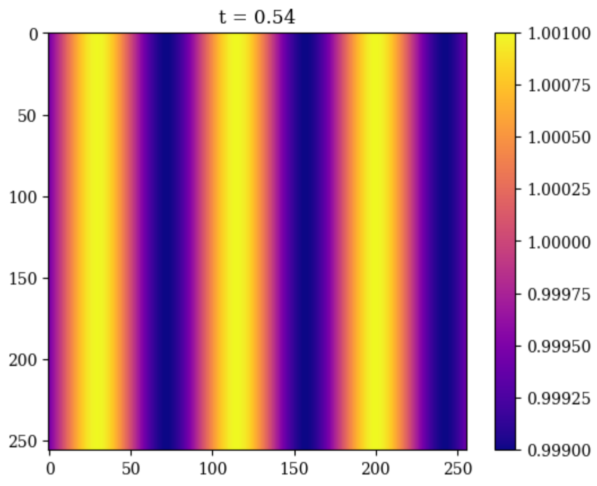
`sound_wave.cpp` utilizes `fluid_solver_2d.h` to produce `.csv` files describing the motion of a sound wave that is moving to the right. We wrote the file `plot_soundwave.py` to produce a graph for each `.csv` file. We then compiled these graphs into a short video, `sound_wave.mp4`. In order to check whether our header file was functioning correctly, we compared the velocity of our sound wave to its expected velocity. We used the following equation to calculate our expected velocity and receive a value of $1.29 m/s^2$.

$$c = \frac{\gamma P_0}{\rho_0} \tag{5}$$

We use the following graphs to approximate v_x.



(a) Sound wave at $t = 0.08s$



(b) Sound wave at $t = 0.54s$

We use the approximation $v_x \equiv \frac{\Delta x}{\Delta t}$ In the first image, we can identify that the vertical bar falls around $x = 50$ at $t = 0.08s$. Then, in the second image, the same vertical bar falls at $x = 200$ at

0.54s. So, we can write:

$$v_x = \frac{200-50}{0.54-0.08} = 1.27m/s^2 \tag{6}$$

We see a 0.02 difference between our expected velocity and our actual velocity. We can utilize the percent error formula to better describe the error in simulation.

$$\delta = |\frac{v_{actual} - v_{expected}}{v_{actual}}| \cdot 100\% \tag{7}$$

We find that $\delta = 1.55\%$ and conclude that the error in our simulation is minimal. Overall, our simulated value is very close to the theoretical value. In fact, some of the difference between the two may not even be due to the simulation but may be due to human observation and estimation.

11 Kelvin-Helmholtz Instability

Having confirmed the effectiveness of our fluid solver with the simple sound wave case, we can now attempt to solve for the motion of a fluid in the case of the Kelvin-Helmholtz instability. Our implementation can be found in the file `kelvin_helmholtz.cpp` We began with the parameters set to the following values:

```
double a = 0.45;
double b =0.55;
double rho_d = 4.0;
double rho_0 = 1;
double v0 = 0.5;
double P0= 2.5;
double k = 6* M_PI;
double v_small =0.01;
double sigma = 0.05;
```

Under these parameters, we plotted the output ρ values using the file `plot_kevin.py` and saw the following behavior (during the process of this project, we affectionately dubbed the Kelvin-Helmholtz simulation "Kevin" for brevity).

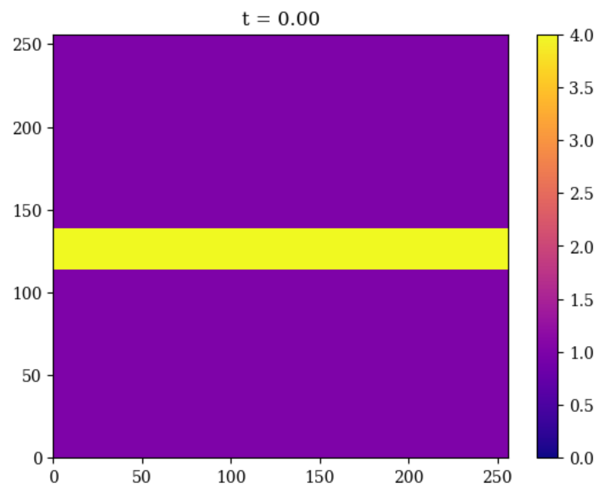


Figure 11.1: The Kelvin-Helmholtz instability at $t = 0.0s$

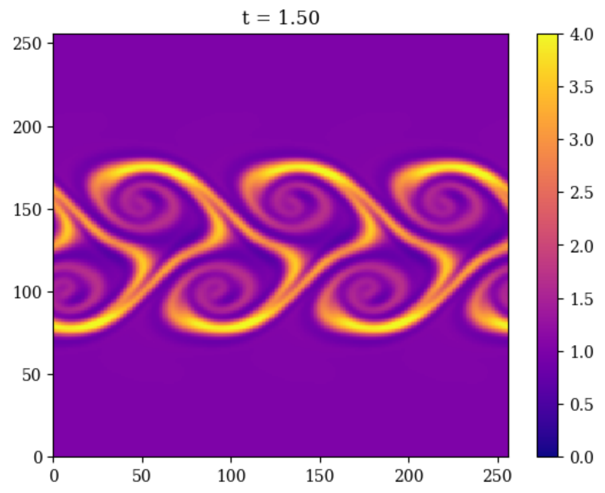


Figure 11.2: The Kelvin-Helmholtz instability at $t = 01.50s$

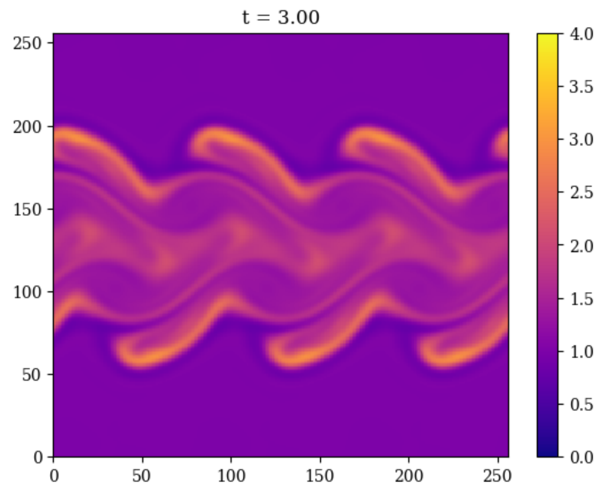


Figure 11.3: The Kelvin-Helmholtz instability at $t = 3.00s$

The full simulation can be found in the file `kevin.mp4`. Parameters a and b define the width of the high density region. The larger the difference between a and b is, the wider the high density stripe will be. When the stripe is very thick, the instability happens at the edges and does not penetrate fully into the high density region, as shown.

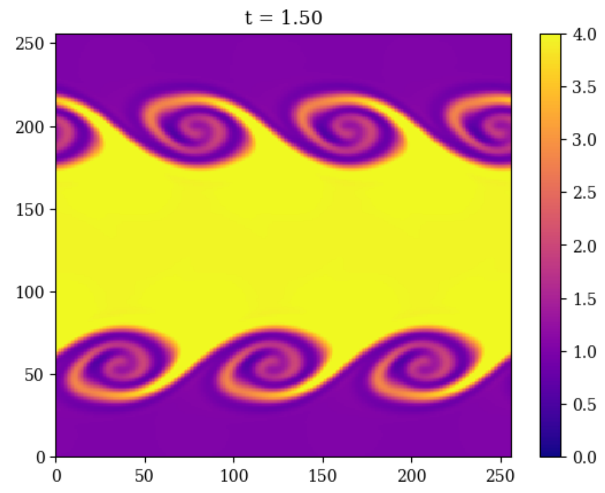


Figure 11.4: The Kelvin-Helmholtz simulation with $a = 0.25$ and $b = 0.75$

Parameters ρ_0 and ρ_d represent the initial densities in the low density and high density regions respectively. When we set these densities to be closer to each other, so $\rho_d = 1.5$, we see the following behavior:

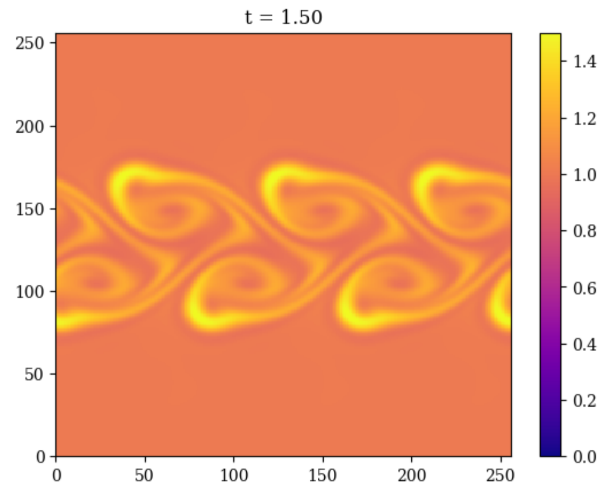


Figure 11.5: The Kelvin-Helmholtz simulation with $\rho_0 = 1$ and $\rho_d = 1.5$ at $t = 1.5s$

At $t = 0$, the figure looks nearly the same as Figure 11.1 except with a lower density background. When we tried to increase the difference between ρ_0 and ρ_d, we found that our simulation was unable to capture a high density gradient. We tried to change our C_{CFL} condition and increase v_0 in order to provide a more diffusive scheme. However, this was unsuccessful and our simulation was unable to run.

Next, we changed the velocity parameter, `v0`. By changing this value, we changed the velocity in the x-direction at which the fluids were moving. The two fluids are moving in opposite directions, so it increased both fluids by a magnitude while keeping their respective directions the same. We found that this greatly increased the speed of our simulation as some of the characteristic features of the simulation appeared much quicker. The full simulation of this can be found in the file `fast_kevin.mp4`.

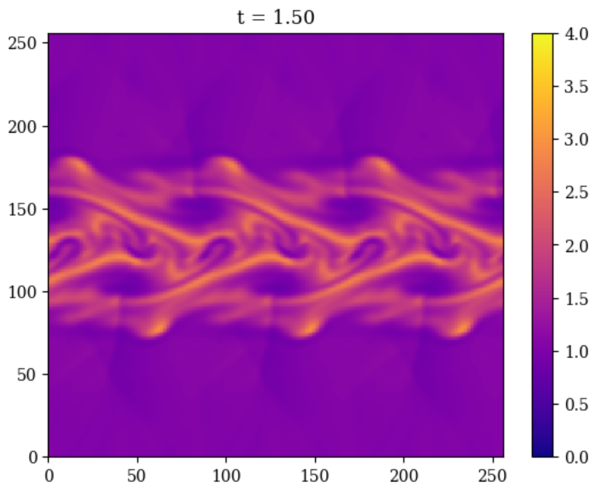


Figure 11.6: The Kelvin-Helmholtz simulation with $v_0 = 1.5$ at $t = 1.5$

The parameter σ enhances the spiral structure that appears in the simulation. By increasing σ the structure became thinner and more defined.

The parameter k controls the number of spiral structures that are produced. It also appears to speed up the rate at which the structures form and instability is reached. The structure that appears in Figure 11.2 appears in this new simulation, except at a much earlier time and with more spiral structures present. The full simulation can be viewed in the file `lots_of_kevin.mp4`.

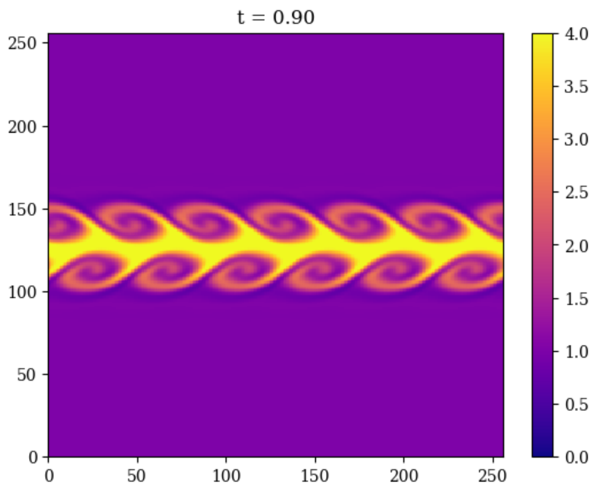


Figure 11.7: The Kelvin-Helmholtz instability when $k = 12\pi$ at $t = 0.90$

Finally, we tested a few different C_{CFL}. We found that that the largest C_{CFL} condition that we could use without reaching instability was $C_{CFL,max} = 0.64$. The graphs that were produced by this C_{CFL} condition were identical to the naked eye to the graphs produced by $C_{CFL} = 0.4$ (Figure 11.2).