

You have **2** free member-only stories left this month. [Sign up for Medium and get an extra one](#)

Create Your Own Finite Volume Fluid Simulation (With Python)



Philip Mocz

Follow

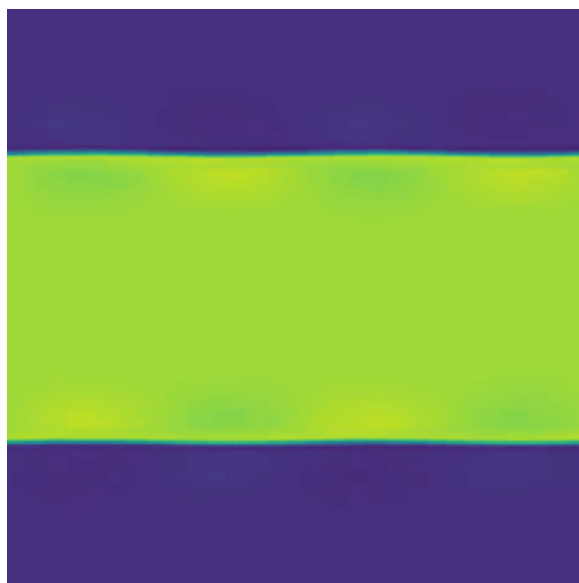


Oct 1, 2020 · 9 min read ★

For today's recreational coding exercise, we will simulate the **Kelvin-Helmholtz Instability** with the Finite Volume method. We will consider a compressible fluid with a high density stream moving in opposite direction of the background. The velocity shear induces a famous instability that is seen sometimes in clouds as well as Jupiter's Great Red Spot.

You may find the accompanying [Python code on github](#).

Before we begin, below is a gif of what running our simulation looks like:



Finite Volume Method

We will describe the finite volume method to simulate an ideal compressible fluid. Extensions of the method exist for the simulation of other types of fluids. An ideal compressible fluid is described by the **Euler fluid equations**. For purposes of this discussion, let us consider the system in 2D (it is not too difficult to extend what is presented here to 3D). The fluid is described by what are called **primitive variables**:

- Density ρ
- Velocity v_x, v_y
- Pressure P

The equations that evolve these parameters in time can be written as the Euler equations in **primitive form**:

$$\frac{\partial}{\partial t} \begin{pmatrix} \rho \\ v_x \\ v_y \\ P \end{pmatrix} + \begin{pmatrix} v_x & \rho & 0 & 0 \\ 0 & v_x & 0 & 1/\rho \\ 0 & 0 & v_x & 0 \\ 0 & \gamma P & 0 & v_x \end{pmatrix} \frac{\partial}{\partial x} \begin{pmatrix} \rho \\ v_x \\ v_y \\ P \end{pmatrix} + \begin{pmatrix} v_y & 0 & \rho & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_y & 1/\rho \\ 0 & 0 & \gamma P & v_y \end{pmatrix} \frac{\partial}{\partial y} \begin{pmatrix} \rho \\ v_x \\ v_y \\ P \end{pmatrix} = 0$$

Here γ is the ideal gas adiabatic index parameter. For example, a monatomic ideal gas has $\gamma=5/3$. The parameter appears in the calculation of the local fluid soundspeed c :

$$c = \sqrt{\gamma \frac{P}{\rho}}$$

The Finite Volume method will actually primarily use a different form of the Euler equations. In an ideal fluid, the total mass, momentum, and energy is conserved and the Finite Volume method aims to ensure this. It will make sense to define **conservative variables**, which are:

- Mass Density ρ
- Momentum Density $\rho v_x, \rho v_y$
- Energy Density ρe

The energy density relates to the pressure through the fluid's **equation of state**:

$$P = (\gamma - 1)\rho u$$

where u is the internal energy (that is, temperature) of the fluid, which relates to the total energy e as:

$$e = u + (v_x^2 + v_y^2)/2$$

(this expression is the sum of the internal energy and the kinetic energy of the bulk fluid motion)

The Euler equations can be re-written in **conservative form**:

$$\frac{\partial}{\partial t} \begin{pmatrix} \rho \\ \rho v_x \\ \rho v_y \\ \rho e \end{pmatrix} + \frac{\partial}{\partial x} \begin{pmatrix} \rho v_x \\ \rho v_x^2 + P \\ \rho v_x v_y \\ (\rho e + P)v_x \end{pmatrix} + \frac{\partial}{\partial y} \begin{pmatrix} \rho v_y \\ \rho v_y v_x \\ \rho v_y^2 + P \\ (\rho e + P)v_y \end{pmatrix} = 0$$

The reason this set of equations is said to be in conservative form is because it has the general structure:

$$\frac{\partial}{\partial t} \mathbf{U} + \nabla \cdot \mathbf{F}(\mathbf{U}) = 0$$

where \mathbf{U} is the set of conservative variables, and \mathbf{F} is the flux function. The time evolution of conservative variables evolves as the divergence of the flux.

On a computer, the fluid is discretized into individual fluid elements (square 'cells') of size Δx by Δx . The cells exchange conservative quantities via **fluxes** through cell

interfaces. Since cells are only exchanging quantities and not gaining or losing it through other means, their total sum always remains the same. This is a very nice property for the numerical method to have.

To see how this works, let us define each cell i 's mass, momentum, and energy by multiplying by the cell volume $(\Delta x)^2$:

$$\mathbf{Q}_i = \mathbf{U}_i \times (\Delta x)^2$$

Then, the heart of the Finite Volume Method lies in calculating fluxes across cell interfaces. We will flesh out the details of how fluxes may be computed in practice, but for now let us assume that they can be obtained. Once obtained, the solution can be updated from timestep n to the next timestep $n+1$ as:

$$\mathbf{Q}_i^{(n+1)} = \mathbf{Q}_i^{(n)} - \Delta t \Delta x \sum_j \hat{F}_{ij}^{(n+1/2)}$$

where F_{ij} refers to the numerical flux between neighboring cells i and j . In general, the flux is calculated as a function of the fluid variables on the 'Left' and 'Right' side of the interface. The sum is taken over all 4 neighbors (in 2D). The Δx factor is the area of an interface. The derivation of this simple equation comes from the Divergence Theorem in calculus.

Converting between Primitive and Conservative Variables

Let's dive into the details of implementing a simple yet powerful Finite Volume code in Python. We will represent the solution variables as a matrix, and perform operations on matrices to avoid slowing down the Python code with For-loops.

It seems useful to define a function that translates primitive variables to conserved quantities \mathbf{Q}

```
1 def getConserved( rho, vx, vy, P, gamma, vol ):  
2     """  
3     Calculate the conserved variable from the primitive  
4     rho      is matrix of cell densities  
5     vx      is matrix of cell x-velocity  
6     vy      is matrix of cell y-velocity  
7     P       is matrix of cell pressures  
8     gamma   is ideal gas gamma  
9     vol     is cell volume  
10    Mass    is matrix of mass in cells  
11    Momx    is matrix of x-momentum in cells  
12    Momy    is matrix of y-momentum in cells  
13    Energy  is matrix of energy in cells  
14    """  
15    Mass = rho * vol  
16    Momx = rho * vx * vol  
17    Momy = rho * vy * vol  
18    Energy = (P/(gamma-1) + 0.5*rho*(vx**2+vy**2))*vol  
19  
20    return Mass, Momx, Momy, Energy
```

getConserved.py hosted with ❤ by [GitHub](#)

[view raw](#)

and vice-versa:

```

1  def getPrimitive( Mass, Momx, Momy, Energy, gamma, vol ):
2      """
3      Calculate the primitive variable from the conservative
4      Mass      is matrix of mass in cells
5      Momx      is matrix of x-momentum in cells
6      Momy      is matrix of y-momentum in cells
7      Energy    is matrix of energy in cells
8      gamma     is ideal gas gamma
9      vol       is cell volume
10     rho       is matrix of cell densities
11     vx        is matrix of cell x-velocity
12     vy        is matrix of cell y-velocity
13     P         is matrix of cell pressures
14     """
15     rho = Mass / vol
16     vx  = Momx / rho / vol
17     vy  = Momy / rho / vol
18     P   = (Energy/vol - 0.5*rho * (vx**2+vy**2)) * (gamma-1)
19
20     return rho, vx, vy, P

```

getPrimitive.py hosted with ❤ by [GitHub](#)

[view raw](#)

Setting the Timestep with the CFL condition

For numerical stability and accuracy, the simulation timestep cannot be arbitrarily large. It must obey the Courant-Friedrichs-Lewy (CFL) condition:

$$\Delta t = C_{\text{CFL}} \times \min_i \left(\frac{\Delta x}{c_i + |v_i|} \right)$$

where $C_{\text{CFL}} \leq 1$ is an order unity constant. The minimum in the expression for the timestep is taken over all the cells i . The speed $c_i + |v_i|$ is a proxy for the maximum signal speed in a cell. Conceptually, what the CFL condition says is that in the duration of a timestep, the max signal speed may not travel more than the length of a cell.

Calculating Gradients

In our Finite Volume representation, we keep track of and evolve cell-centered fluid variables. In the calculation of fluxes, we need to know fluid variable values on the ‘Left’ and ‘Right’ sides of interfaces. Of course, we may approximate the face values just by taking the cell-centered values. But we can do better. We can measure spatial gradients

and use them to extrapolate variables from cell centers to faces. This yields a higher-order method, which is more accurate and less numerically diffusive.

So let us construct a function that calculates the gradient of an arbitrary field f . We will use this function to calculate gradients of primitive variables later, but let's just construct a general method for now. We can use the second-order finite difference formula:

$$\left\{ \frac{\partial}{\partial x} f_{i,j}, \frac{\partial}{\partial y} f_{i,j} \right\} \simeq \left\{ \frac{f_{i+1,j} - f_{i-1,j}}{2\Delta x}, \frac{f_{i,j+1} - f_{i,j-1}}{2\Delta x} \right\}$$

where in this equation cells are indexed in a logical fashion indicating their location in a 2D grid according to a pair of indices i,j .

Below is the function that calculates the gradient on a periodic domain in a vectorized fashion (that is, acting on a matrix of variables all at once). Note the use of the `np.roll()` function, which is an efficient way to periodically shift around matrix elements along the x or y directions in order to look up neighboring values.

```

1  def getGradient(f, dx):
2      """
3      Calculate the gradients of a field
4      f          is a matrix of the field
5      dx         is the cell size
6      f_dx       is a matrix of derivative of f in the x-direction
7      f_dy       is a matrix of derivative of f in the y-direction
8      """
9      # directions for np.roll()
10     R = -1    # right
11     L = 1     # left
12
13     f_dx = ( np.roll(f,R,axis=0) - np.roll(f,L,axis=0) ) / (2*dx)
14     f_dy = ( np.roll(f,R,axis=1) - np.roll(f,L,axis=1) ) / (2*dx)
15
16     return f_dx, f_dy

```

`getGradient.py` hosted with ❤ by [GitHub](#)

[view raw](#)

Slope Limiters

We will not get into the advanced topic of slope limiters here, but just briefly mention

that sometimes a fluid exhibits discontinuities in the fluid variables, which are called **shocks**. This can happen when you have supersonic motions, which is common in astrophysics. In this case, the gradient is not well-defined and measuring it can lead to numerical artifacts. We will not need slope limiters in our example, but I make note that in general it is possible to identify interfaces where discontinuities may exist, in which case the slope often gets set to 0 or some reduced value that prevents small-scale oscillations in the reconstructed field from cell-centers.

Second-Order Extrapolation in Space

We mentioned that we would like to look up fluid variables at on the ‘Left’ and ‘Right’ sides of cell faces for the flux calculation. The simplest first-order approach is to just take the cell-centered values on the left and right of the interface. But with estimated gradients, we can use that information to extrapolate values along a distance $\Delta x/2$ from the cell center to a face. For example, spatially extrapolating from a cell i,j to the face $(i+1/2,j)$ on its right is accomplished as:

$$f_{i+1/2,j} \simeq f_{i,j} + \frac{\partial}{\partial x} f_{i,j} \times \frac{\Delta x}{2}$$

The following function performed spatial extrapolation on an arbitrary field to each of the 4 faces of a cell.


```

1  def extrapolateInSpaceToFace(f, f_dx, f_dy, dx):
2      """
3      Calculate the gradients of a field
4      f          is a matrix of the field
5      f_dx       is a matrix of the field x-derivatives
6      f_dy       is a matrix of the field y-derivatives
7      dx         is the cell size
8      f_XL       is a matrix of spatial-extrapolated values on `left' face along x-axis
9      f_XR       is a matrix of spatial-extrapolated values on `right' face along x-axis
10     f_YR       is a matrix of spatial-extrapolated values on `left' face along y-axis
11     f_YR       is a matrix of spatial-extrapolated values on `right' face along y-axis
12     """
13     # directions for np.roll()
14     R = -1     # right
15     L = 1     # left
16
17     f_XL = f - f_dx * dx/2
18     f_XL = np.roll(f_XL, R, axis=0)
19     f_XR = f + f_dx * dx/2
20
21     f_YL = f - f_dy * dx/2
22     f_YL = np.roll(f_YL, R, axis=1)
23     f_YR = f + f_dy * dx/2
24
25     return f_XL, f_XR, f_YL, f_YR

```

It turns out that in general it is better to extrapolate primitive variables and convert back to conservative, rather than extrapolate conservative variables directly, in order to ensure the pressure does not accidentally get reconstructed to negative values due to truncation errors.

Second-Order Extrapolation in Time

In addition to extrapolating in space, to make the method more accurate it is useful to extrapolate in time by half a timestep before calculating the fluxes. This is done by expressing the time gradient simply as a function of spatial gradients (which we know) using the primitive form of the Euler equations.

We will include code that does this in the simulation main loop that we will present soon. Do not worry if the details do not yet make sense. We are sketching out the big picture and main steps and will fill in the details soon.

Calculating and Applying Fluxes

The heart of the Finite Volume lies in calculating the numerical flux, given a fluid state u (expressed as a collection of conservative variables) on the 'Left' (L) and 'Right' (R) sides of the interface. This may be done in a number of ways, with different levels of accuracy (and some may be better at handling shocks than others). Here we describe a simple robust approximation called the **Rusanov flux**. The flux is:

$$\hat{F} = \frac{1}{2} (F_L + F_R) - \frac{c_{\max}}{2} (u_R - u_L)$$

The first term is a simple average of the fluxes as derived from the left or the right fluid variables. Then, there is an added term which creates numerical diffusivity. It keeps the solution numerically stable. c_{\max} is the maximum signal speed. Advanced versions of flux solvers exist which solve strong shock structures more accurately with less numerical diffusivity, but for our purposes here the Rusanov flux will suffice.

Below is code that calculates the Rusanov flux.

```

1  def getFlux(rho_L, rho_R, vx_L, vx_R, vy_L, vy_R, P_L, P_R, gamma):
2      """
3      Calculate fluxed between 2 states with local Lax-Friedrichs/Rusanov rule
4      rho_L      is a matrix of left-state density
5      rho_R      is a matrix of right-state density
6      vx_L      is a matrix of left-state x-velocity
7      vx_R      is a matrix of right-state x-velocity
8      vy_L      is a matrix of left-state y-velocity
9      vy_R      is a matrix of right-state y-velocity
10     P_L        is a matrix of left-state pressure
11     P_R        is a matrix of right-state pressure
12     gamma      is the ideal gas gamma
13     flux_Mass   is the matrix of mass fluxes
14     flux_Momx   is the matrix of x-momentum fluxes
15     flux_Momy   is the matrix of y-momentum fluxes
16     flux_Energy is the matrix of energy fluxes
17     """
18
19     # left and right energies
20     en_L = P_L/(gamma-1)+0.5*rho_L * (vx_L**2+vy_L**2)
21     en_R = P_R/(gamma-1)+0.5*rho_R * (vx_R**2+vy_R**2)
22
23     # compute star (averaged) states
24     rho_star = 0.5*(rho_L + rho_R)
25     momx_star = 0.5*(rho_L * vx_L + rho_R * vx_R)
26     momy_star = 0.5*(rho_L * vy_L + rho_R * vy_R)
27     en_star = 0.5*(en_L + en_R)
28
29     P_star = (gamma-1)*(en_star-0.5*(momx_star**2+momy_star**2)/rho_star)
30
31     # compute fluxes (local Lax-Friedrichs/Rusanov)
32     flux_Mass = momx_star
33     flux_Momx = momx_star**2/rho_star + P_star
34     flux_Momy = momx_star * momy_star/rho_star
35     flux_Energy = (en_star+P_star) * momx_star/rho_star
36
37     # find wavespeeds
38     C_L = np.sqrt(gamma*P_L/rho_L) + np.abs(vx_L)
39     C_R = np.sqrt(gamma*P_R/rho_R) + np.abs(vx_R)
40     C = np.maximum( C_L, C_R )
41
42     # add stabilizing diffusive term
43     flux_Mass -= C * 0.5 * (rho_L - rho_R)
44     flux_Momx -= C * 0.5 * (rho_L * vx_L - rho_R * vx_R)
45     flux_Momy -= C * 0.5 * (rho_L * vy_L - rho_R * vy_R)
46     flux_Energy -= C * 0.5 * ( en_L - en_R )
47


```

Once the fluxes are computed, they can be applied to the conserved fluid quantities Q in each cell

```

1  def applyFluxes(F, flux_F_X, flux_F_Y, dx, dt):
2      """
3      Apply fluxes to conserved variables
4      F          is a matrix of the conserved variable field
5      flux_F_X is a matrix of the x-dir fluxes
6      flux_F_Y is a matrix of the y-dir fluxes
7      dx          is the cell size
8      dt          is the timestep
9      """
10     # directions for np.roll()
11     R = -1 # right
12     L = 1  # left
13
14     # update solution
15     F += - dt * dx * flux_F_X
16     F +=  dt * dx * np.roll(flux_F_X, L, axis=0)
17     F += - dt * dx * flux_F_Y
18     F +=  dt * dx * np.roll(flux_F_Y, L, axis=1)
19
20     return F

```

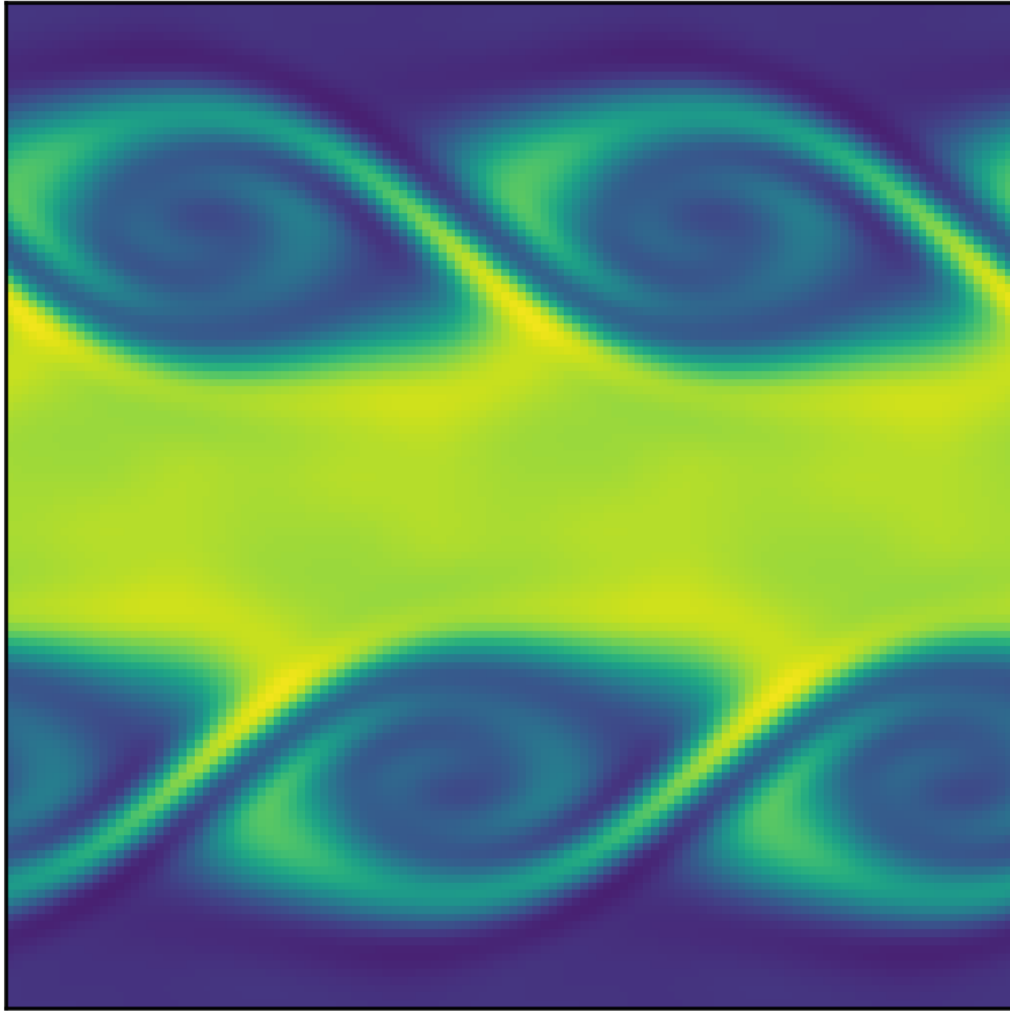
[Running the code](#) is hosted with  by [GitHub](#) to visualize the simulation in real time and will yield the figure:

Time Integration

Let's put all the details together by fleshing out the time integration main loop. At each timestep, the Finite Volume method will:

- Get cell-centered primitive variables from conservative variables
- Calculate the next timestep Δt
- Calculate gradients of primitive variables
- Extrapolate primitive variables in time by $\Delta t/2$ using gradients
- Extrapolate primitive variables to faces using gradients
- Feed in face Left and Right fluid states to compute the fluxes across each face

- Update the solution by applying fluxes to the conservative variables

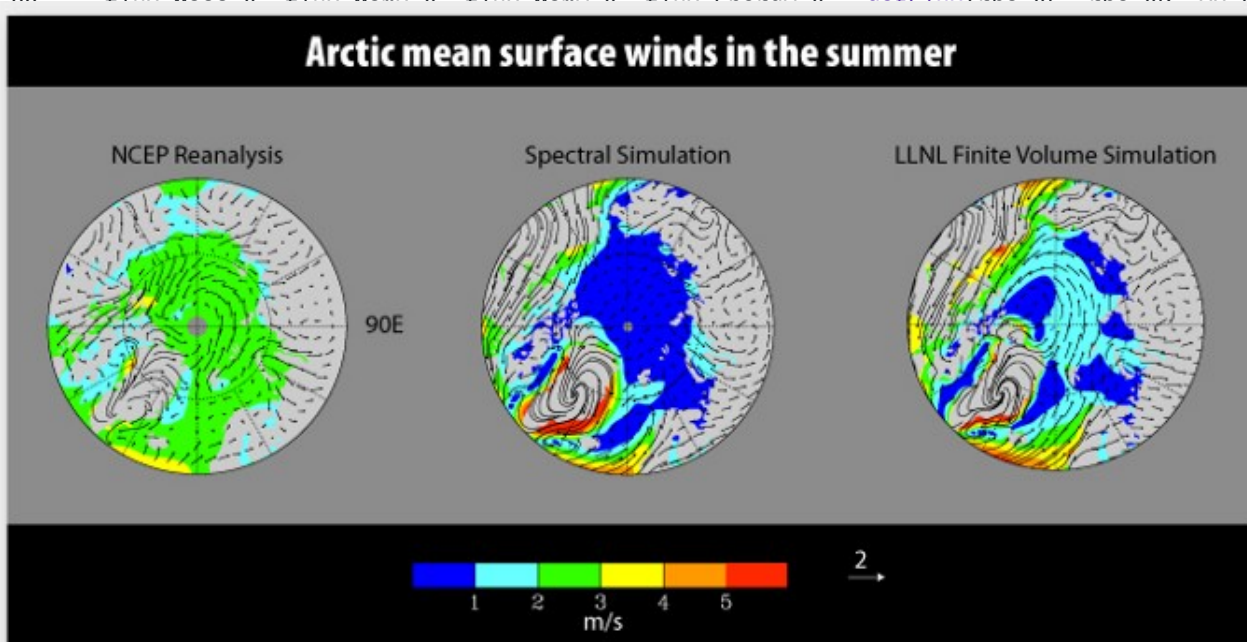


The Finite Volume method is a very powerful tool in computational astrophysics. It is particularly a good formulation to capture gas that is highly compressible, with shocks and negligible viscosity. Much of the interstellar and intergalactic gas is in this physical regime. Below is a simulation of compressible turbulence with the Finite Volume method.

Compressible, supersonic turbulence at Mach 10



```
23 rho_XL, rho_XR, rho_YL, rho_YR = extrapolateInSpaceToFace(rho_prime, rho_dx, rho_dy, dx)
24 vx_XL, vx_XR, vx_YL, vx_YR = extrapolateInSpaceToFace(vx_prime, vx_dx, vx_dy, dx)
25 vy_XL, vy_XR, vy_YL, vy_YR = extrapolateInSpaceToFace(vy_prime, vy_dx, vy_dy, dx)
26 P_XL, P_XR, P_YL, P_YR = extrapolateInSpaceToFace(P_prime, P_dx, P_dy, dx)
27
28 # compute fluxes (local Lax-Friedrichs/Rusanov)
```



Initial Conditions Climate simulations of surface wind circulation from [Lawrence Livermore National Lab](#)

Our domain in this simulation is assumed to be 2-dimensional and periodic. The code specifies the initial primitive variables (density, velocity, pressure fields), and the ideal gas γ parameter. And, as we have mentioned, Jupiter's Great Red Spot is a great example in nature of the Kelvin-Helmholtz Instability that we have simulated with our code tutorial. To set up the Kelvin-Helmholtz instability, the code initializes a high-density region moving to the right and the background moving to the left. Pressure is uniform. A small perturbation is



Jupiter's Great Red Spot. Credits: [NASA/JPL/Space Science Institute](#)

[About](#) [Write](#) [Help](#) [Legal](#)

Get the Medium app

