Lattice Boltzmann (D2Q9) Simulation of Generic Fluids

The following simulation is adapted from a project I did in my introductory computational physics class. My goal here is to develop a simple 2-D fluid simulation of a fluid flowing through a tunnel, with different obstacles inside it.

We are using the lattice boltzmann method, where space is discretized into "voxels", each of which holds some number of particles indicating density. Then these particles are allowed to move in different directions, collide with obstacles, and appear/disappear based on initial and boundary conditions. The simulations have also been animated, which will be shown at the end. Each simulation takes about 20 minutes to be loaded.

In [1]:

```
# Importing Libraries and Setting Up Environment
nx, ny = 520, 180

import numpy as np
import matplotlib.pyplot as plt
import math
from matplotlib.animation import FuncAnimation
import itertools
from IPython.display import HTML
import pickle
def resetMe(keepList=[]):
    ll=%who_ls
    keepList=keepList+['resetMe','np','plt','math','FuncAnimation',
                       'HTML','itertools','pickle','testFunc']
    for iiii in keepList:
        if iiii in ll:
            ll.remove(iiii)
    for iiii in ll:
        jjjj="^"+iiii+"$"
        %reset_selective -f {jjjj}
    ll=%who_ls
    return

def testFunc(func,inFiles,outFiles):
    inputs  = [pickle.load(open(f,"rb")) for f in inFiles]
    outputs = [pickle.load(open(f,"rb")) for f in outFiles]
    result  = func(*inputs)
    allGood = True
    if not isinstance(result, tuple): result = (result,)
    for i in range(len(outputs)):
        if np.max(np.abs(result[i]-outputs[i])) > 1e-14:
            print("Failed test for",outFiles[i],i,
                  np.max(np.abs(result[i]-outputs[i])))
            allGood = False
    if allGood: print("Test Passed!")
    else:       print("Test Failed :(")
```

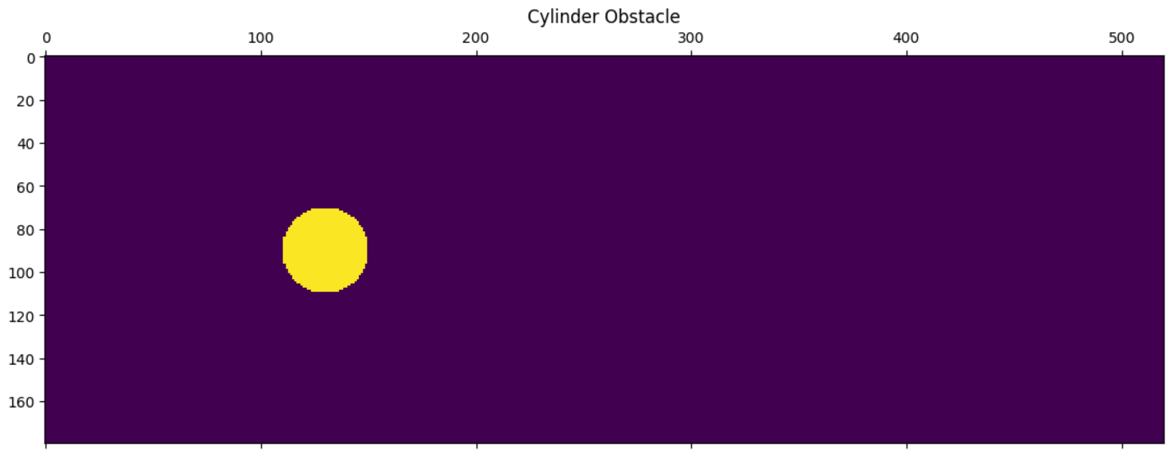
a. Obstacle Construction

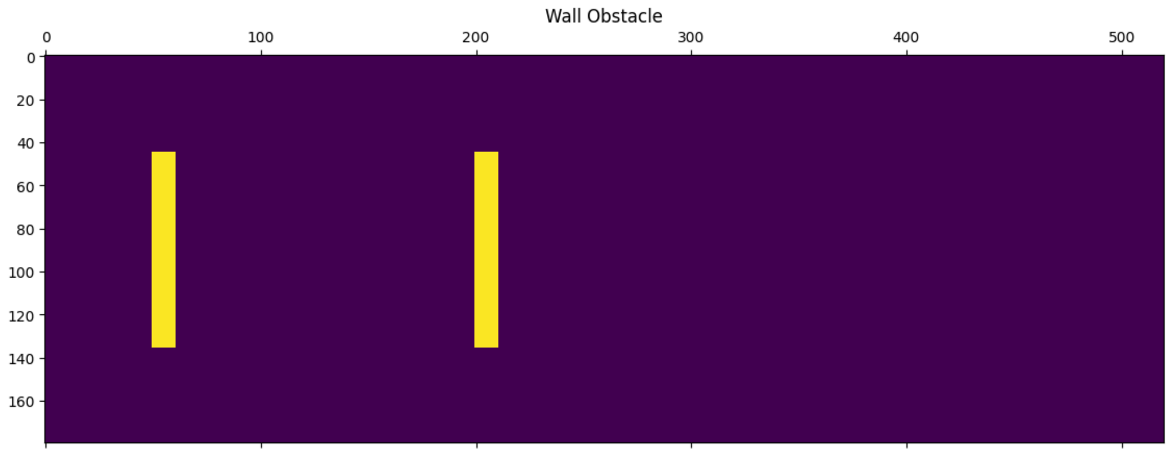
Three obstacles the simulation will run on:

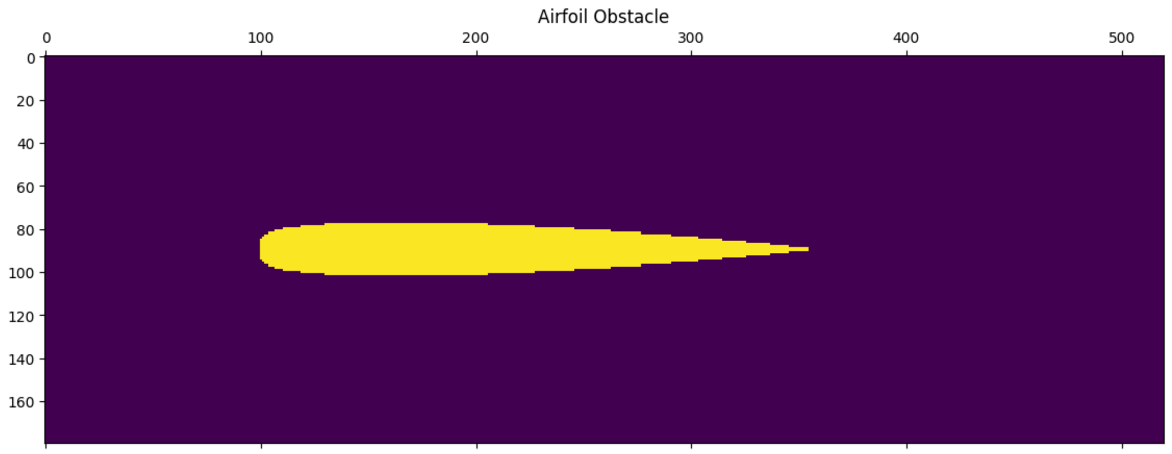
- Circle/Cylinder
- Two Smaller Walls
- Airfoil (airplane wing cross-section)

In [20]:
```
def GenerateCylinderObstacle():
  obs = np.empty((nx,ny), dtype = 'bool')
  obs[:,:] = False
  h, k, r = nx//4, ny//2, ny//9

  for x in range(nx):
    for y in range(ny):
      if ((x-h)**2 + (y-k)**2) < r**2:
        obs[x,y] = True
  return obs

def GenerateWallObstacle():
  obs = np.empty((nx,ny), dtype = 'bool')
  obs[:,:] = False
  obs[50:61, ny//4 : (3*ny//4 + 1)] = True
  obs[200:211, ny//4 : (3*ny//4 + 1)] = True
  return obs

def create_airfoil_mask(grid_size, chord_length, max_thickness):

  grid_width, grid_height = grid_size
  x = np.linspace(0, chord_length, grid_width)
  y = np.linspace(-grid_height/2, grid_height/2, grid_height)
  X, Y = np.meshgrid(x, y)

  # Simple airfoil approximation (NACA 0012)
  sizefactor = X/(0.4*chord_length)
  yt = max_thickness/0.2 * (0.2969*np.sqrt(sizefactor)
                       - 0.1260*(sizefactor)
                       - 0.3516*(sizefactor)**2
                       + 0.2843*(sizefactor)**3
                       - 0.1015*(sizefactor)**4)

  # Create mask
  mask = np.empty((grid_height, grid_width), dtype = 'bool')
  mask[:,:] = False
  mask[np.where(np.abs(Y) <= yt + 5)] = True
  mask = np.roll(mask, 100, axis=1)
  return mask.T
```

In [21]:
```
## Call your function here to see if if works.
plt.matshow(GenerateCylinderObstacle().transpose())
plt.title("Cylinder Obstacle")
plt.show()
plt.matshow(GenerateWallObstacle().transpose())
plt.title("Wall Obstacle")
plt.show()
plt.matshow(create_airfoil_mask((nx, ny), 3, 15).transpose())
plt.title("Airfoil Obstacle")
plt.show()
```







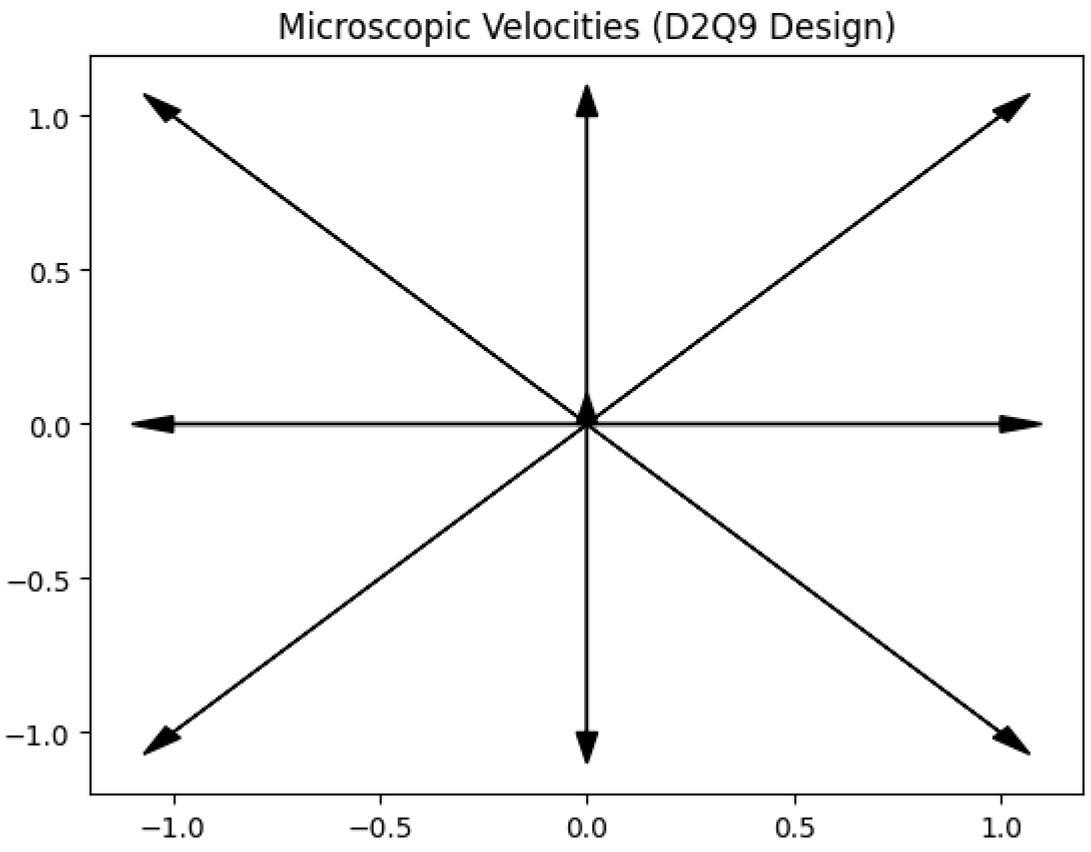
b. Microscopic velocities $v_i$

Here we will define a list of vectors that indicate which directions each particle in a voxel can travel.

In [3]:
```
v = np.empty((9,2), dtype = 'int')
listOfVecs = ([0,0], [0,1], [0,-1], [1,0], [-1,0],
 [-1,-1], [-1,1], [1,-1], [1,1])

for i, vec in enumerate(listOfVecs):
 v[i,:] = vec

print(v)
```

```
[[ 0  0]
 [ 0  1]
 [ 0 -1]
 [ 1  0]
 [-1  0]
 [-1 -1]
 [-1  1]
 [ 1 -1]
 [ 1  1]]
```

In [4]:
```python
for i in range(0,9):
    plt.arrow(0,0,v[i,0],v[i,1],head_width=0.05, head_length=0.1, fc='k',
              ec='k')
plt.xlim(-1.2,1.2)
plt.ylim(-1.2,1.2)
plt.title("Microscopic Velocities (D2Q9 Design)")
#plt.savefig("images/densityDirection.png")
plt.show()
```



c. Computing macroscopic quantities from the microscopic density

To simulate the real world on the computer, we'll break the tunnel into $n_x\times n_y$ squares which we'll call *voxels*.

The key quantity in the simulation is nine microscopic degrees of freedom

- $n_k(i,j)$ where $0\leq k \leq 8$ and $(i,j)$ are over the $n_x \times n_y$ voxels of the simulation

Given the microscopic densities, there are two macroscopic quanties:

- the macroscopic density $\rho(i,j)=\sum_k n_k$ (*size: $n_x \times n_y$*)
- the macroscopic velocity $\vec{u}(i,j) \equiv (u_x(i,j),u_y(i,j))$ (*size: $2 \times n_x \times n_y$*) where
  - $u_x(i,j) = 1/\rho \sum_k v_{k,x} n_k(i,j)$
  - $u_y(i,j) = 1/\rho \sum_k v_{k,y} n_k(i,j)$

In [24]:
```python
n = np.zeros((9, nx, ny), dtype = 'int')

def Micro2Macro(n):
  rho = np.sum(n, axis = 0)

  u = 1/rho * np.array((np.sum(v[:, 0].reshape(9, 1, 1) * n, axis = 0),
                        np.sum(v[:, 1].reshape(9, 1, 1) * n, axis = 0)))

  return rho, u
```

d. Getting the equilibrium microscopic densities.

Given the macroscopic densities $\rho$ and $\vec{u}$, there is a *equilibrium* microscopic densities $n_{eq}$ (*size: $9 \times n_x \times n_y$*) which we compute using `n_eq=Macro2Equilibrium(rho,u)`

The relevant formula for this is

$$ n^{eq}(v_k) \equiv n_k^{eq} = \omega_k \rho \left(1 + 3 \vec{v_k} \cdot \vec{u} + \frac{9}{2}(\vec{v_k} \cdot \vec{u})^2- \frac{3}{2}(\vec{u}\cdot \vec{u}) \right) \tag{A} $$

where

- $\omega_{0}=4/9 $ (stay in place)
- $\omega_{1-4}=1/9$ (up/down/left/right)
- $\omega_{5-8}=1/36$ (corner directions)

Given some microscopic densities $n$ we can figure out the equilibrium microscopic densities $n_{eq}$

In [25]:
```python
def Macro2Equilibrium(rho, u):
  w = np.array([4/9, 1/9, 1/9, 1/9, 1/9, 1/36, 1/36, 1/36, 1/36])
  n_eq = np.zeros((9, nx, ny))

  v_u_term = np.einsum("ik,kab->iab",v,u)
  u_term = u[0]*u[0] + u[1]*u[1]

  n_eq = (w.reshape(9,1,1) * rho.reshape(1, nx, ny) *
   (1 + 3 * v_u_term + 4.5 * v_u_term**2 - 1.5*u_term))

  return n_eq
```

e. Implementing collision

When we collide the microscopic densities, we get new microscopic densities given as

- `nout = n * (1-omega) + omega * neq`

Here are the steps for the function: $$

1. Calculate `neq`
   - Take the microscopic densities $\rightarrow$ compute the macroscopic density and velocity `(rho,u) = Micro2Macro(n)`
   - Take the macroscopic density and velocity $\rightarrow$ compute the equilibrium microscopic densities `neq=Macro2Equilibrium(rho,u)`
2. Calculate `nout = n * (1-omega) + omega * neq` where $\omega =1.9572953736654806$ is the viscosity parameter
3. We moved fluid where the obstacle is, so we need to undo this. Reset all `nout` where the obstacle is to be the same as `n`

In [26]:
```python
def Collision(n, obstacle):
  omega = 1.9572953736654806
  rho, u = Micro2Macro(n)
  neq = Macro2Equilibrium(rho, u)
  nout = (1-omega) * n + omega * neq
  nout[:, obstacle] = n[:, obstacle]
  return nout
```

f. Movement

We'll now move on to getting the fluid to move. This will be a function called `Move(n,obstacle)`. But first we need to helper functions:

- `Bounce(n,obstacle)`
  - bounce the velocities - that is every velocity where the obstacle is gets reversed
- `MoveDensity(n)`
  - Move all your densities in the direction of the velocity (*hint* the velocity tells you where to move it)
  - Assume periodic boundary conditions here Thus a `Move(n,obstacle)` call is just a `Bounce` then `MoveDensity`.

In [27]:
```python
def Bounce(n, obstacle):
  reverse_idx = [0, 2, 1, 4, 3, 8, 7, 6, 5]
  n_b = np.copy(n)
  for k in range(9):
    n_b[k, obstacle] = n[reverse_idx[k], obstacle]

  return n_b

def MoveDensity(n):
  n_moved = np.zeros_like(n)

  for k in range(9):
    n_moved[k] = np.roll(
        np.roll(n[k], shift = v[k][0], axis=0), shift=v[k][1], axis=1)

  return n_moved
```

```
def Move(n, obstacle):
  n_b = Bounce(n, obstacle)
  n_m = MoveDensity(n_b)
  return n_m
```

g. Boundary conditions

In the y-direction, we are just going to assume that the boundary conditions are periodic. For the x-direction, we are going to assign certain boundary conditions on the left and right.

Let's call this function `FixBoundary(n,n_init)` which applies these boundary conditions and returns the new density `n`.

Left: we are going to assume that there is a flow - this means that the microscopic densities are the same at each step. Therefore, what we should do is simply replace the current densities on the left-most row with the initial microscopic densities (`n_init`).

Right: we are going to assume that the gradient is zero - i.e. the important physics has dissapeared by this point. To do this, we will only allow particles on the right boundary that are moving left exist, letting the others disappear out of the simulation.

In [28]:
```
def FixBoundary(n, n_init):
  n[:,0,:] = n_init[:,0,:] # replace all left column with incoming fluids
  for left_v in [4,5,6]: # only let left moving particles go back
    n[left_v, -1, :] = n[left_v, -2, :] # otherwise ignore them

  # all other directions don't matter b/c periodic

  return n
```

h. Setting up the initial conditions

We need to generate an initial conditions. The way that we are going to do that is

- Pick some macroscopic density $\rho$ of size $(n_x,n_y)$ (uniformly equal to 1.0) and a
- Macroscopic velocity $\vec{u}(i,j)$ (sizes $(2,n_x,n_y)$)
  - `0.04*(1.0+1e-4*np.sin(y/(ny)*2*np.pi))` in the x-direction. This introduces a very tiny anisotropy to the system.
  - `0` in the y-direction
- Compute the equilibrium density $n_{eq}$ associated with the initial density and velocities.
- This function is called `Setup()`

In [29]:
```
def Setup():
  rho = np.ones((nx, ny))
  u = np.zeros((2, nx, ny))
  u[0, :, :] = 0.04 * (1.0 + 1e-4 * np.sin(np.arange(ny) / ny * 2 * np.pi
  neq = Macro2Equilibrium(rho,u)
  return neq
```

i. Putting it all together

The function `Run(steps,record,n,n_init,obstacle)` (up in your function-cell) takes as input:

- the number of steps `steps` you want to run
- how often you want to `record` the velocity and density
- the microscopic densities `n` to start running from
- as well as the initial microscopic densities `n_init` for the boundary conditions (often equal to `n.copy()`)
- and the obstacle boolean array.

Each step consists of:

1. Adjust boundary conditions
2. Collide the microscopic densities
3. Move the microscopic densities

And returns

- a list of macroscopic densities `rhos`
- a list of macroscopic velocities `us`
  - This will be `u2=` $\sqrt{\vec{u}\cdot\vec{u}}$
  - To help us later, store these as `u2.transpose()`
- the last microscopic density `n`

To run the simulation, call the `Setup` and `Run` functions. Then plot the final configuration (something like `plt.matshow(us[-1])`).

NOTE: The simulations aren't run in this step, but instead in the next one so that the animations can also be produced.

In [30]:
```
def Run(steps,record,n,n_init,obstacle):
  rhos = []
  us = []
  for time in range(steps):
    n = FixBoundary(n, n_init)
    n = Collision(n, obstacle)
    n = Move(n, obstacle)

    if time % record == 0:
      rho, u = Micro2Macro(n)
      rhos.append(rho)
      us.append(np.sqrt(u[0]**2 + u[1]**2).transpose())

  return rhos, us, n
```

j. Animation

`AnimateMe(us_flat,vMax)` which is going to take a list of velocities `us_flat` (and maximum for the `vMax`) and going to return an animation which is going to be then produced by calling

```
anim = AnimateMe(us,vMax)
HTML(anim.to_jshtml())
```

In [31]:
```
def AnimateMe(us_flat,vMax):
    fig, ax = plt.subplots()
    cax = ax.imshow(us_flat[1],cmap=plt.cm.Reds,vmin=0,vmax=vMax)
    plt.close(fig)
    def animate(i):
         cax.set_array(us_flat[i])

    anim = FuncAnimation(fig, animate, interval=100, frames=len(us_flat))
    return anim
```

Circle/Cylinder Simulation

In [32]:
```
#!Start
fin=Setup()
feq_init=fin.copy()
obstacle=GenerateCylinderObstacle()

fins=rhos=us= [None] * 21
fins[0]=fin.copy()
for i in range(0,20):
    %time (rhos[i],us[i],fins[i+1])=Run(2001,100,fins[i],feq_init,obstacl

us_flat = list(itertools.chain.from_iterable(us[:-1]))
anim=AnimateMe(us_flat,0.07)
HTML(anim.to_jshtml())
#!Stop
```
```
Output hidden; open in https://colab.research.google.com to view.
```

Walls Simulation

In [34]:
```
#!Start
fin=Setup()
feq_init=fin.copy()
obstacle=GenerateWallObstacle()

fins=rhos=us= [None] * 21
fins[0]=fin.copy()
for i in range(0,20):
    %time (rhos[i],us[i],fins[i+1])=Run(2001,100,fins[i],feq_init,obstacl

us_flat = list(itertools.chain.from_iterable(us[:-1]))
anim=AnimateMe(us_flat,0.07)
HTML(anim.to_jshtml())
#!Stop
```
```
Output hidden; open in https://colab.research.google.com to view.
```

Airfoil Simulation

```
In [35]: #!Start
fin=Setup()
feq_init=fin.copy()
obstacle=create_airfoil_mask((nx, ny), 3, 15)

fins=rhos=us= [None] * 21
fins[0]=fin.copy()
for i in range(0,20):
    %time (rhos[i],us[i],fins[i+1])=Run(2001,100,fins[i],feq_init,obstacl

us_flat = list(itertools.chain.from_iterable(us[:-1]))
anim=AnimateMe(us_flat,0.07)
HTML(anim.to_jshtml())
#!Stop
```

```
Output hidden; open in https://colab.research.google.com to view.
```

k. Conclusions

- The flow in the cylinder and walls simulations gets turbulent towards the end, as we can see with the vortices that start to form and emanate out of the simulation.
- The wall simulation in particular seems to be symmetric across the top and bottom. It has two pairs of strong circular vortices form: one in-between the walls and another that flows past the walls to the right.
- The flow in the airfoil simulation, however, is almost devoid of turbulence, which is what we expect/want from the cross-section of a wing. It seems like the density of air around the wing also decreases as the simulation goes on as (as seen by the light colors around the wings).