"H_2S04" : A Gas Simulation

Submission to website: Monday, September 12, 10pm

Checkoff by LA/TA: Tuesday, September 13, 10pm

This lab assumes you have Python 2.7 installed on your machine. Please use the Chrome web browser.

Introduction

Have you watched a film lately? Beautifully complex simulations are everywhere: explosions, wreckage, ocean storms, sparks, and volcanoes, just to name a few. Simulations are tremendously useful: given an initial condition and a set of rules, we can straightforwardly use a computer to evolve the initial condition according to those rules, often giving rise to vastly complex behavior, more complex than we could animate manually, or even understand analytically.

In science and engineering, simulations are often used to study high-order effects of rules such as the laws of physics. Enforcing conservation of energy in a particle model of light yields photo-realistic images (slowly). Applying what little we know of hydrodynamics to 3D models lets us study boat hulls in a variety of weather conditions. We also use simulations to predict the weather, crime, outcomes of sports matches, and nearly everything else one could quantify. These simulations are often computationally complex, requiring a deep understanding of numeric precision, and make simplifying assumptions grounded in mathematics.

Early forms of simulation, cellular automata, sidestepped issues of precision and exponential complexity by discretizing (treating as integer values) time, space, motion, and anything else playing a role in the simulation. Perhaps the most well known of these cellular automaton simulations is Conway's Game of Life, which uses only two trivial rules to govern "life" in a square grid.

In this lab, you will implement a surprisingly capable cellular automaton simulation of a two-dimensional gas represented by an upright square lattice, inspired by the PhD work of Brian Wylie in the late 80s. In this simple 2-dimensional world, time is partitioned into discrete "steps", gas particles occupy discrete locations within the lattice, traveling at unit speeds in exactly one of the four directions: up, left, down, or right. While the real world is of course much more complex, this very simple universe is quite sufficient to give rise to some interesting and complex behavior. In fact, simulations very much like this one were applied to early rocket propulsion systems.

The 6.009 representation of a gas

Our gases are conceptually a 2-dimensional array of cells inrow-major order. For each gas, we'll specify the number of rows as the `height` and the number of columns as the `width`. In Python, it's best to number the rows from `0` to `height-1` and the columns from `0` to `width-1`. With this convention the indices for a 3x4 (`height` =3, `width` =4) 2D array are

```
0,0   0,1   0,2   0,3
1,0   1,1   1,2   1,3
2,0   2,1   2,2   2,3
```

Since Python doesn't have 2D arrays, we'll use a list of cells, listed in left-to-right, top-to-bottom order. So the list representation of the 3x4 2D array would be

```
[  <contents of cell 0,0>,
   <contents of cell 0,1>,
   <contents of cell 0,2>,
   <contents of cell 0,3>,
   <contents of cell 1,0>,
   ...
   <contents of cell 2,2>
   <contents of cell 2,3> ]
```

You should refer to the `width` and `height` parameters when converting a 2D coordinate to the appropriate list index.

Each cell is a *set* (though we use a list of unique elements for simplicity in this lab) of features, which may be a wall or one of four moving particles:

- `"w"` denotes that a wall is present
- `"u"` denotes a particle moving *up* (towards lower row numbers)
- `"r"` denotes a particle moving *right* (towards higher column numbers)
- `"d"` denotes a particle moving *down* (towards higher row numbers)
- `"l"` denotes a particle moving *left* (towards lower column numbers)

No two particles moving in the same direction may occupy the same cell. Each cell in the gas is any subset of the above elements, represented as a list in arbitrary order. For example, a wall with a right-moving particle crashing into it may be represented as `["r", "w"]` or as `["w", "r"]` (The *Simulation Step* section below details why this particle is crashing into the wall instead of moving away to the right from it).

Some examples:

an empty 4x3 gas with no walls or particles:

```
gas_1 = { "width": 3,
          "height": 4,
          "state":  [ [], [], [],
                      [], [], [],
                      [], [], [],
                      [], [], [] ] }
```

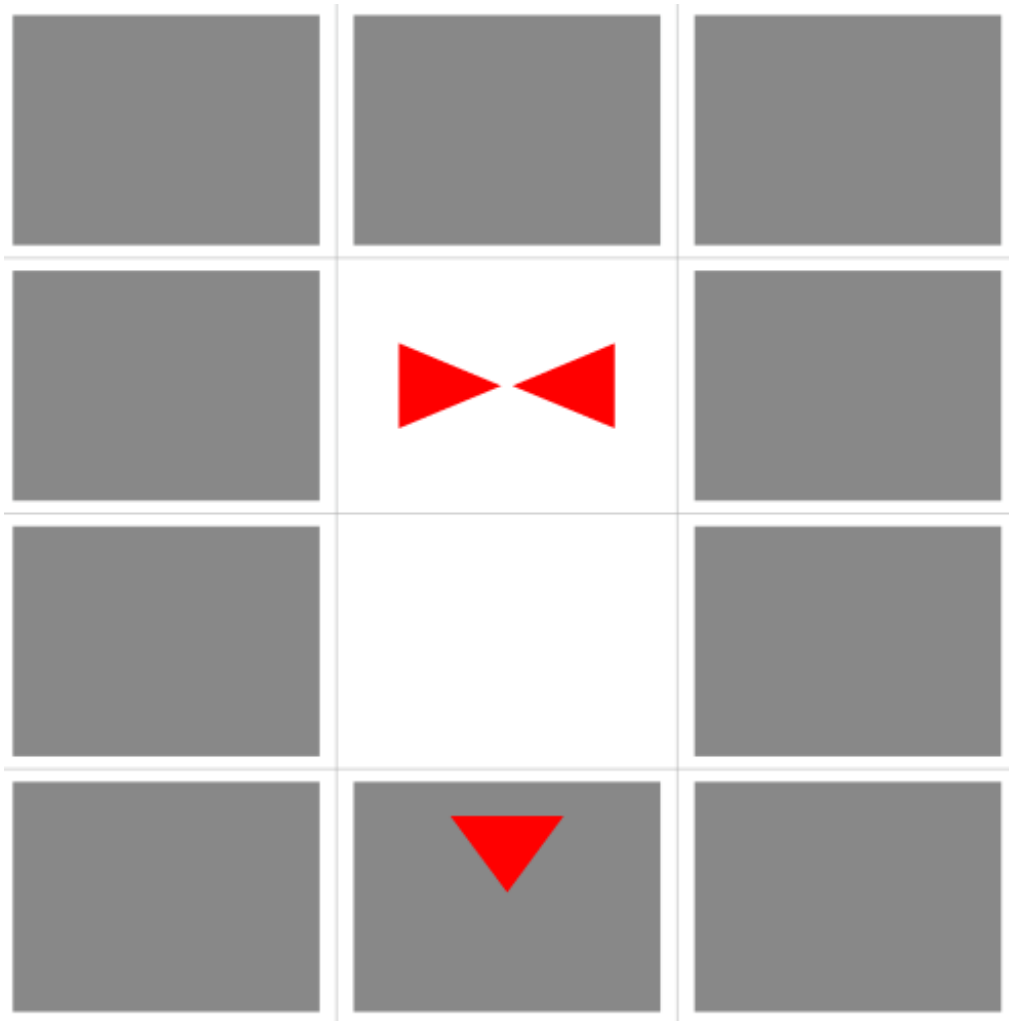
Now let's surround the grid with walls on all sides.

```
gas_2 = { "width": 3,
          "height": 4,
          "state":  [ ["w"], ["w"], ["w"],
                      ["w"], [   ], ["w"],
                      ["w"], [   ], ["w"],
                      ["w"], ["w"], ["w"] ] }
```

Finally, add two particles in the middle, crashing into each other from the right and left. We also add a particle crashing downwards into the bottom wall.

```
gas_3 = { "width": 3,
          "height": 4,
          "state":  [ ["w"],   ["w"],   ["w"],
                      ["w"], ["r","l"], ["w"],
                      ["w"],   [   ],   ["w"],
                      ["w"], ["w","d"], ["w"] ] }
```

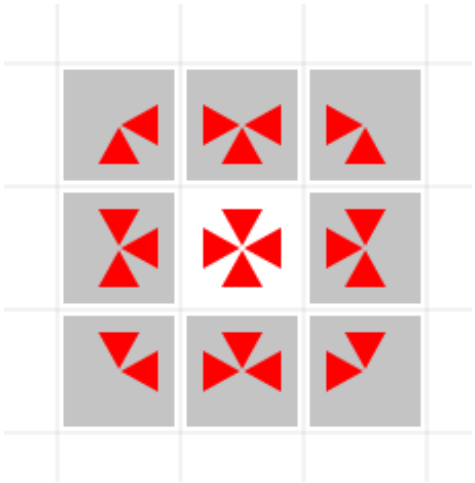
The `server.py` visualization would show `gas_3` (a minimal gas) like this:



A "full" gas would be represented as:

```
{ "width": 5,
  "height": 5,
  "state": [ [],[],[],[],[],
             [],["u","l","w"],["r","u","l","w"],["r","u","w"],[],
             [],["u","l","w","d"],["r","u","d","l"],["r","u","d","w"],[],
             [],["l","w","d"],["r","d","w","l"],["r","d","w"],[],
             [],[],[],[],[] ] }
```

And would be rendered on the server as:



Simulation step

A single function `step(gas)` produces a **gas** according to the rules in this document. To compute the new gas state from the input, three steps are performed:

1. Resolve any collisions with walls.
2. Resolve any collisions among particles.
3. Advance particles according to their directions of motion (propagation).

The steps are described in detail below.

Particle collisions

When colliding with a wall, a particle simply reverses direction. For example, `["r","w"]` becomes `["l","w"]`. Likewise, `["u","w"]` becomes `["d","w"]`, and `["d","l","w"]` becomes `["u","r","w"]`. Note that a wall collision should occur in *any* cell with a "w", independent of the wall's location in the larger diagram.

Collisions between particles are more interesting. In order to preserve momentum, particles must rebound in opposing directions, but the simulation wouldn't be very interesting if the particles simply reversed their directions (indeed, the vertical and horizontal-moving particles would not interact at all). To give rise to some interesting complexity, we rotate head-on collisions by 90 degrees.

A 6.009 gas has *only* two particle collision rules: `["r","l"]` results in `["u","d"]`. Similarly, `["u","d"]` produces `["l","r"]`. No other collision rules are defined in our 6.009 gas. Note that particle-particle collisions happen in addition to any particle-wall collisions. Also remember that ["r","l"] is equivalent to ["l","r"].

What does `["u","d","l","r"]` produce after a collision? There is no magic here, it does not match either of the collision rules, so it remains unchanged, `["u","d","l","r"]`, the same set as the input, and correctly preserves momentum. Three-particle interactions are also straightforward: `["l","r","u"]` does not match either collision rule, and produces `["l","r","u"]`, again unchanged, again correctly preserving momentum.

From the example above, `gas_3` would produce the following intermediate state after a collision (before propagation):

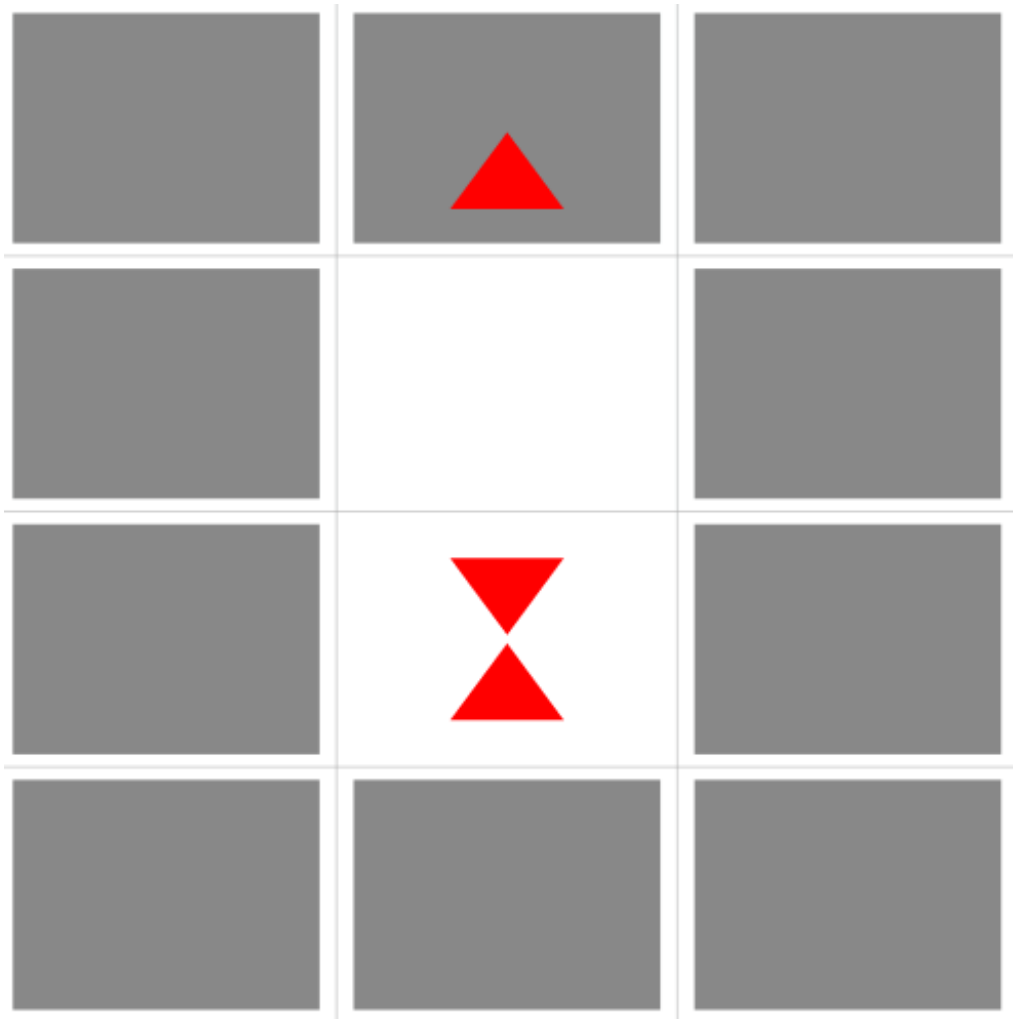
```
["w"],   ["w"],   ["w"]
["w"], ["u","d"], ["w"]
["w"],   [   ],   ["w"]
["w"], ["w","u"], ["w"]
```

Particle propagation

This step is simple. All collisions have already been resolved, so each particle in the gas unconditionally moves one cell in its appropriate direction. **Hint**: collisions can happen "in place" in the sense that collisions in one cell are independent of collisions in other cells. Propagation changes what's in neighboring cells, so you have to be careful that the new gas state is computed using only the positions of the particles in the old gas state. Unless you're clever, mixing the new and old states might result in a particle moving many cells in one step!

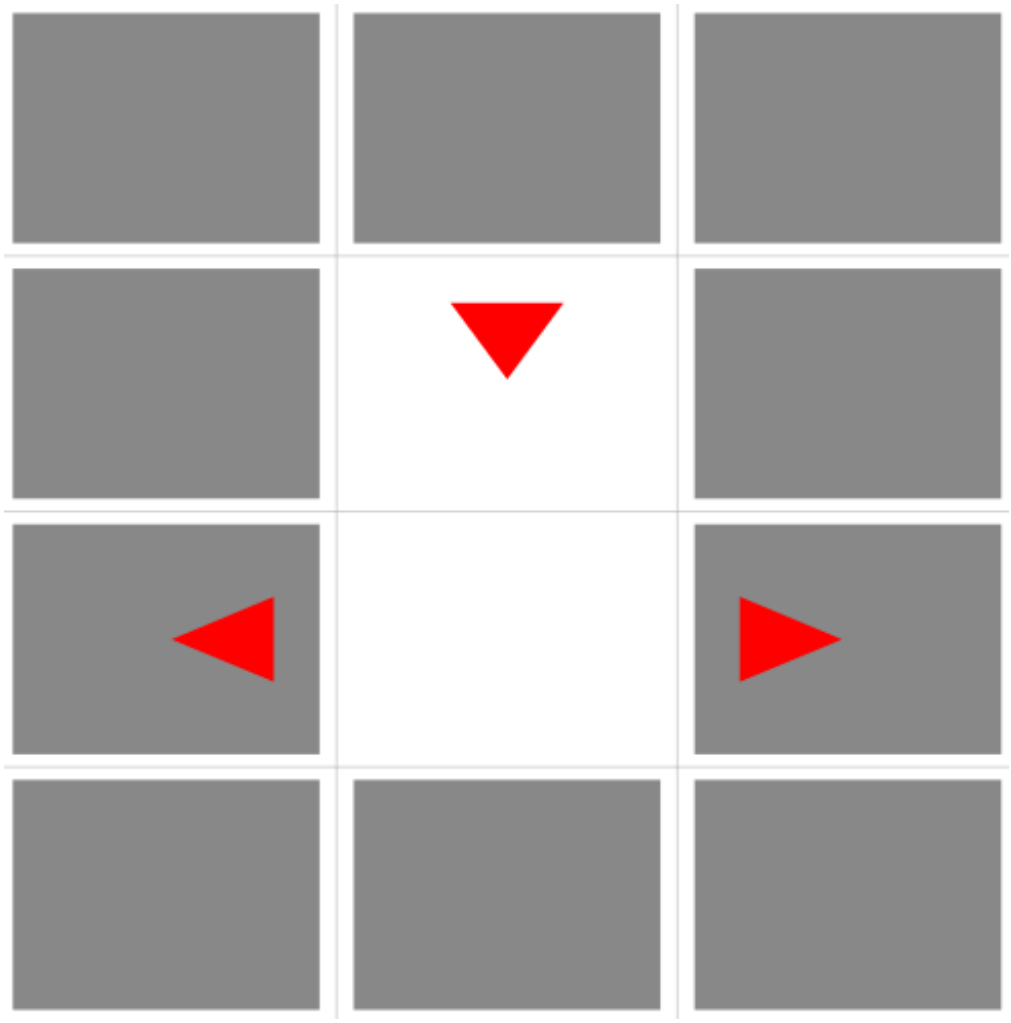
Again, from the example above, `step(gas_3)`, would produce:

```
step(gas_3) = {
          "width": 3,
          "height": 4,
          "state":  [ ["w"], ["w","u"], ["w"],
                      ["w"],    [ ],    ["w"],
                      ["w"], ["d","u"], ["w"],
                      ["w"],   ["w"],   ["w"] ] }
```



The next step would produce

```
step(step(gas_3)) = {
          "width": 3,
          "height": 4,
          "state":  [ ["w"],    ["w"],    ["w"],
                      ["w"],    ["d"],    ["w"],
                      ["w","l"], [ ], ["w","r"],
                      ["w"],    ["w"],    ["w"] ] }
```



Note: particles that move outside the bounds of the gas are forever lost.

`lab.py`

This file is yours to edit in order to complete this lab. You are not expected to read or write any other code provided as part of this lab. In `lab.py`, you will find the interface your solution should support: please correctly implement the method `step` according to this document in order to earn full credit.

Your code will be loaded into a small server (`server.py`) and will serve up a visualization website, which we recommend using to debug your simulation visually. To use the visualization, run `server.py` and use your web browser to navigate to localhost:8000. You will need to restart `server.py` in order to reload your code if you make changes. If you find yourself stuck, **create your own test cases to aid your debugging process**

It's possible to use the Python `import` command in your solution to load other libraries of code, but none of them are officially sanctioned for any 6.009 labs. Arbitrary inconveniences, including our automatic grader giving you no credit for an assignment, may result from importing libraries not already mentioned in the starter `lab.py` file! We've designed these labs to be doable and educational without importing any library modules.

Testing your code

Once your `step` outputs a valid 6.009 gas, you can visualize the output with `server.py` to help you debug your code. The web UI uses all `.gas` gases in `./cases/` in our special 6.009 `.gas` format (it's JSON, a standard textual data format that is compatible with Python). You can generate your own with our handy gas generator: `./create_gas.py --help`. Our web UI will list them alongside the ones we provide!

The visualization shows an animation of gas states produced by your `step`. For gases smaller than `16x16`, the visualization shows each particle in the gas container individually, but larger gases are shown as *density vector fields*, partitioning the simulation into `16x16` *windows* and coloring each proportionally to the ratio of particles to empty space. This simulation also shows a direction of motion for the average direction of particles in each window. Suddenly, this looks like science!

There are two ways to run the simulation:

- Press the "step" button. This advances the simulation by one step.
- Press the "run" button (the one with the triangle on it). This "plays" the simulation freely. To stop it running, press the "pause" button.

To load a different gas, pause the simulation. Click on the "..." button above the gas visualization. A list of gases will pop up, so you can click on the one you want to run next.

Use the `test.py` script to help you **verify** the correctness of your code. You can select which tests run like this: `test.py 1 3 7` runs only tests # 1, 3, and 7. We will only use the provided test cases to auto-grade your work. You may also wish to investigate the Python debugger (PDB) to help you find bugs efficiently.

Does your lab work? Do all tests in `test.py` pass? You're done! Submit your `lab.py` at `fun.csail.mit.edu` and get your lab checked off by a friendly staff member. Consider tackling the bonus section below.

Bonus

- What happens when you mess with the collision and propagation rules? Add a small, random chance for particles to spontaneously turn if they are not at a wall. How does varying this chance affect the behavior of the gas?

- Something a little tougher: Can you think of how you can do the simulation *in place*? That is, do not create a new state structure, but modify the existing one for *both* collisions and propagation.

You may have noticed that our simulation falls flat in some obvious cases, most notably being completely unable to generate circular ripples. A square lattice unfortunately does not have enough symmetries to represent something like a circular ripple, but a hexagonal lattice has many more.

The collision rules in a hexagonal gas are given on page 16 ofthis thesis. The Lab 3 visualization does not know anything about hexagonal lattices, but this article shows how to embed a hexagonal lattice in a square grid, allowing you to use this lab's visualization to show off your hex gas (it will be a bit distorted, but works quite well nontheless). This is all you need to get started. Have fun!