Extended Random Walks, Ideal Gas Law

(download slides and .py files to follow along!)

Tim Kraska

MIT Department Of Electrical Engineering and Computer Science

Inheritance
(from Monday)

Inheriting from `object`

- Every Python type inherits from **`object`**
  - already provides **`__init__()`** , **`__str__()`** , **`__eq__()`**
  - so we can create, print, compare
  - but default behavior may not be what we want
    - **`object.__str__()`** prints memory address
    - **`object.__eq__()`** compares type and memory address
  - so when we define them in our own classes, **they override access to `object`'s attributes**

Inheritance rules for attribute access

`print(critter.age)`

- When looking up attribute on an instance object
  - **If the attribute name is in the instance,** evaluate to the object it references
    - remember: this can evaluate to any **object** in memory, but NOT a name/attribute
  - If not in the instance, **look in that instance's class**
    - this is how method lookups work
    - also works on any class attribute
  - **If not in the class object, look in that class's parent class**
  - Etc...
- The above is applicable only for **evaluating an attribute**
  - When **setting an attribute** for an instance, the attribute is set directly inside the instance, even if the attribute name exists in the class hierarchy

`critter.size = "tiny"`

Using inheritance

- Subclasses can **override** methods of their parent/superclass
  - We already know how to override **object**'s **`__init__()`** and **`__str__()`**
  - **`Cat`** overrides **Animal**'s **`__str__()`**
  - **`Cat`** also overrides **`Animal`**'s **`speak()`**, providing a working implementation
- Subclasses can **reuse** methods of their parent/superclass
  - **`Cat`** relies on **`Animal`**'s **`__init__()`**
  - **`Cat.confuse()`** relies on **Animal**'s **`get_age_diff()`**
- Same applies to class and instance attributes
  - subclass instances can rely on attributes initialized in superclass **`__init__()`**
  - but risky if superclass implementation changes, consider using getters/setters

Retaining and extending superclass functionality

- Sometimes, want to preserve superclass method functionality while extending it
  - simply overriding it would require code duplication
- Strategy: superclass methods still available through explicit ***`superclass.method()`*** reference
  - because not accessing as object's method, requires passing in the object (usually **`self`**) as first argument
- ***Inside a subclass method's body,*** can also use **`super()`** to reinterpret **`self`** as an instance of the superclass
  - **`Animal.__init__(self, age, name)`**
  - **`super().__init__(age, name)`**
  - differing opinions on which is better, but **`super()`** is common

Question

```python
class Rabbit(Animal):

    next_tag = 1

    def __init__(self, age, parent1=None, parent2=None):
        super().__init__(age)
        self.parents = (parent1, parent2)
        self.id = Rabbit.next_tag
        Rabbit.next_tag += 1

    def __str__(self):
        return f"<Rabbit {self.id:>03}>"

    __repr__ = __str__

    def __add__(self, other):
        """Make a new Rabbit offspring of self and other."""
        return Rabbit(0, self, other)

    def __eq__(self, other):
        return (
            self.parents == other.parents
            or self.parents == other.parents[::-1]
        )
```

```python
def demo_rabbit_equality():
    r1 = Rabbit(age=3)
    r2 = Rabbit(age=4)
    r3 = Rabbit(age=5)
    r4 = r1 + r2
    r5 = r3 + r4
    r6 = r4 + r3
    print(f"{(r5 == r6) = }")

demo_rabbit_equality()
```

What does demo_rabbit_equality return?

A) True

B) False

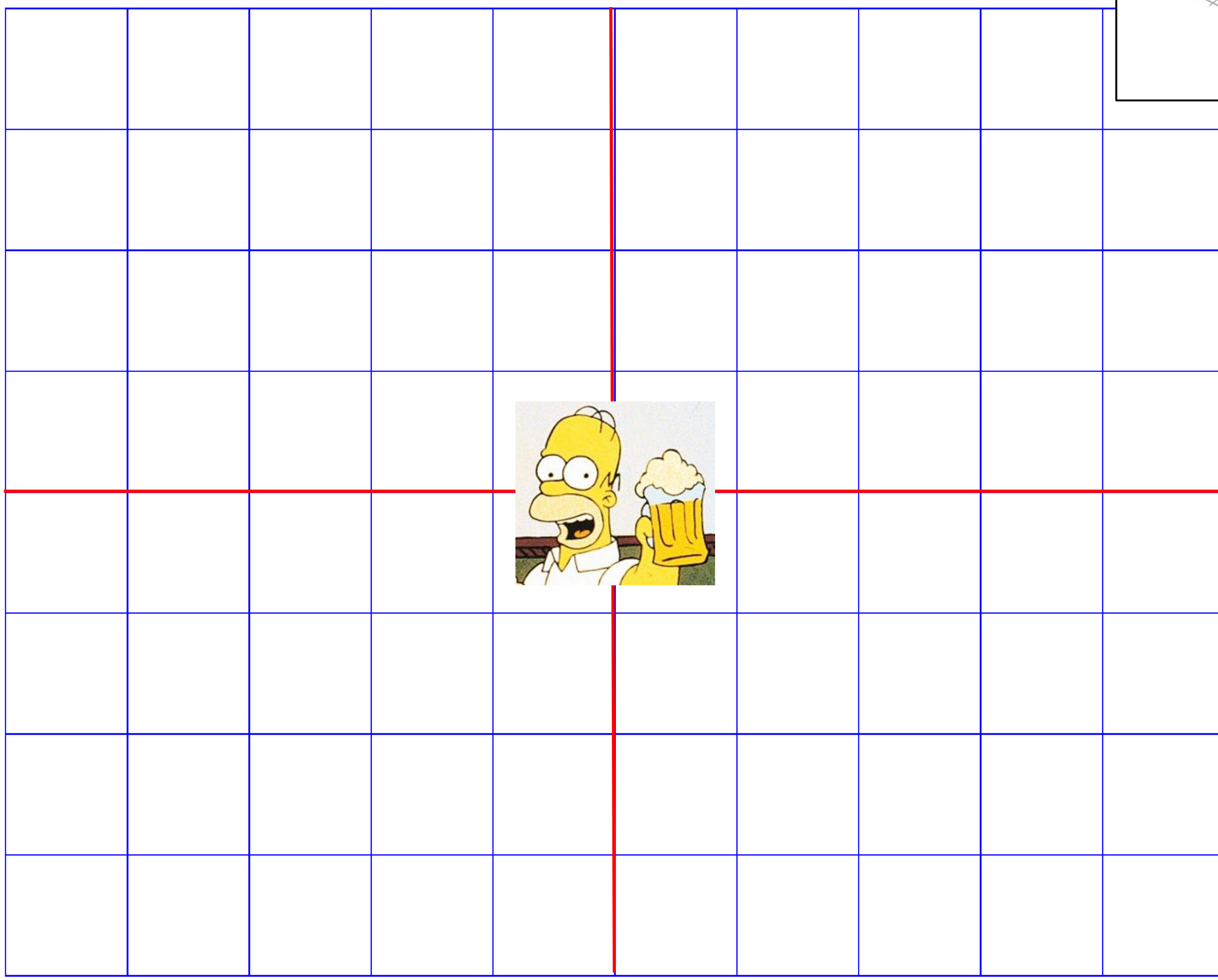
C) Error

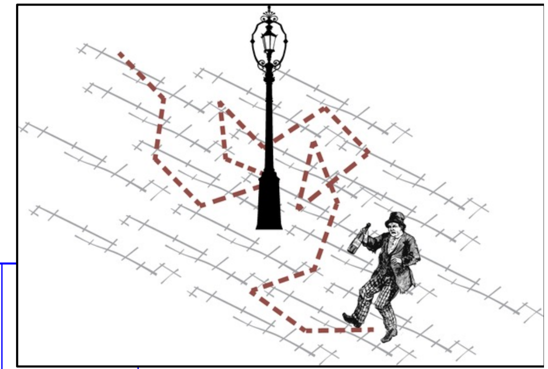
Designing __eq__()

- **`Rabbit`** example
  - each **`Rabbit`** has two parents and a unique ID
  - new **`Rabbit`**s are created by **`+`**
  - want siblings from the same parents to compare **`==`**
- Version 1
  - compare **`==`** on parent tuples
  - triggers **`==`** and hence **`__eq__()`** on elements, recursion!
  - if self is a **`Rabbit`** and other is **`None`**, then invalid to access **`other.parents`**
- Version 2
  - directly compare parent IDs
  - avoids recursion, saves computation
  - runs into same problem retrieving **`parent.id`** if **`parent`** is **`None`**
- Version 3
  - so close! need a valid "ID" for a **`None`** parent
  - wrap that concept in a helper function
  - good opportunity to use **`lambda`**

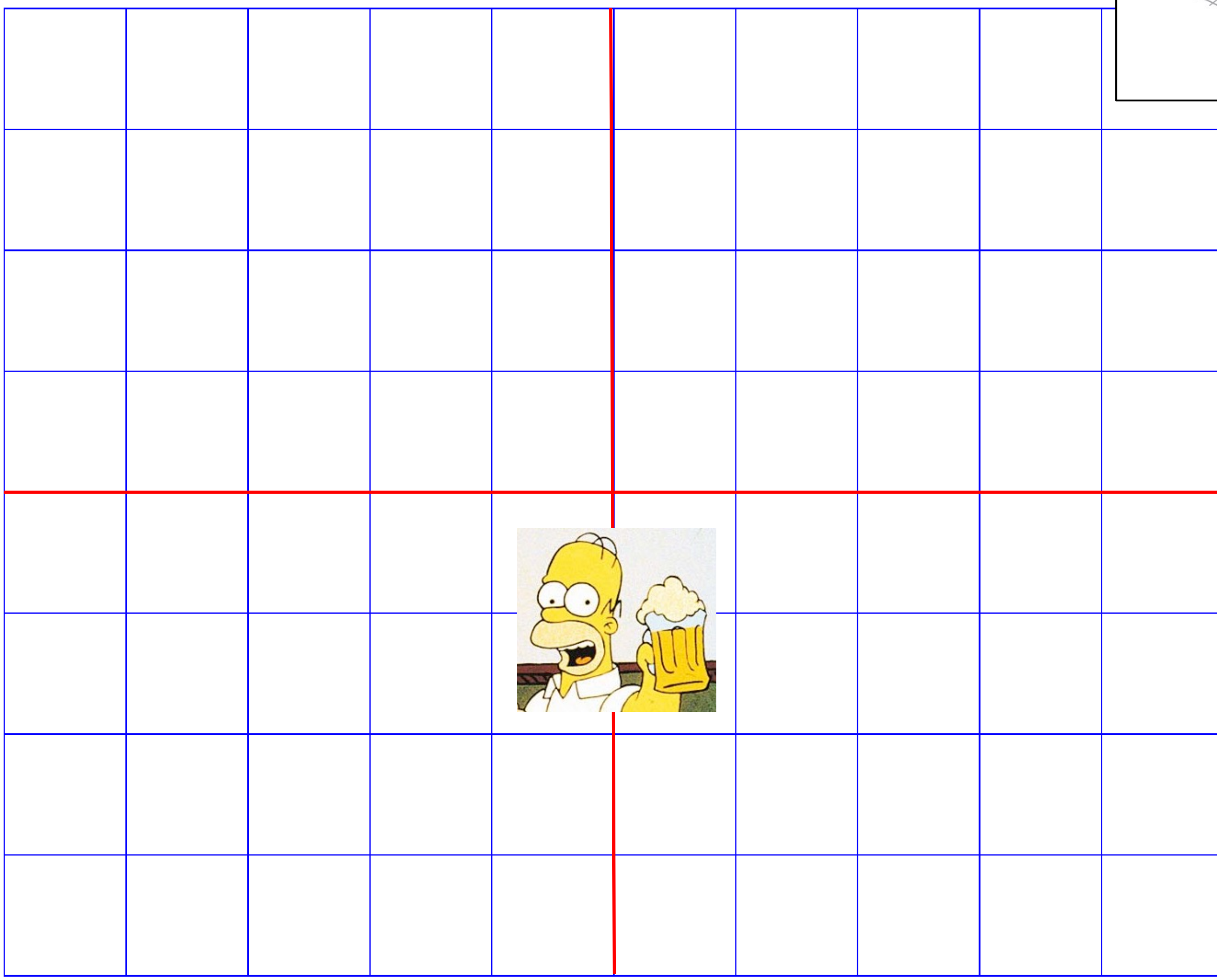
Let’s revist our drunk simulation with classes

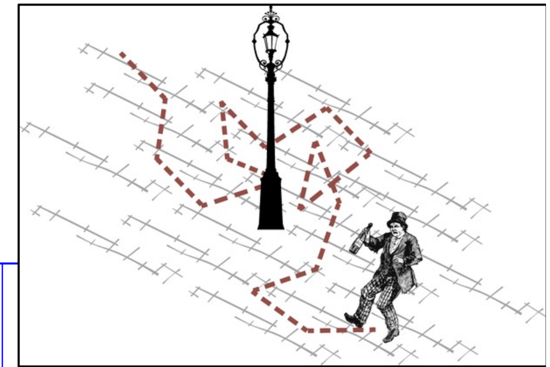
Hand Simulation



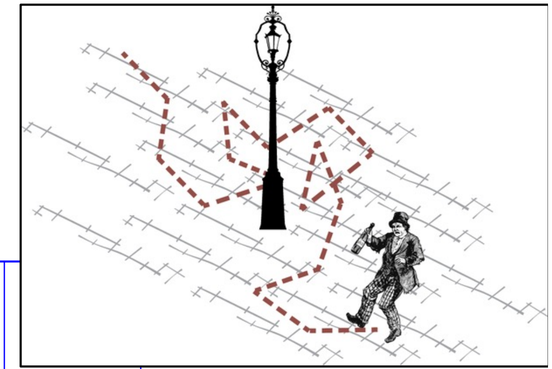


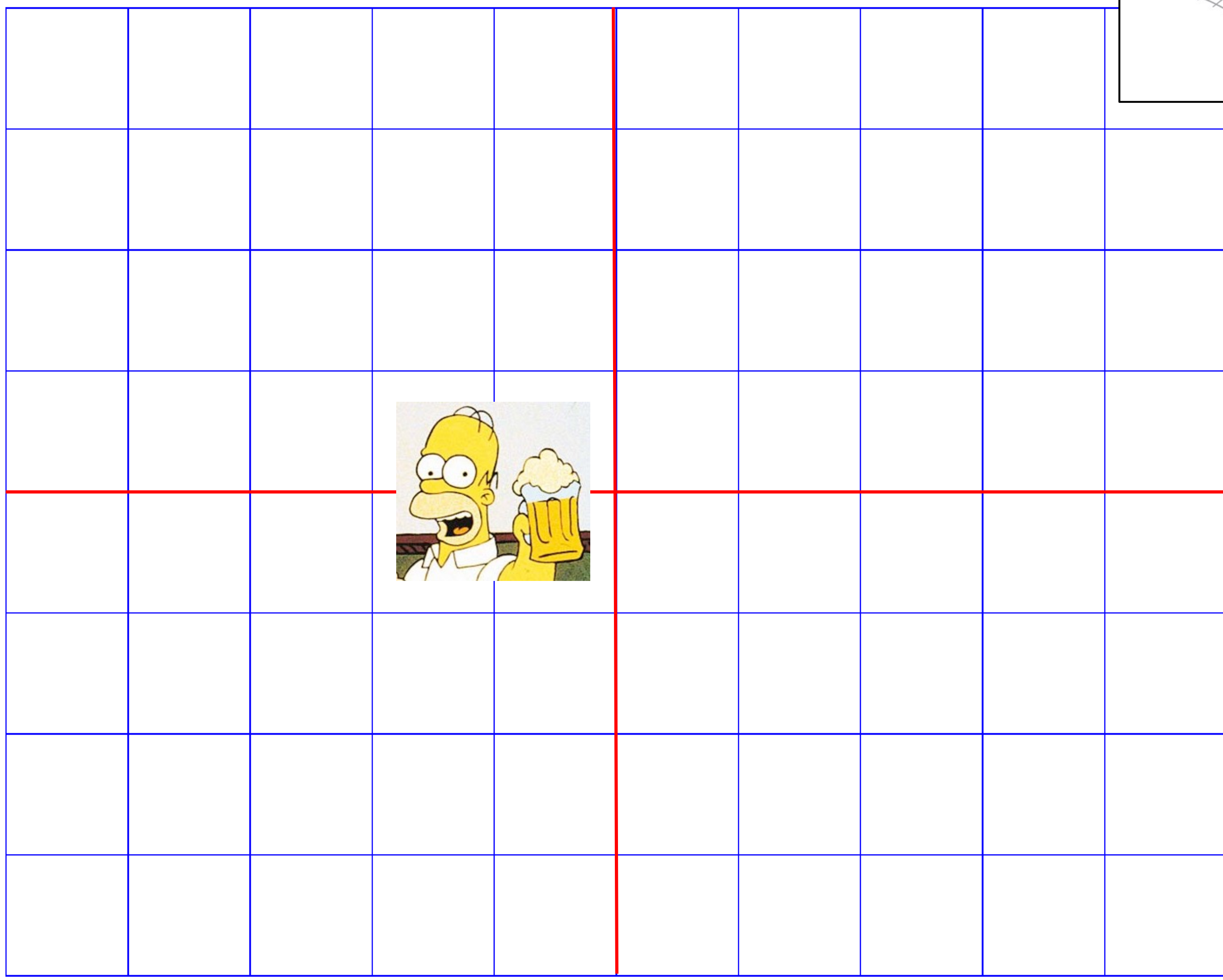
One Possible First Step



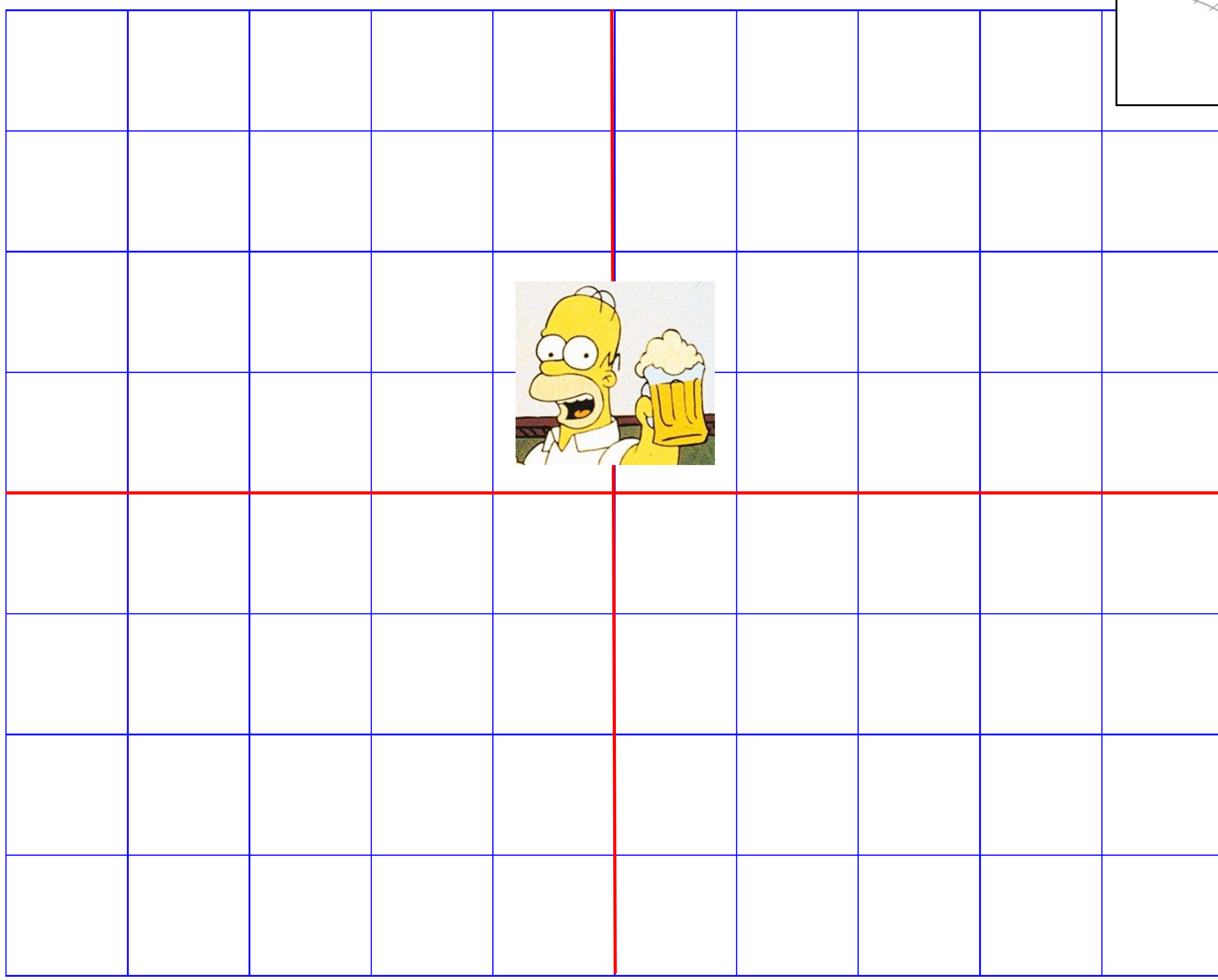


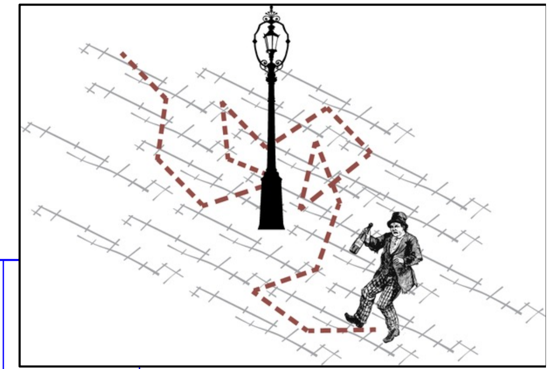
Another Possible First Step



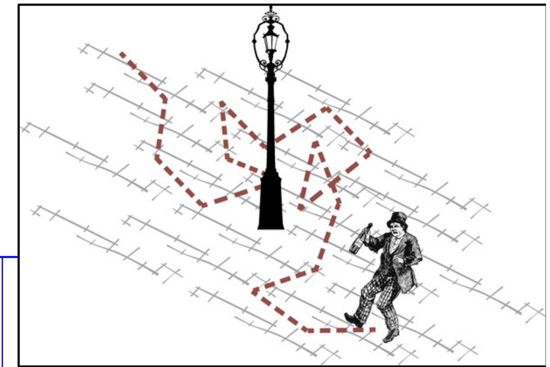


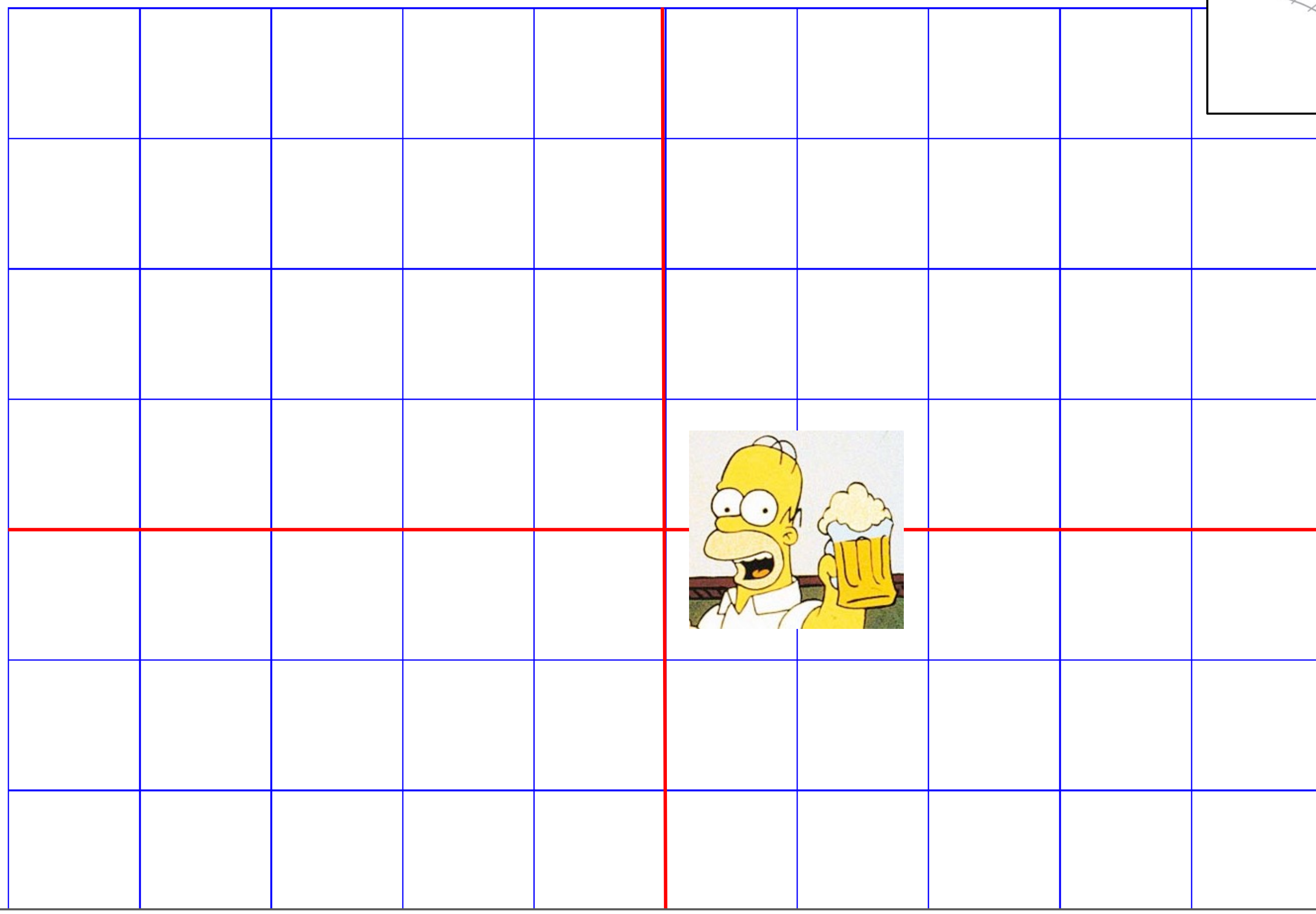
Yet Another Possible First Step





Last Possible First Step





After one step drunk is distance 1.0 away from start

Location

```python
def make_location(x, y):
    return (x, y)

def move(loc, dx, dy):
    x, y = loc
    return (x + dx, y + dy)

def get_x(loc):
    return loc[0]

def get_y(loc):
    return loc[1]

def dist(loc_a, loc_b):
    return hypot(loc_a[0] - loc_b[0], loc_a[1] - loc_b[1])

def loc_str(loc):
    return '<' + str(get_x(loc)) + ', ' + str(get_y(loc)) + '>'
```

```python
class Location:

    def __init__(self, x, y):
        """x and y are numbers"""
        self.x = x
        self.y = y

    def move(self, delta_x, delta_y):
        """deltaX and deltaY are numbers"""
        return Location(self.x + delta_x, self.y + delta_y)

    def get_x(self):
        return self.x

    def get_y(self):
        return self.y

    def dist_from(self, other):
        x_dist = self.x - other.get_x()
        y_dist = self.y - other.get_y()
        return (x_dist**2 + y_dist**2)**0.5

    def __str__(self):
        return '<' + str(self.x) + ', ' + str(self.y) + '>'
```

Field

```python
def make_field():
    return {}

def add_drunk(field, drunk_id, loc):
    if drunk_id in field:
        raise ValueError("Duplicate drunk")
    field[drunk_id] = loc

def get_loc(field, drunk_id):
    if drunk_id not in field:
        raise ValueError("Drunk not in field")
    return field[drunk_id]
```

```python
class Field:

    def __init__(self):
        self.drunk_locs = {}

    def add_drunk(self, drunk, loc):
        if drunk in self.drunk_locs:
            raise ValueError('Duplicate drunk')
        else:
            self.drunk_locs[drunk] = loc

    def get_loc(self, drunk):
        if drunk not in self.drunk_locs:
            raise ValueError('Drunk not in field')
        return self.drunk_locs[drunk]

    def move_drunk(self, drunk):
        if drunk not in self.drunk_locs:
            raise ValueError('Drunk not in field')
        x_dist, y_dist = drunk.take_step()
        # use move() method of Location to set new location
        self.drunk_locs[drunk] = self.drunk_locs[drunk].move(x_dist, y_dist)

    def move_drunk_n_steps(self, drunk, steps):
        if drunk not in self.drunk_locs:
            raise ValueError('Drunk not in field')
        start = self.get_loc(drunk)
        for _ in range(steps):
            self.move_drunk(drunk)
        return start.dist_from(self.get_loc(drunk))
```

Drunks

```python
def usual_step():
    return random.choice([(0, 1), (0, -1), (1, 0), (-1, 0)])

def masochist_step():
    return random.choice([(0.0, 1.1), (0.0, -0.9), (1.0, 0.0), (-1.0, 0.0)])

def liberal_step():
    return random.choice([(0.0, 1.0), (0.0, -1.0), (0.9, 0.0), (-1.1, 0.0)])

def conservative_step():
    return random.choice([(0.0, 1.0), (0.0, -1.0), (1.1, 0.0), (-0.9, 0.0)])

def liberal_masochist_step():
    # Flip between liberal and masochist tendencies
    if random.choice([True, False]):
        return liberal_step()
    else:
        return masochist_step()

def corner_step():
    return random.choice([(0.71, 0.71), (0.71, -0.71), (-0.71, 0.71), (-0.71, -0.71)])

def continuous_step():
    return (random.uniform(-1,1), random.uniform(-1,1))
```

```python
class Drunk:

    def __init__(self, name=None):
        """Assumes name is a str"""
        self.name = name

    def __str__(self):
        if self is not None:
            return self.name
        return 'Anonymous'


class UsualDrunk(Drunk):

    def take_step(self):
        step_choices = [(0, 1), (0, -1),
                        (1, 0), (-1, 0)]
        return random.choice(step_choices)

class MasochistDrunk(Drunk):

    def take_step(self):
        step_choices = [(0.0, 1.1), (0.0, -0.9),
                        (1.0, 0.0), (-1.0, 0.0)]
        return random.choice(step_choices)
```

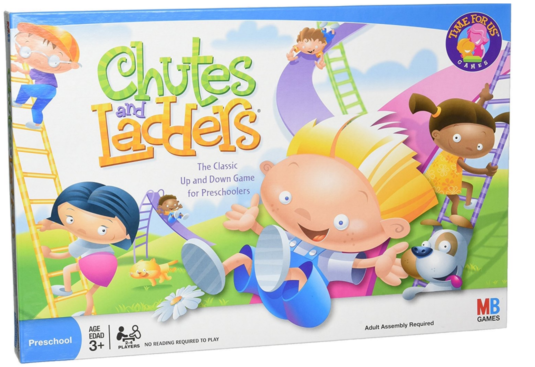
```python
class LiberalDrunk(Drunk):

    def take_step(self):
        step_choices = [(0.0, 1.0), (0.0, -1.0),
                        (0.9, 0.0), (-1.1, 0.0)]
        return random.choice(step_choices)


class ConservativeDrunk(Drunk):

    def take_step(self):
        step_choices = [(0.0, 1.0), (0.0, -1.0),
                        (1.1, 0.0), (-0.9, 0.0)]
        return random.choice(step_choices)
```

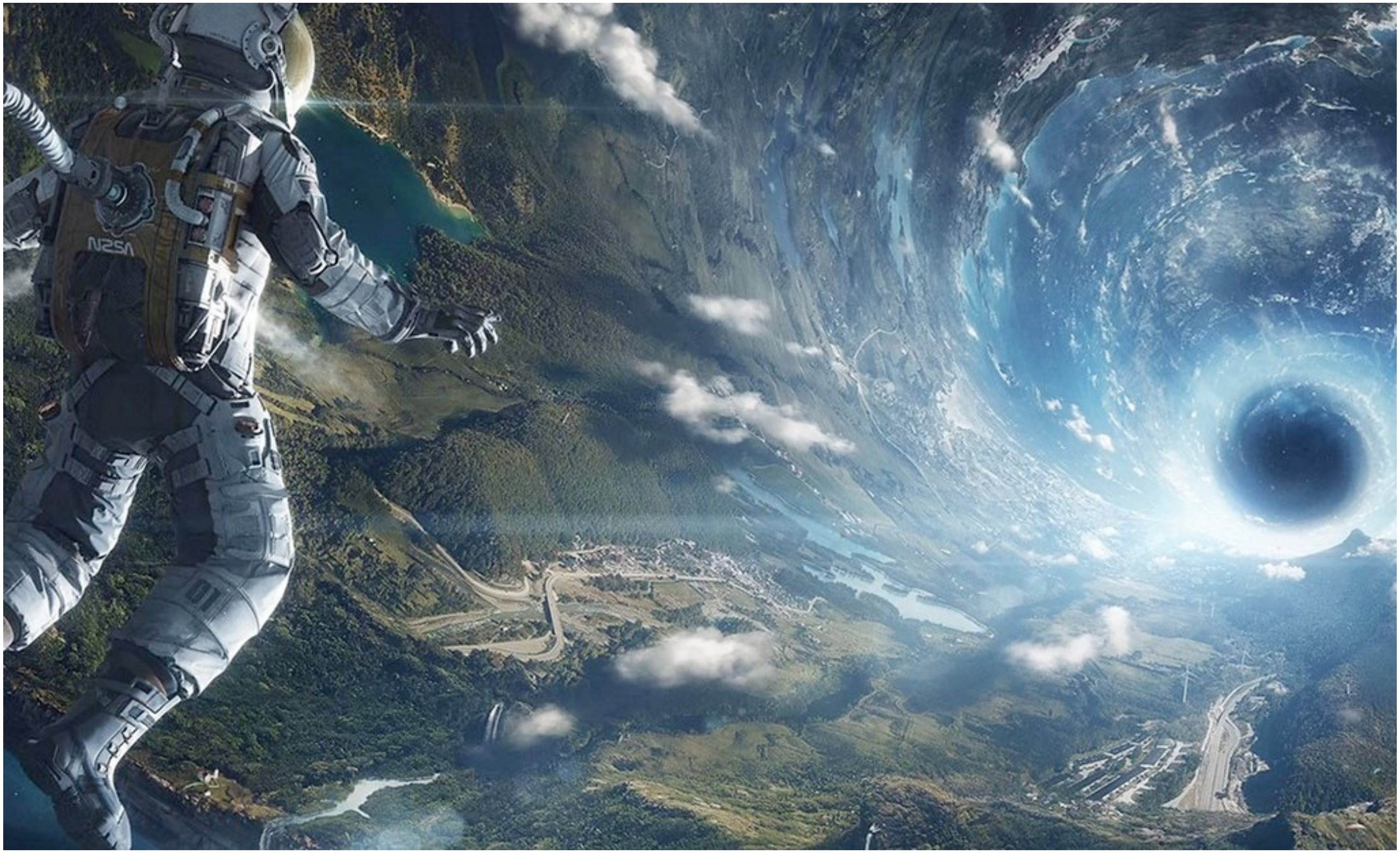
```python
class LiberalMasochistDrunk(MasochistDrunk):

    def take_step(self):
        if random.choice([True, False]):
            step_choices = [(0.0, 1.0), (0.0, -1.0),
                            (0.9, 0.0), (-1.1, 0.0)]
            return random.choice(step_choices)
        else:
            return MasochistDrunk.take_step(self)


class CornerDrunk(Drunk):

    def take_step(self):
        step_choices = [(0.71, 0.71), (0.71, -0.71),
                        (-0.71, 0.71), (-0.71, -0.71)]
        return random.choice(step_choices)
```

The Simulation

```
def sim_walks(num_steps, num_trials, d_class):
    """Assumes num_steps an int >= 0, num_trials an int > 0,
         d_class a subclass of Drunk
       Simulates num_trials walks of num_steps steps each.
       Returns a list of the final distances for each trial"""
    Homer = d_class('Homer')
    origin = Location(0, 0)
    distances = []
    for _ in range(num_trials):
        f = Field()
        f.add_drunk(Homer, origin)
        distances.append(round(f.move_drunk_n_steps(Homer, num_steps), 1))
    return distances


def drunk_test(walk_lengths, num_trials, d_class):
    """Assumes walk_lengths a sequence of ints >= 0
         num_trials an int > 0, d_class a subclass of Drunk
       For each number of steps in walk_lengths, runs
       sim_walks with num_trials walks and prints results"""
    for num_steps in walk_lengths:
        distances = sim_walks(num_steps, num_trials, d_class)
        print(d_class.__name__, 'random walk of', num_steps, 'steps')
        print(' Mean =', round(sum(distances)/len(distances), 4))
        print(' Max =', max(distances), 'Min =', min(distances))
```

Fields with Wormholes





A Subclass of Field, part 1

Inherit attributes and methods of Field class – drunks, move_drunk

Create coordinates of start of a wormhole

Create location of end of a wormhole

Install wormhole in dictionary, keyed by start point

```
class Odd_field(Field):
    def __init__(self, num_holes = 1000,
                 x_range = 100, y_range = 100):
        Field.__init__(self)
        self.wormholes = {}
        for w in range(num_holes):
            x = random.randint(-x_range, x_range)
            y = random.randint(-y_range, y_range)
            new_x = random.randint(-x_range, x_range)
            new_y = random.randint(-y_range, y_range)
            new_loc = Location(new_x, new_y)
            self.wormholes[(x, y)] = new_loc
```

Note: we are assuming steps are of integer length, since using actual x,y values as entry for wormhole

A Subclass of Field, part 2

```
def move_drunk(self, drunk):
    Field.move_drunk(self, drunk)
    x = self.drunks[drunk].get_x()
    y = self.drunks[drunk].get_y()
    if (x, y) in self.wormholes:
        self.drunks[drunk] = self.wormholes[(x, y)]
```

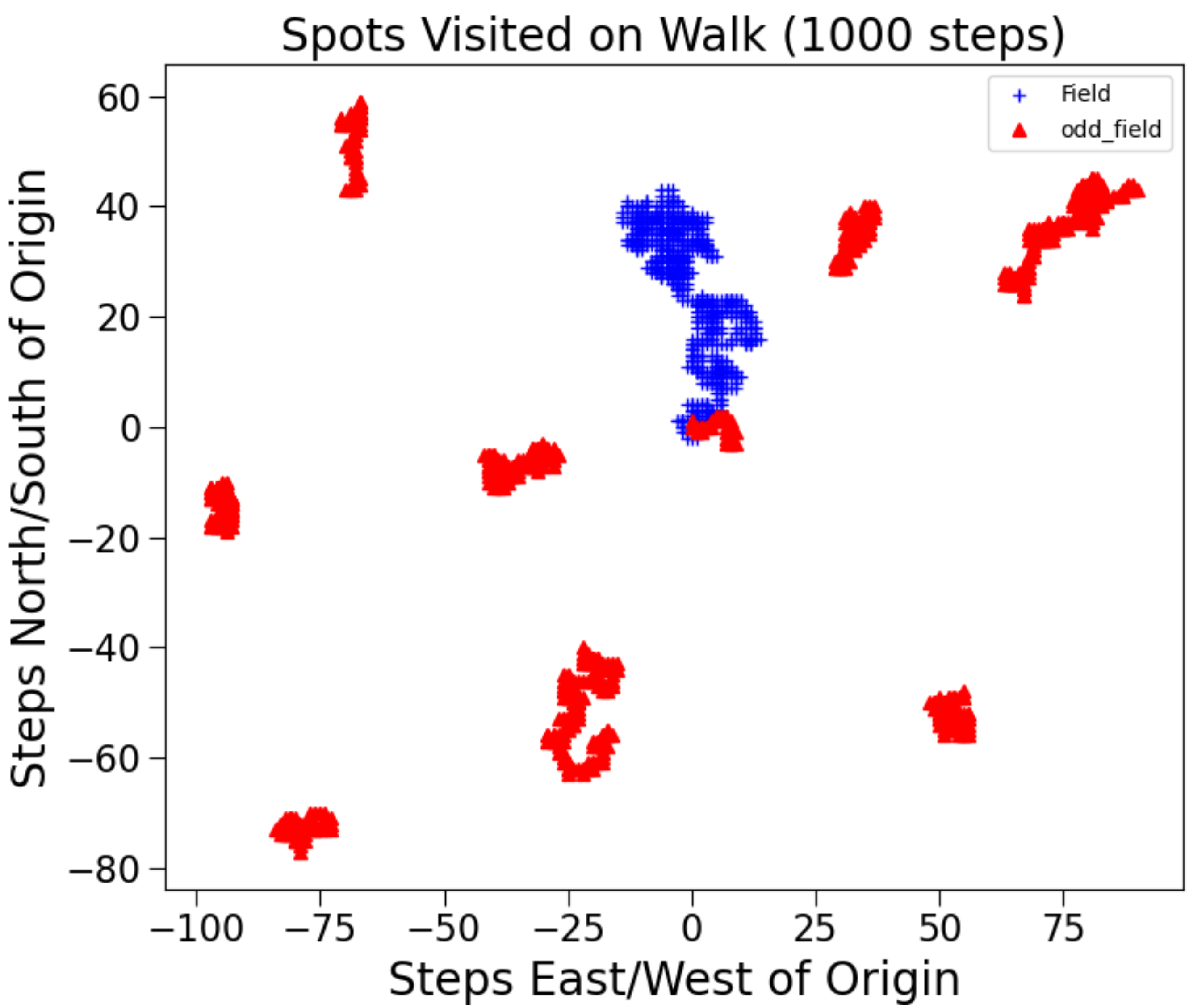
Basic move method;
Drunk now in new location

After move drunk to new location, check to see if have landed on a wormhole

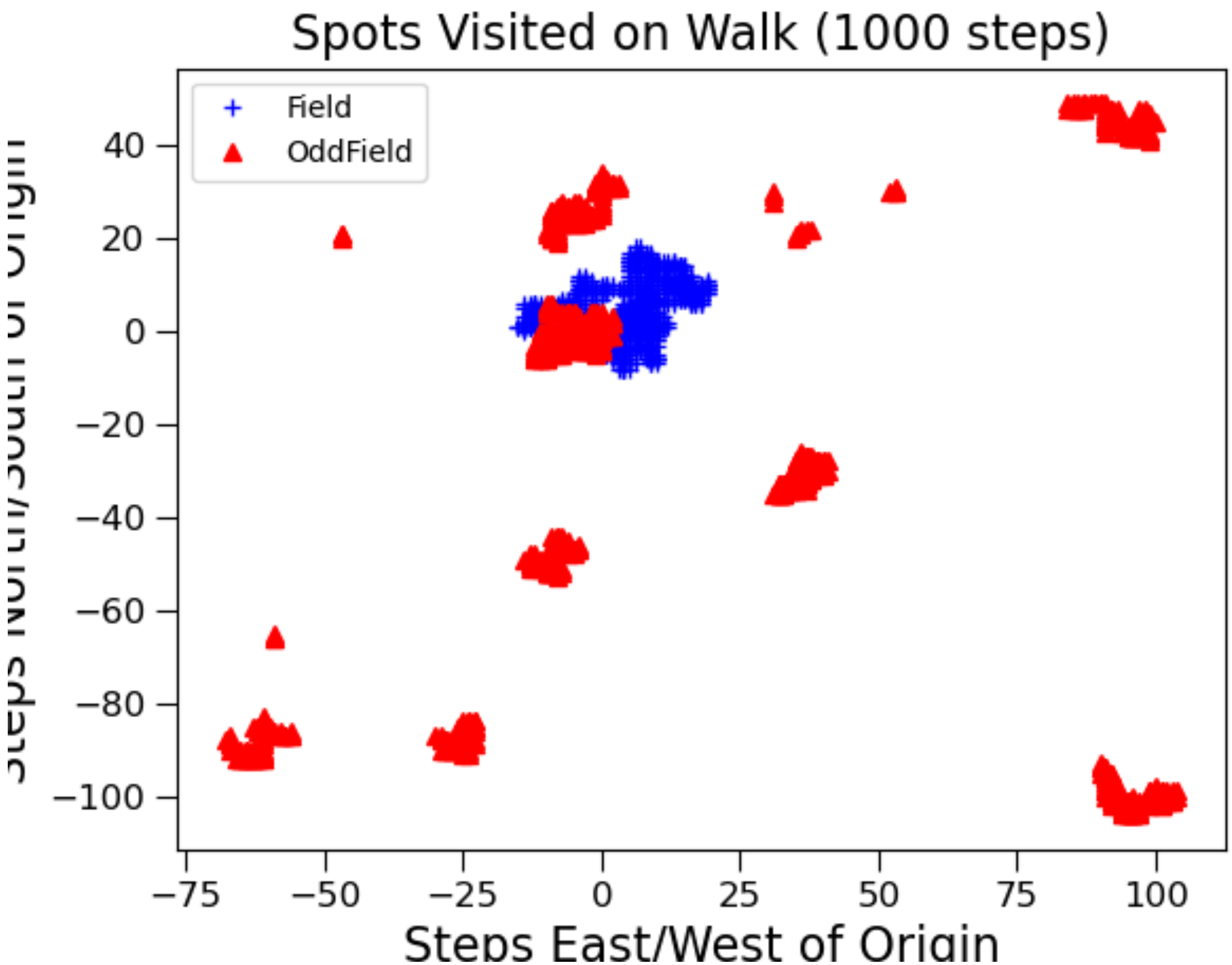
If yes, then get end location of wormhole

Change location of drunk to now be at end of wormhole

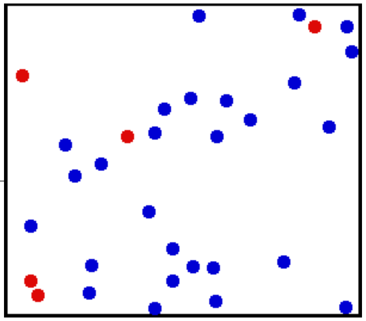
Spots Reached During One Walk



Spots Reached During A Different Walk



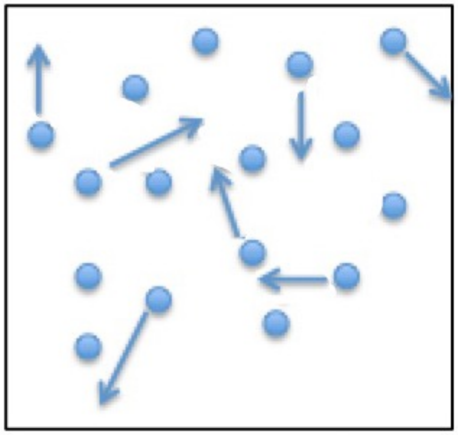
But the World is Not Just a Collection of Drunks



- Have used a simple, two-dimensional world to explore random walk of drunks
- Can imagine how could simulate a field with multiple drunks at the same time
- Can imagine extending simulation so that if two drunks collide trying to move to same location, their movements would change (e.g., rebound backwards); or if a drunk hits a border of the world, it bounces back
- Now imagine replacing drunks with molecules in a gas
- Brownian motion simulation

Modeling a Gas

Hot-air balloon



Molecules inside balloon

Ideal Gas Law

$pV = nRT$

p – pressure
V – volume
n – number of molecules
R – a constant
T - temperature

We will model this in two dimensions

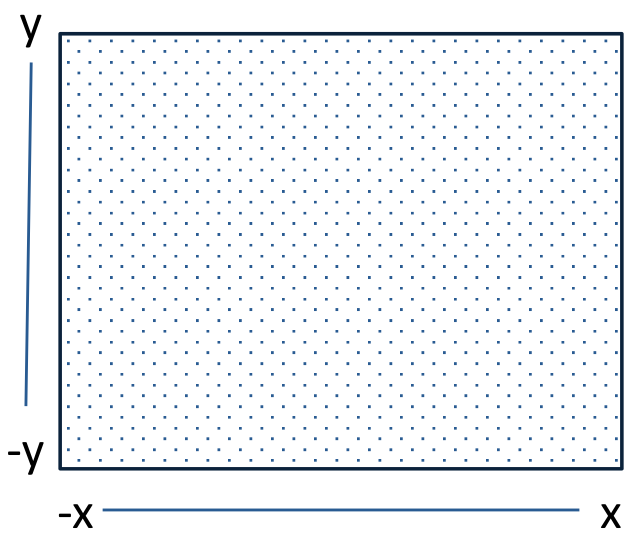
V corresponds to size of field
n corresponds number of drunks (particles)
T corresponds to walk length (velocity of particles)

Simple Trial

- Have included some basic code to explore this in 2D
- Changes
  - Field includes a limit on x and y dimensions (think rigid container)
  - `move` method checks:
    - If would move beyond limit, don't move
    - If move would take particle too close to any other, don't move
    - (simplification, since in reality particles bounce)
  - Start all particles at random locations
    - Measure final distance from each start, rather than origin
  - Record average final distance for all particles in a trial

Complexity

- Two implementations of Field (think of them as 2D containers)
  - One optimized for small number of particles
  - One for larger number of particles



Implementation 1

- Associate location with each particle
  - Space linear in number of particles
- At each step, for each particle, check for collision with each other particle
  - Time of each step quadratic in number of particles

Implementation 2

- For each 1x1 cell of field, keep track of location of any particle in that cell
- Space linear in size of field (x_len*y_len) + number of drunks
  - Initialization of field linear in size of field
- At each step, for each particle, check if any of the cells neighboring the destination contain a particle. If so, see if particle in that cell is too close to the intended destination
  - Time of each step constant in number of particles

Common algorithmic technique: use a fast way to get in neighborhood of a solution, then a slower algorithm to explore neighborhood

Code

```
class Field_multi(object):
    """ Optimized for large fields with small number of particles"""
    def __init__(self, x_lim, y_lim):
        self.drunks = {}
        self.x_lim = x_lim
        self.y_lim = y_lim
        self.wall_hits, self.collisions = 0, 0
```

Multi because it can contain multiple particles

```
class Field_multi_opt_particles(Field_multi):
    """ Optimized for large number of particles"""
    def __init__(self, x_lim, y_lim):
        super().__init__(x_lim, y_lim)
        # Used so that collisions can be detected in constant time
        self.drunks_by_loc = {(x, y): []
                              for x in range(-(x_lim + 1), x_lim + 2)
                              for y in range(-(y_lim + 1), y_lim + 2)}
```

Complexity

Complexity: Implementation 1

number of particles^2 * total number of steps

Independent of size of field

1024 particles, 10 trials, 1,000 steps per trial

$1024^2*10*1000 = 10,485,760,000$

Complexity: Implementation 2

Independent of number of particles

2*x_lim*y_lim*number of trials + total number of steps

Initialize field

Simulate walk

x_lim, y_lim = 1024, 10 trials, 1,000 steps per trial

$2*1024^2*10 = 20,971,520$

Some Simple Experiments

- Questions to explore, e.g.,
  - How does average over a set of trials of average final distance for a set of particles change
    - As the size of the container changes
    - As the number of particles changes
    - As the number of steps each particle takes changes
- Run a set of trials for different choices of container size, number of particles, number of steps
- Are there interesting trends?

Explore Impact of Different Parameter Values

```
def drunk_test_multi(walk_lengths, num_trials, d_class, num_particles,
                     boundaries, opt_space = False, verbose = False):
    """Assumes walk_lengths a sequence of ints >= 0
         num_trials an int > 0, d_class a subclass of Drunk
       For each number of steps in walk_lengths, runs
       sim_walks with num_trials walks and prints results"""
    for l in walk_lengths:
        for num in num_particles:
            for d in boundaries:
                random.seed(1)
                print(f'particles = {num}, size = {d:,}, steps = {l:,}')
                mean_dists, max_dists, min_dists, wall_hits, cols =\
                            sim_walks_multi(l, num_trials, d_class, num, d,
                                            verbose = verbose,
                                            opt_space = opt_space)
                max_d = round(max(max_dists))
                min_d = round(min(min_dists))
                mean_d = sum(mean_dists)/len(mean_dists)
                mean_wh = sum(wall_hits)/len(wall_hits)
                mean_col = sum(cols)/len(cols)
                print(f' Distance: Max = {max_d}, Min = {min_d},',
                      f'Mean = {mean_d:.2f}')
                print(f' Mean wall hits = {round(mean_wh):,}')
                if mean_col != 0:
                    print(f' Mean collisions = {round(mean_col):,}')
```

Vary Size of Field (container)

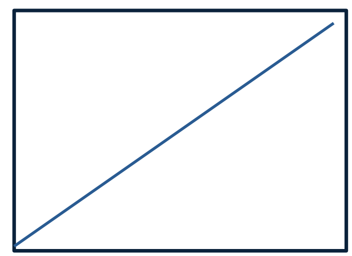
```
random.seed(1)
num_particles = (1,)
sizes = (10, 20, 50, 100, 1000, 10000)
lengths = (500,)
num_trials = 50
drunk_test_multi(lengths, num_trials, Particle, num_particles, sizes, opt_space=True)
```

Particle = Continuous_drunk

particles = 1, size = 10, steps = 500
Distance: Max = 23, Min = 2, Mean = 12
Mean wall hits = 39
particles = 1, size = 20, steps = 500
Distance: Max = 47, Min = 4, Mean = 22
Mean wall hits = 20
particles = 1, size = 50, steps = 500
Distance: Max = 110, Min = 2, Mean = 52
Mean wall hits = 9
particles = 1, size = 100, steps = 500
Distance: Max = 205, Min = 27, Mean = 106
Mean wall hits = 4
particles = 1, size = 1,000, steps = 500
Distance: Max = 598, Min = 65, Mean = 313
Mean wall hits = 1
particles = 1, size = 10,000, steps = 500
Distance: Max = 661, Min = 78, Mean = 354
Mean wall hits = 0

?

Upper bound on distance
$(2*size^2)^{0.5}$



Distances highly variable

As size grows, distance grows, up to a point

Diffusion is slow

Vary Size of Field (container), cont.

```
random.seed(1)
num_particles = (1,)
sizes = (10, 20, 50, 100, 1000, 10000)
lengths = (500,)
num_trials = 50
drunk_test_multi(lengths, num_trials, Particle, num_particles, sizes, opt_space=True)
```

```
particles = 1, size = 10, steps = 500
 Distance: Max = 23, Min = 2, Mean = 12
 Mean wall hits = 39
particles = 1, size = 20, steps = 500
 Distance: Max = 47, Min = 4, Mean = 22
 Mean wall hits = 20
particles = 1, size = 50, steps = 500
 Distance: Max = 110, Min = 2, Mean = 52
 Mean wall hits = 9
particles = 1, size = 100, steps = 500
 Distance: Max = 205, Min = 27, Mean = 106
 Mean wall hits = 4
particles = 1, size = 1,000, steps = 500
 Distance: Max = 598, Min = 65, Mean = 313
 Mean wall hits = 1
particles = 1, size = 10,000, steps = 500
 Distance: Max = 661, Min = 78, Mean = 354
 Mean wall hits = 0
```

<u>Ideal Gas Law</u>

pV = nRT

p = nRT/V

Wall hits corresponds to pressure

As area (V) grows, pressure decreases

Vary Number of Steps

Number of steps corresponds to velocity which increases with temperature

particles = 1, size = 50, steps = 32
Distance: Max = 37, Min = 3, Mean = 20
Mean wall hits = 1
particles = 1, size = 50, steps = 64
Distance: Max = 67, Min = 6, Mean = 35
Mean wall hits = 1
particles = 1, size = 50, steps = 128
Distance: Max = 104, Min = 3, Mean = 49
Mean wall hits = 3
particles = 1, size = 50, steps = 256
Distance: Max = 110, Min = 5, Mean = 51
Mean wall hits = 4
particles = 1, size = 50, steps = 512
Distance: Max = 119, Min = 9, Mean = 52
Mean wall hits = 9
particles = 1, size = 50, steps = 1,024
Distance: Max = 112, Min = 7, Mean = 48
Mean wall hits = 18

<u>Ideal Gas Law</u>

$pV = nRT$

$p = nRT/V$

As temperature rises, distances increase pressures increase

Vary Number of Particles

```
random.seed(1)
sizes = (50,)
lengths = (400,)
num_trials = 50
num_particles = [50, 100, 200, 300, 400]
wall_hits, collisions = [], []
for n in num_particles:
    mean_d, mean_wh, mean_col = drunk_test_multi(
        lengths, num_trials, Particle, (n,),
        sizes, opt_space=False, verbose=False)
    wall_hits.append(mean_wh)
    collisions.append(mean_col)
```

Vary Number of Particles

particles = 50, size = 50, steps = 200
Distance: Max = 136, Min = 2, Mean = 55
Mean wall hits = 204
Mean collisions = 138
particles = 100, size = 50, steps = 200
Distance: Max = 135, Min = 0, Mean = 52
Mean wall hits = 432
Mean collisions = 575
particles = 200, size = 50, steps = 200
Distance: Max = 129, Min = 0, Mean = 46
Mean wall hits = 864
Mean collisions = 2,357
particles = 300, size = 50, steps = 200
Distance: Max = 129, Min = 0, Mean = 41
Mean wall hits = 1,280
Mean collisions = 5,208
particles = 400, size = 50, steps = 200
Distance: Max = 123, Min = 0, Mean = 37
Mean wall hits = 1,671
Mean collisions = 9,209

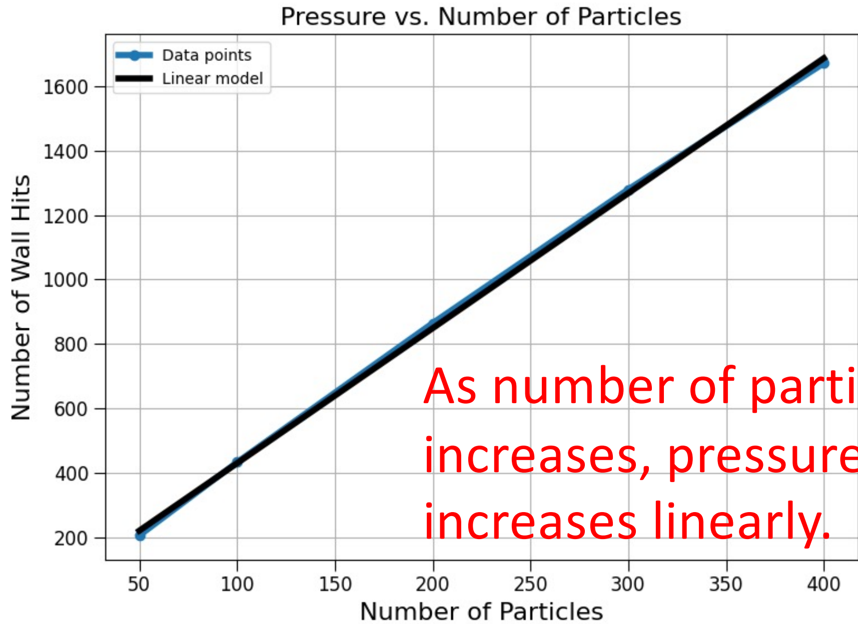
All pretty much the same distances
Differences probably not meaningful

But something regarding distances seems to be happening here.

Density of particles leads to collisions, which reduces distance particles travel

Vary Number of Particles, cont.

particles = 50, size = 50, steps = 200
Distance: Max = 136, Min = 2, Mean = 55
Mean wall hits = 204
Mean collisions = 138
particles = 100, size = 50, steps = 200
Distance: Max = 135, Min = 0, Mean = 52
Mean wall hits = 432
Mean collisions = 575
particles = 200, size = 50, steps = 200
Distance: Max = 129, Min = 0, Mean = 46
Mean wall hits = 864
Mean collisions = 2,357
particles = 300, size = 50, steps = 200
Distance: Max = 129, Min = 0, Mean = 41
Mean wall hits = 1,280
Mean collisions = 5,208
particles = 400, size = 50, steps = 200
Distance: Max = 123, Min = 0, Mean = 37
Mean wall hits = 1,671
Mean collisions = 9,209



As number of particles increases, pressure increases linearly.

As predicted by ideal gas law!

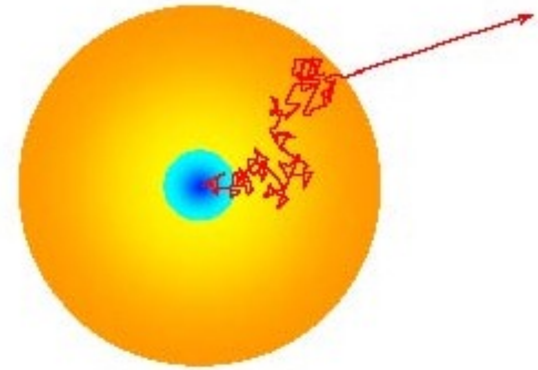
<u>Ideal Gas Law</u>

$$pV = nRT$$

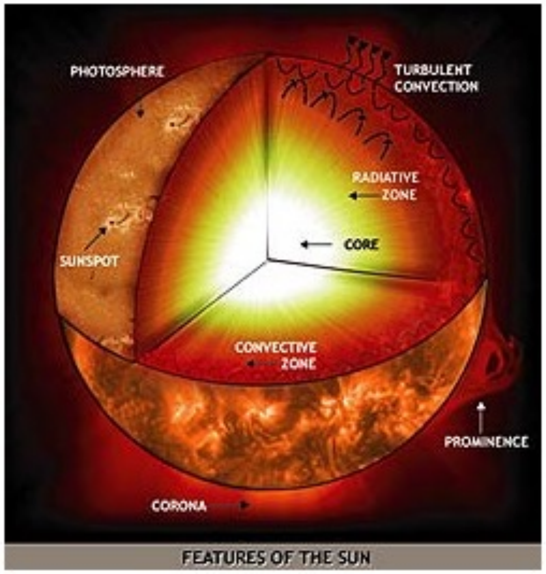
$$p = nRT/V$$

A Factoid

- Light takes ~8.3 minutes to reach the earth from the surface of the sun (93 million miles)
- How long does it take a "photon" to get from center of sun to surface (radius = 432,865 miles)?
  - 8.33/93,000,000 = TS/432,865
  - TS = 8.33*865,370/93,000,000 = ~0.156 minutes?



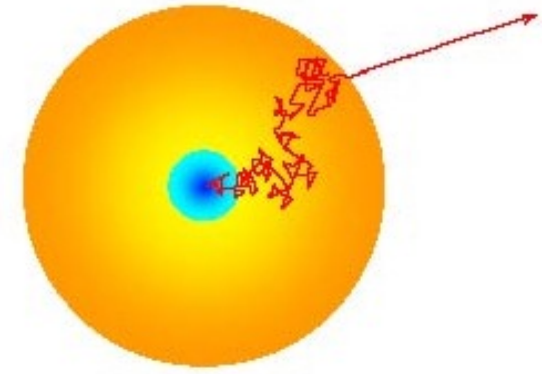
Light escapes the sun's core through a series of random steps as it is absorbed and emitted by atoms along the way (Courtesy - Richard Pogge Ohio State U.)



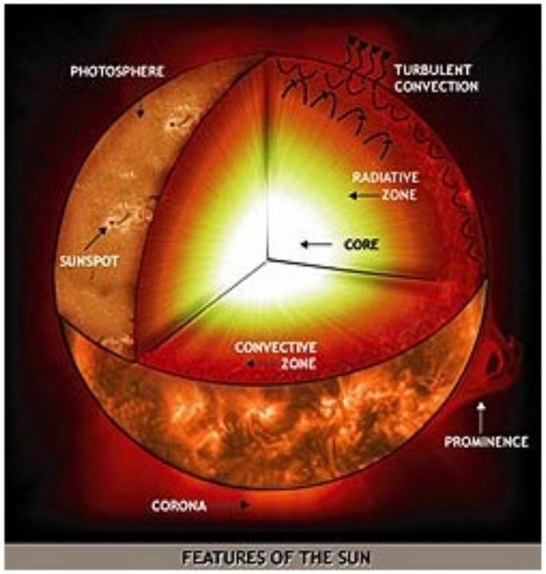
The interior of the sun consists of three major zones, each with its own unique properties. (Courtesy: Berkeley - SSL)

A Factoid

- Light takes ~8.3 minutes to reach the earth from the surface of the sun (93 million miles)
- How long does it take a “photon” to get from center of sun to surface (radius = 432,865 miles)?
    - 8.33/93,000,000 = TS/432,865
    - TS = 8.33*865,370/93,000,000 = ~0.156 minutes?
- **Wrong!**
- Probably between 10,000 and 170,000 YEARS
- Because it’s a **random walk**, and interior of sun is very dense

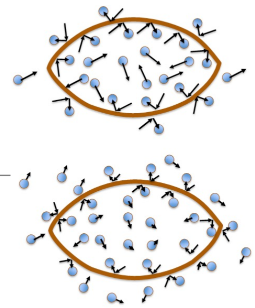


Light escapes the sun's core through a series of random steps as it is absorbed and emitted by atoms along the way (Courtesy - Richard Pogge Ohio State U.)



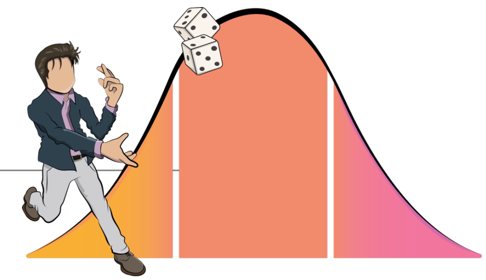
The interior of the sun consists of three major zones, each with its own unique properties. (Courtesy: Berkeley - SSL)

Summary



- Random walks allow us to model many physical and social phenomena
- We can build simulations that help us understand the behavior of various kinds of random walks that would be hard to analyze directly
- Looked at classic drunkards walk
- Looked at a simple particle simulation
  - Showed effect of varying various paramters.
- Point is not the simulations themselves, but how we built, evaluated, and used them

Summary, cont.



- Started by defining classes
  - Good use of sub-classing
- Built functions corresponding to:
  - One trial, multiple trials, result reporting
- Got simple version working first
  - Did a sanity check!
- Made series of incremental changes to simulation so that we could investigate different questions
  - Enhanced simulation a step at a time
- By changing properties of objects, can explore range of behaviors
- Much more on simulation coming up