Nuclear Fallout Simulation
By Young David Choe

Abstract:

Gathering 14 meteorological stations from the Bay Area, I interpolated wind data, such as speed and direction, for a 100 by 100 grid overlaid on a Google map of 700 by 700 pixels. The grid incorporates a heatmap for wind speed, arrow glyphs for each weather station, and an arrow plot. The program described up to this point was the basis for Programming Assignment 1 for CMPS 161, Winter 2019. In order to extend this program, I added the feature of simulating the trajectory of nuclear fallout given the location of the detonation, expected height of the mushroom cloud, radius and density of the cloud plume, and average descent rate of each particle. In order to show the trajectory of each particle over a certain period of time, I upgraded my station data from a single snapshot of one hour of wind data to 24 hours. Using the interpolated wind data, I can interpolate for particle trajectory.

Introduction:

The Cuban Missile Crisis was considered to be the hottest part of the Cold War (insert source). 1962 was the year when two global superpowers almost initiated a full-scale nuclear war, the closest humans ever came to World War III. Although the Cold War ended 27 years ago, the threat of nuclear war is still very real today. People go on with their lives without realizing how fast the world could come to an end.

Although the blast and fire from a nuclear explosion is considered to be deadly in its own right, radiation poisoning from nuclear fallout is potentially a larger issue. If a bomb were to detonate close to the earth’s surface, rather than in air, it would upheave massive amounts of debris and produce significant amounts of radioactive material. Wind could carry this radioactive fallout miles away from ground zero, poisoning people in its path.

My goal is to simulate the trajectory of nuclear fallout using real wind data from the Bay Area. I hope to visualize the dangers of nuclear detonation and the overall scope of the issue.

Related Works:

My program was heavily inspired by a site called NUKEMAP. It allows a user to select a location and detonate a bomb with a specified yield (in kilotons). Having the bomb detonate as a surface burst will give contour lines of the radioactive fallout. The user can also pick a general wind direction.
Problem:

There already exists programs that simulate the trajectory of nuclear fallout, such as NUKEMAP (https://nuclearsecrecy.com/nukemap/). However, programs such as these are based on scaling models instead of actual models based on weather conditions. Scaling models are computationally easy but not necessarily accurate. The scaling model that NUKEMAP uses is based on declassified documents written by Carl F. Miller and assumes a constant wind condition of 15 mph with no variations in wind direction.

My solution is to visualize a fallout model based on real weather condition and particle simulation.

User Documentation:

The user will receive a directory containing an HTML file called “driver.html” and a subdirectory called “/js” which the necessary JavaScript files needed to run simulation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>js</td>
<td>3/21/2019 3:28 AM</td>
<td>File folder</td>
<td></td>
</tr>
<tr>
<td>driver.html</td>
<td>3/21/2019 3:02 AM</td>
<td>Chrome HTML Do...</td>
<td>3 KB</td>
</tr>
</tbody>
</table>

(Image Shows what the directory should look like)

If the user does not already have this directory, he or she can retrieve it at:

https://people.ucsc.edu/~ydchoe/CMPS%20161/proj/FalloutSimulation/

To start the program, the user must open the HTML file on a web browser of his or her choice (i.e. Chrome, Firefox, Safari, Edge, etc). Please note that internet connection is required in order
to use the D3.js library and Google Map API. Once opened, a static Google Map will be rendered on screen along with a heat and the initial starting position of the nuclear fallout. Buttons below the map can be used to toggle different features (heat map, station arrows, arrow plot, and streamlines) on and off.

The user can change the hour of the simulation by using the slider underneath the map. Fallout particles are colored magenta and get darker as it gets closer to the ground. The red number on the top right corner specifies what hour the simulation is currently in.

**Features:**

The user is given a 5 color heatmap that represents wind speed (bluish means low, reddish means high). Along with the heatmap, the user is given station arrows, an arrow plot, streamlines, and fallout particle position. Size and color of the arrows correlate to its wind speed, while the direction of the arrowheads point to the direction of the wind. The streamlines help show the general path of the wind.

**What Was Used to Represent the Data:**

My data is made up of 14 weather stations that were listed on the Bay Area Air Quality Management District (BAAQMD) website. The time frame I had chosen was February 02, 2019. I picked this particular time frame because I was emailed a high wind advisory by Public Affairs at UCSC. The BAAQMD provided station names, station geo locations, winds speeds, and wind directions.

I decided to use HTML, JavaScript, D3.js (Data Driven Documents), and a Google Maps API to represent my data. JavaScript provided mve of the calculations and data structure needed for the program, while D3.js provided the user with a visual representation of the data (i.e. Heat Map, Station Arrows, Arrow Plot, Streamlines, and Fallout Particles). The Google Maps API also provides the user with visual representation of the geographical region of interest, but its main function is to help convert between HTML pixel coordinates and GPS coordinates in order accurately interpolate wind data.
Spatial Extent and Resolution:

The grid data is 100 x 100 in resolution and it is visually represented on a SVG element of 700 x 700 pixels. Underneath the SVG element lies a static Google Map that has a resolution of 700 x 700 pixels. The map is centered at latitude 37.93997352043745 and longitude -122.2193390178572 with a map zoom factor of nine. This gives a map of the Bay Area. For reference, the boundaries for the map are North-East Corner (38.6941920402164, -121.25803019084822), South-West Corner (37.17793481617096, -121.25803019084822).

Generating the Grid Data:

```javascript
class StationData {
  constructor(name, latitude, longitude) {
    this.stationName = name; // Name of the wind station
    this.coordinates = [parseFloat(latitude), parseFloat(longitude)]; // Coordinates of the wind station (in Decimal)
    this.windSpeed = []; // Wind speed (mph) for every 5 hour
    this.windDirection = []; // Wind direction for every 5 hour (compass bearing)
    this.xPos = null;
    this.yPos = null;
  }
}
var stationArray = [];

const stationData = new StationData("Sausalito Island", 38.009631, -121.641918);
stationData.windSpeed = [23, 21, 23, 24, 22, 7, 20, 6, 8, 9, 13, 12, 11, 17, 10, 14, 17, 20, 20];
stationData.windDirection = [131, 126, 129, 127, 150, 143, 142, 146, 144, 166, 153, 169, 176, 173, 192, 197, 166, 137, 126, 133, 132, 132];
stationArray.push(stationData);
```

*(Image above shows what initializing a StationData Object looks like)*

I manually take each station’s data from BAAQMD and assign them to a JS class object called `StationData`. Each class contains the station’s name, geo location (in decimal degrees), 24 hours of wind speed and wind direction, and pixel coordinates relative to the SVG element. Then each `StationData` object is added to an array called `stationArray`. `stationArray` is a global variable that be accessed anywhere in the program. If the user needs to add more stations, this is the place to do it.

```javascript
var map = new google.maps.Map(d3.select("#map").node(), {
  zoom: 9.0,
  center: new google.maps.LatLng(37.93997352043745, -122.2193390178572),
  mapTypeId: google.maps.MapTypeId.ROADMAP,
  disableDefaultUI: true,
  gestureHandling: "none",
  clickableIcons: false,
  draggableCursor: "arrow"
});
```

*(Image above shows what initializing a Google Maps API looks like)*

In `main.js` I initialize a Google Map element and assign it to a variable called `map`. Here the user can pick a center location and a zoom factor. This would be the place to change the geographical
location of the map. The Google Maps API allows users to create an overlay to add custom graphic elements. The overlay also allows the user to convert between pixel coordinates and GPS coordinates. Using GPS coordinates instead of pixel coordinates allows for a more accurate interpolation of data.

```javascript
var gridData = new GridData(100, 700);
var projection = overlay.getProjection();
var translatedRes = (gridData.resolution)/2;

for(let i = 0; i < gridData.gridPoints.length; i++)
{
    for(let j = 0; j < gridData.gridPoints[i].length; j++)
    {
        /*
         * I am translating pixel coordinates to GPS coordinates and storing them into each GridPoint.
         * I then use Shepard's algorithm to interpolate each grid point's lat and lng using the
         * 14 weather stations we have gathered
         * */
        var mapCoord = projection.fromDivPixelToLat_lng(new google.maps.Point{
            gridData.gridPoints[i][j].xPos - translatedRes, gridData.gridPoints[i][j].yPos - translatedRes});
        gridData.gridPoints[i][j].mapCoord[0] = mapCoord.lng;
        gridData.gridPoints[i][j].mapCoord[1] = mapCoord.lng;
        shearpAlgorithm(stationArray, gridData.gridPoints[i][j]);
    }
}
```

*(Image above shows code initializing the grid map with a width and height of 100)*

Next I initialize the data grid and convert each pixel coordinate to its relative GPS coordinates on the Google Map. Using Shepard’s Interpolation and the data from the 14 weather stations, I interpolate the wind speed and direction for each grid point. The formula is as follows:

\[
u(x) = \begin{cases} 
\sum_{i=1}^{N} w_i(x) u_i, & \text{if } d(x, x_i) \neq 0 \text{ for all } i, \\
\sum_{i=1}^{N} w_i(x), & \text{if } d(x, x_i) = 0 \text{ for some } i,
\end{cases}
\]

where

\[
w_i(x) = \frac{1}{d(x, x_i)^p}
\]

*(Image above is from Wikipedia - “Inverse Distance Weighting”)*

At this point, I have all the information I need in order to generate a heat map, station arrows, and a arrow plot.

**Generating the Streamlines:**

```javascript
class Streamline {
  constructor()
  {
    this.dataRight = []; // moving in positive direction
    this.startX = null;
    this.startY = null;
    this.dataLeft = []; // moving in negative direction
  }
}
```

*(Image shows what a Streamline Object looks like)*
In order to generate streamlines I need the values for every 10th grid point. Using Euler’s integration, we can find the next and previous streamline point for a nth grid point. The formula is as follows:

\[ P_{k+1} = P_k \pm hV \]

To keep the streamlines smooth, the \( h \) constant must be kept small. For my program I set the \( h \) value to 0.001. In order to find the \( V \) (vector) for \( P_{k+1} \) I simply use Shepard’s Interpolation for that point. The values are stored into an array that is part of the Streamline object.

**Generating the Fallout Particles:**

```javascript
class RadioactiveFallout {
  constructor(cloudHeight, cloudRadius, cloudDensity, descentRate, groundZeroXPos, groundZeroYPos) {
    this.cloudHeight = cloudHeight; // height of mushroom cloud in miles
    this.cloudRadius = cloudRadius; // in pixels value
    this.cloudDensity = cloudDensity; // number of particles
    this.descentRate = descentRate; // descent rate in mph
    this.groundZeroXPos = groundZeroXPos; // pixel coordinate
    this.groundZeroYPos = groundZeroYPos; // pixel coordinate
    this.numberOfParticles = 1;
    this.particleData = []; // every index i represents an hour snapshot of the particles
    var tempArray = [];
    for (let i = 0; i < cloudDensity; i++) {
      tempArray.push(generateParticles(cloudRadius, groundZeroXPos, groundZeroYPos, cloudHeight));
    }
    this.particleData.push(tempArray);
  }

  function generateParticles(radius, xCenter, yCenter, elevation) { // Use SVG pixel coordinates
    let a = Math.random() * 2 * Math.PI;
    let r = radius * Math.sqrt(Math.random());
    let x = r * Math.cos(a) + xCenter;
    let y = r * Math.sin(a) + yCenter;
    var particle = new Particle();
    particle.xPos = x;
    particle.yPos = y;
    particle.elevation = elevation;
    return particle;
  }
}
```

*Image shows how fallout particles are generated *

In order to generate the fallout particles, the user must provide cloud height, cloud radius, cloud density, particle descent rate, and location of the bomb detonation. I generate points randomly but uniformly in a circle described by the parameters above. Using a similar method I used to generate streamlines, I use Shepard’s Interpolation to find the next position of each particle for
every hour. However, instead of using Euler’s integration, I get the new position by using the Haversine Formula.

```
function findNewPoint(lat1, lng1, bearing, distance)
{
    // Need to convert lat, lng, and bearing to radians
    let r = 3958.756; // Earth's Average Radius in miles
    let1 = lat1 * (Math.PI / 180);
    lng1 = lng1 * (Math.PI / 180);
    bearing = bearing * (Math.PI / 180);

    var lat2 = Math.sin(Math.sin(lat1) * Math.cos(distance / r) +
        Math.cos(lat1) * Math.sin(distance / r) * Math.cos(bearing));
    var lng2 = lng1 + Math.atan2(Math.sin(bearing) * Math.sin(distance / r) * Math.cos(lat1),
        Math.cos(distance / r) - Math.sin(lat1) * Math.sin(lat2));

    return [lat2 * (180 / Math.PI), lng2 * (180 / Math.PI)];
}
```

(Image above shows the Haversine Formula)

Given a starting latitude and longitude, bearing, and distance, the Haversine Formula returns coordinates to a new location some X distance (in miles) away at some Θ angle. For every hour, I update the headings and position of a particle 10 times (every 6 minutes), but only display a particle’s final location every hour while it’s in the air. Please note that if the user were to connect each particle to its next corresponding position using a line, he or she would end up with a pathline, not a streamline.

**Displaying Data Using D3:**

D3 allows a user to easily manipulatehtml elements, such as an SVG, based on data. For the heatmap, I drew a grid of squares and filled in the appropriate color using wind speed data I interpolated earlier. Areas that are more blue indicate low wind speeds while areas that are more red indicate high wind speeds. Each station arrow is a line extending from its origin with its angle corresponding to a given wind direction. Each line is terminated with a triangle shape that serves as an arrow head. The length and color of each arrow corresponds with the wind speed at its origin. The streamline is just an array of points connected by a line. The fallout particles were done in a similar fashion without connecting each point (if we did connect them we would have a pathline).
Observing the Data:

Because I am only using 14 weather stations, it is important to note that the wind map is probably only accurate where the stations are bunched up together, mostly around the center. This is apparent when changing the map from a road map to a terrain map. With the streamlines on, we can see that wind is trying to avoid mountainous areas near Concord and Walnut Creek. However, the streamlines make no attempt to avoid mountainous areas around Santa Rosa. As there aren’t any wind stations near the area being used in my program, the wind map has no way of knowing that a mountain is blocking the wind that particular area.

If we pretend that our wind map is completely accurate despite only using 14 weather stations, our fallout particles seem to follow the correct trajectory based on wind. For this particular example, I had the bomb detonate in Santa Rosa, set the height of the mushroom cloud to about 20 miles, set the blast radius to 10 miles, and set cloud density to 2000 particles with a descent rate of 2.1 mph. It is interesting to see that the particles break up into two distinct clumps as it passes through areas with drastic wind changes near Rio Vista. I believe it is safe to say that the trajectory of the particles look reasonable.
**Limitations and Shortcomings:**

I did not generate the streamlines nor the fallout particles in the most efficient manner. The more weather stations the user adds the longer it will take to generate the data. This is because the program is re-interpolating with Shepard’s algorithm for every point of interest. Using bilinear interpolation with the existing wind data that the program rendered earlier would have kept the streamline and fallout particle trajectory runtime to $O(N)$ instead of $O(N^N)$ with the input being the number of weather stations. It also doesn’t help that the program now has to interpolate for wind data that spans over 24 hours.

I do not have easy access to information such fallout particle density or radiation levels of an actual nuclear weapon. As a result, it’s hard to specify and determine a bomb yield in my program. Using an average descent rate is also not ideal, but I don’t know where I could get access to wind data for the 3rd dimension, such as vertical wind shears.