Jeffrey Ding
UID: 104928991
CEE M20
May 26th, 2018

Homework 8

1. Spatial Hashing

1.1 Introduction

The goal of this problem is to map particle data onto a 2D grid by discretizing entire domain into a series of rectangular bins.

1.2 Model and Methods

The script begins by setting the constants of the problem, the number of particles, the minimum bin dimension, setting the max grid values, calculating the number of bins, and the bin dimensions:

```
N = 20;
h = 0.25;
xMax = 1;
yMax = 1;
Nx = floor(xMax/h);
Ny = floor(yMax/h);
dx = xMax/Nx;
dy = yMax/Ny;
```

Then, the script creates two N sized arrays, one of random x values and one of random y values:

```
x = rand(1, N);
y = rand(1, N);
```

Then, the script creates another array to track the bin number of each coordinate. It uses a for loop to determine which bin the particle is in:

```
binNum = zeros(1, N);
for index = 1:N
   binNum(index) = (ceil(x(index)/dx)-1)*Ny + ceil((yMax-y(index))/dy);
end
```

Afterwards, the script creates two more arrays, an array that holds the bin average of X values and the bin average of Y values. Then, the script runs nested for loops for each bin number and each particle and calculates the average value of each bin number. If a bin has no particles, the bin will just hold no value:

```
binAvgX = zeros(1, Nx*Ny);
binAvgY = zeros(1, Nx*Ny);
for binIndex = 1:Nx*Ny
    averageX = 0;   % average x location
    averageY = 0;   % average y location
    count = 0;
    for index = 1:N
        if binNum(index) == binIndex
            averageX = averageX + x(index);
            averageY = averageY + y(index);
            count = count + 1;
        end
    end
    binAvgX(binIndex) = averageX/count;
    binAvgY(binIndex) = averageY/count;
end
```

Then, the script plots the data. It plots the x values against the y values, then plots the bin average x values against the bin average y values on the same plot. The script also plots these values against x and y axes that are divided by dx and dy:

```
scatter(x, y, 50, 'b', 'filled');
hold on;
scatter(binAvgX, binAvgY, 25, 'r', 'x');
grid on;
xticks(0:dx:xMax);
yticks(0:dy:yMax);
```

Then, the script runs nested for loops to print out the each bin and the particles that are in the bin. If the bin is empty, the script prints “[ ]”:

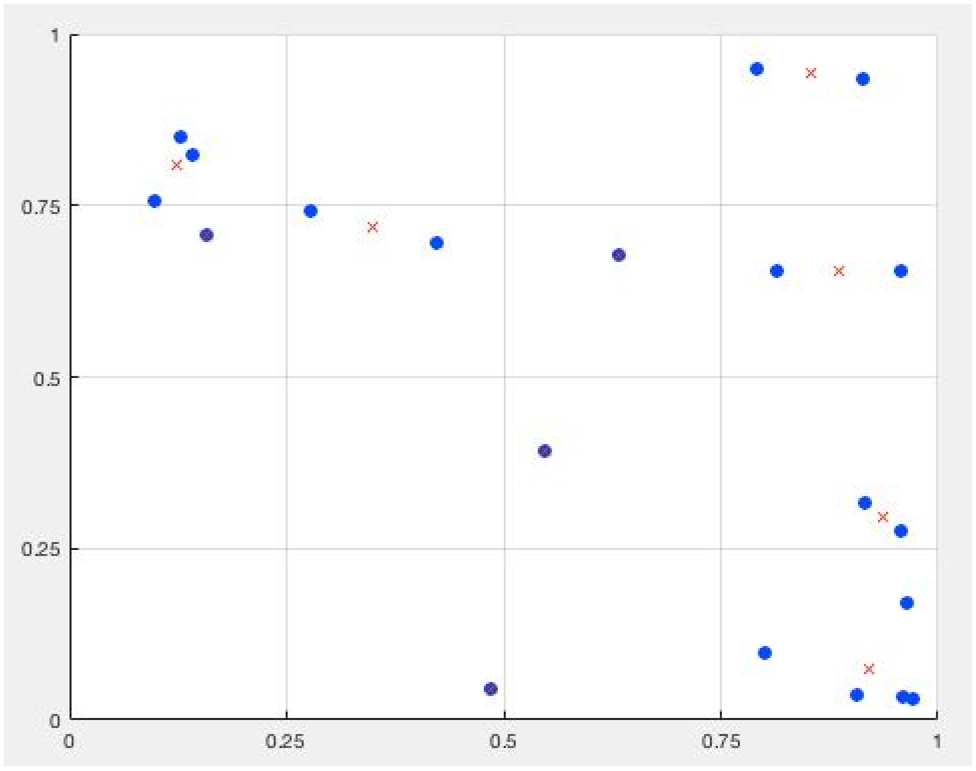
```
for binIndex = 1:Nx*Ny
    matches = 0;
    fprintf("Bin %d: ", binIndex);
    for index = 1:N
        if binNum(index) == binIndex
            matches = matches + 1;
            fprintf("%d ", index);
        end
    end
    if matches == 0
        fprintf("[]\n");
    else
        fprintf("\n");
    end
end
```

1.3 Calculations and Results

When the program is executed the following is printed in the command window:

```
Bin 1: 3 6 16
Bin 2: 11
Bin 3: []
Bin 4: []
Bin 5: []
Bin 6: 7 17
Bin 7: []
Bin 8: 14
Bin 9: []
Bin 10: 5
Bin 11: 8
Bin 12: []
Bin 13: 4 19
Bin 14: 1 9
Bin 15: 13 18
Bin 16: 2 10 12 15 20
```

The following plot is produced as well:



The averages seem to line up at the right places.

1.4 Discussion

For a very sparse set of data, the bin averages seem to be line up well, but some bins don't end up having any particles in them. Using a more dense data set, the bin averages end up more closely centered at the midpoint of each grid. This is to be expected, as more particle will naturally normalize the data.

When my code classifies a point at (0, yMax), the code says the point is in bin -4. Logically, this point should be in bin 1. In order to fix problem such as this, I would modify my algorithm by checking for edges cases such as when x or y is equal to 0 or their respective max values.

Supposing each of these particles has a maximum interaction radius of h, it would take at most N number of bins to check all particles within distance h of a particle located in Bin 1 because you need to construct a bin of h by h dimensions to check particles around a specific particle.

2. Newton's Method

2.1 Introduction

The goal of this problem is to run a zero-finding method known as Newton's Method, which is to use a initial guess of x and use the central difference approximation to calculate the the zero.

2.2 Model and Methods

The script begins by constructing a complicated cubic function, a delta value that represents the error threshold of the function, and a maximum number of evaluations:

```
complicatedCubic = @(x) 816*x.^3 - 3835*x.^2 + 6000*x + -3125;
d = 10^-6;
fEvalMax = 10;
```

Then, the script sets an initial x value and runs a while loop to evaluate and print out the data. The script prints out the initial guess, the number of evaluations the function returns, and the calculated zero value:

```
x0 = 1.43;
while x0 <= 1.71
   [xc, fEvals] = Newton(complicatedCubic, x0, d, fEvalMax);
   fprintf("x0 = %.2f, evals = %d, xc = %.6f\n", x0, fEvals, xc);
   x0 = x0 + 0.01;
end
```

The function in question is called Newton. The Newton function takes in four inputs: a value f that is a handle to a continuous function; x0, an initial guess to the root of f; delta, a positive real number; and fEvalMax, a positive integer greater than 2 that indicates the maximum number of f-evaluations allowed. The function returns 2 values, xc, the calculated x value for the zero and fEvals, the number of f-evaulations required to obtain said value.

First, the function sets xc as the inital guess and fEvals as zero. It also sets the h value for the central difference approximation:

```
xc = x0;
fEvals = 0;
h = 10^-6;
```

Then, the function runs a while loop that continuously evaluates f(x0) and its derivative to find the next x value. The while loop runs while the number of evaluations is less than the max number and that the function evaluation is greater than delta:

```
while fEvals < fEvalMax && abs(f(xc)) > delta
   fEvals = fEvals + 1;
   % evaluate f(x0) and derivative
   fx0 = f(xc);
   dx0 = (f(xc+h)-f(xc-h))/(2*h);
   xc = xc - fx0/dx0;
end
```

2.3 Calculations and Results

When the program is executed the following is outputted:

```
x0 = 1.43, evals = 5, xc = 1.470588
x0 = 1.44, evals = 4, xc = 1.470588
x0 = 1.45, evals = 4, xc = 1.470588
x0 = 1.46, evals = 3, xc = 1.470588
x0 = 1.47, evals = 2, xc = 1.470588
x0 = 1.48, evals = 3, xc = 1.470588
x0 = 1.49, evals = 4, xc = 1.470588
x0 = 1.50, evals = 6, xc = 1.470588
x0 = 1.51, evals = 10, xc = 2.023534
x0 = 1.52, evals = 8, xc = 1.470588
x0 = 1.53, evals = 4, xc = 1.562500
x0 = 1.54, evals = 3, xc = 1.562500
x0 = 1.55, evals = 3, xc = 1.562500
x0 = 1.56, evals = 2, xc = 1.562500
x0 = 1.57, evals = 2, xc = 1.562500
x0 = 1.58, evals = 3, xc = 1.562500
x0 = 1.59, evals = 3, xc = 1.562500
x0 = 1.60, evals = 4, xc = 1.562500
x0 = 1.61, evals = 6, xc = 1.666667
x0 = 1.62, evals = 8, xc = 1.470588
x0 = 1.63, evals = 7, xc = 1.666667
x0 = 1.64, evals = 5, xc = 1.666667
x0 = 1.65, evals = 4, xc = 1.666667
x0 = 1.66, evals = 3, xc = 1.666667
x0 = 1.67, evals = 3, xc = 1.666667
x0 = 1.68, evals = 3, xc = 1.666667
x0 = 1.69, evals = 4, xc = 1.666667
x0 = 1.70, evals = 4, xc = 1.666667
```

The calculated zeros values seem to be indicate that the three zeros are at x = 1.470588, 1.562500, and 1.666667.

2.4 Discussion

My results indicate that there are three zero values: 1.470588, 1.562500, and 1.666667. I was able to deduce this because those values are repeated the most in the output. However, there are a few values in between where zeros actually should be such as x = 2.023534 at x0 = 1.51 and x = 1.470588 at x0 = 1.62.

This is not fixed by the effect of delta on the zero. Making delta larger will decrease the number of evaluations required, by make the zeros are less accurate and precise. As we decrease delta, the function is more accurate and precise, but the number of evaluations increases.

The behavior of the values printed when x0 is between 1.61 and 1.62 is caused by the inaccuracies of Newton’s method. In this domain, the evaluation of the function ends up discovering an y value that is small enough to be an appropriate value.