David Curry
ID: 304755606
CEE/MAE M20
May 29, 2018

Homework 8

1 Spatial Hashing

1.1 Introduction

The purpose of this problem is to map particle data into bins to speed up the identification of particle collisions. This is done by finding the bin each random particle is in and then computing the average location of all the particles in each bin. I plotted the initial random points with a grid and the final average points on the same graph. I also printed the bin number of all the particles.

1.2 Models and Methods

In first defined the initial conditions and created the random matrix of particles by using the `rand` function. I also defined the bin spacing, number of bins, and initially defined a matrix for the bin number with the `zeros` function. This is all shown below.

```
N = 50;
x = rand(1,N);
y = rand(1,N);
xmax = 1;
ymax = 1;
h = 0.25;
Nx = floor(xmax/h);
Ny = floor(ymax/h);
dx = xmax/Nx;
dy = xmax/Ny;
binNum = zeros(1,N);
```

Next I assigned the bin number to every particle with a `for` loop. Inside the for loop I used the formula given to assign the bin number. This formula used the `ceil` function to make all the values integers. This is shown below.

```
for k = 1:N
      binNum(k) = (ceil(x(k)/dx) - 1)*Ny + ceil((ymax -y(k))/dy);
end
```

I also initialized an x bin average matrix with the `zeros` function. I then filled this matrix with average values by using a `for` loop. In the loop, I initialized a sum and counter variable so that each loop it would be updated. I then looped thru all the particles with another `for` loop. In this

nested loop I checked inthe particle was in the current bin with an `if` statement. If it was in the bin, the x position was added to the sum and the counter variable was added by one. Outside this loop, the x bin average was computed for every bin number. This entire process was repeated for the y positions of the particles, and is shown below.

```
binAvgx = zeros(1,Nx*Ny);
for k = 1:Nx*Ny
      sum = 0;
      counter = 0;
      for m = 1:N
            if binNum(m) == k
                  sum = sum + x(m);
                  counter = counter + 1;
            end
      end
      binAvgx(k) = sum/counter;
end
```

I then graphed the initial positions of the particles and the average positions on the same graph. I also included a grid with the `grid on` call and edited how many grids with the `xticks` and `yticks` function. I used `hold on` to keep all the data on the same graph and the `plot` function to graph the positions. This is all shown below.

```
grid on;
xticks(0:dx:xmax);
yticks(0:dy:ymax);
hold on;
plot(x,y,'.','MarkerSize',8);
hold on;
plot(binAvgx,binAvgy, 'x','MarkerSize', 6);
```

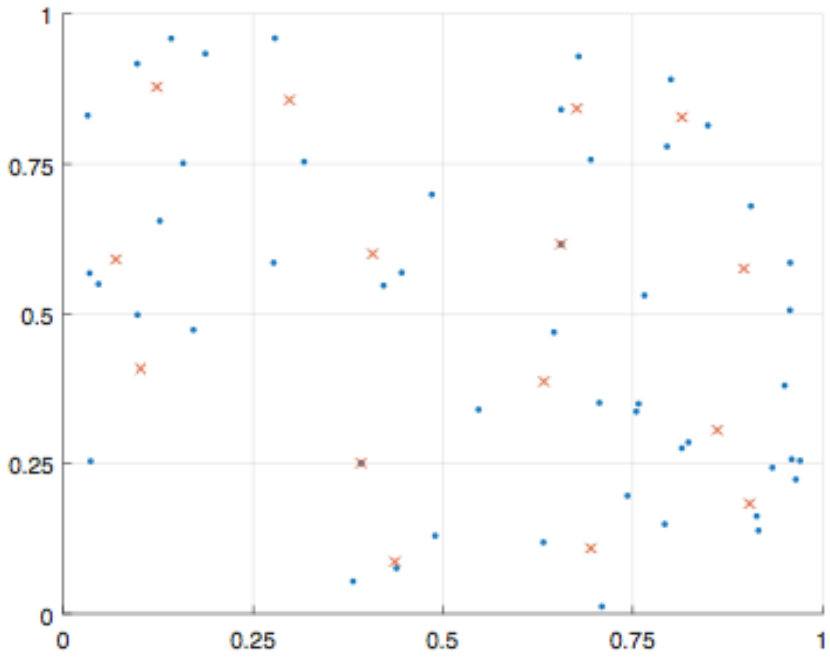
Finally I printed out the particles in each bin. This was done by iterating thu all the bins with a `for` loop. In this loop, I first defined a number variable to keep track of how many particles are in each bin, and then printed the current bin number. Then I looped thru all the particles to check if the particles were in the current bin. If the particle was in the bin, I printed that particles number on the same line as the bin number. Then after looping thru all the particles, I checked if any bin had zero particles in it. If so, the script printed out '[]' on that line. Finally, I printed a new line and repeated for every bin number. This is all shown below.

```
for k = 1:Nx*Ny
      number = 0;
      fprintf('Bin %2.0f: ',k);
      for m = 1:N
            if binNum(m) == k
                  fprintf('%.0f ',m);
                  number = number + 1;
            end
      end
      if number == 0;
            fprintf('[]');
      end
      fprintf(' \n');
end
```

1.3 Calculations and Results

To check my results with the results given, I used the default random number generator. The following is printed and plotted using the default random numbers.

```
Bin  1: 11 16 32 35 45
Bin  2: 3 34 40
Bin  3: 6 22 30
Bin  4: []
Bin  5: 7 38
Bin  6: 14 17 33 47
Bin  7: 28
Bin  8: 41 42 46
Bin  9: 21 25 37
Bin 10: 29
Bin 11: 8 31 48
Bin 12: 5 27 49
Bin 13: 15 23 44
Bin 14: 2 9 13 43
Bin 15: 1 12 20 26 36 39 50
Bin 16: 4 10 18 19 24
```



1.4 Discussion

As shown above, the script found the averages of all the particles in each bin. However this code does not work for every random number. For the case `x = 0, y = ymax`, the code assigns the bin number `-4` to the particle when it should be in the first bin. This is because of the `ceil` function used in the bin number calculation. The `ceil` of 0 is zero, when I really want it to round up to 1. To fix this I could just include an if statement for the `x = 0` case, shown below.

```
for k = 1:N
      if x(k) == 0;
      binNum(k) = (ceil(x(k)/dx))*Ny + (ceil((ymax -y(k))/dy) + 1);
      else
      binNum(k) = (ceil(x(k)/dx) - 1)*Ny + ceil((ymax -y(k))/dy);
      end
end
```

In order to check all the particles that are located within a distance *h* of a particle in Bin 1, I would need to check 3 bins: Bin 2, Bin 5, and Bin 6. This is based on the numbering convention given in the problem statement.

2 Newton's Method

2.1 Introduction

The purpose of this script is to find the zeros of any given function over a specified range. This is done with Newton's Method, which uses the derivatives of each value of the function to find the zeros. I created a separate function script for the method, and then called this function in another script to find the zeros for a range of initial x values on a given function. Finally I printed these results in a table.

2.2 Models and Methods

I first created a function called `Newton` in a separate script with the `function` call shown below.

```
function [xc, fEvals] = Newton(f, xo, delta, fEvalMax)
```

Inside this function, I first defined the constant `h` and the initial value for `fEvals`. I then used a `while` loop to iterate until the function was less than `delta` or the number of iterations was larger than `fEvalMax`. In the loop, I used the central difference approximation formula to find the derivative at the current point. I then found the next point using the given formulas and reset `xo`. I also increased the counter variable by 1 each iteration. This is all shown below.

```
h = 10^-6;
fEvals = 0;
while abs(f(xo)) > delta && fEvals < fEvalMax
      f_prime_x = (f(xo + h) - f(xo - h))/(2*h);
      xc = xo - (f(xo)/f_prime_x);
      xo = xc;
      fEvals = fEvals + 1;
end
```

I then created a new script and defined some initial conditions. I chose 10^{-6} for `delta` and 50 for `fEvalMax` because the output for the zeros was to 6 decimal places and 50 seemed reasonable amount of iterations to stop at. I also defined the function that I was finding the zeros for. This is all shown below.

```
change = 0.01;
delta = 10^-6;
fEvalMax = 50;
xo = 1.43;
xf = 1.71;
f = @(x) 816*x^3 - 3835*x^2 + 6000*x - 3125;
```

Finally I iterated through all the `xo` values to check for zeros near with a `for` loop. Inside the loop, I called the `Newton` function to get the `xc` and `fEvals` values and then printed these with the `fprintf` function. This is all shown below.

```
for k = xo:change:xf
      xo = k;
      [xc, fEvals] = Newton(f,xo,delta,fEvalMax);
      fprintf('xo = %4.2f, evals = %2.0f, xc = %.6f\n',xo, fEvals,xc);
end
```

2.3 Calculations and Results

After running the code with the above initial conditions, the following is printed.

```
xo = 1.43, evals =  5, xc = 1.470588
xo = 1.44, evals =  4, xc = 1.470588
xo = 1.45, evals =  4, xc = 1.470588
xo = 1.46, evals =  3, xc = 1.470588
xo = 1.47, evals =  2, xc = 1.470588
xo = 1.48, evals =  3, xc = 1.470588
xo = 1.49, evals =  4, xc = 1.470588
xo = 1.50, evals =  6, xc = 1.470588
xo = 1.51, evals = 18, xc = 1.666667
xo = 1.52, evals =  8, xc = 1.470588
xo = 1.53, evals =  4, xc = 1.562500
xo = 1.54, evals =  3, xc = 1.562500
xo = 1.55, evals =  3, xc = 1.562500
xo = 1.56, evals =  2, xc = 1.562500
xo = 1.57, evals =  2, xc = 1.562500
xo = 1.58, evals =  3, xc = 1.562500
xo = 1.59, evals =  3, xc = 1.562500
xo = 1.60, evals =  4, xc = 1.562500
xo = 1.61, evals =  6, xc = 1.666667
xo = 1.62, evals =  8, xc = 1.470588
xo = 1.63, evals =  7, xc = 1.666667
xo = 1.64, evals =  5, xc = 1.666667
xo = 1.65, evals =  4, xc = 1.666667
xo = 1.66, evals =  3, xc = 1.666667
xo = 1.67, evals =  3, xc = 1.666667
xo = 1.68, evals =  3, xc = 1.666667
xo = 1.69, evals =  4, xc = 1.666667
xo = 1.70, evals =  4, xc = 1.666667
xo = 1.71, evals =  5, xc = 1.666667
```

2.4 Discussion

As shown above with a `delta` of 10^{-6}, all of the 3 zeros are accurate up to 6 digits if the zero showed up multiple times in the output. Decreasing `delta` increases the number of evals and decreases the accuracy. For example, changing `delta` to 10^{-3} decreases the accuracy to only the third decimal place and increases the number of `evals` by 1 for all. Looking at the output, there

is a drastic change in zero values between `xo = 1.61` and `xo = 1.62`. This is because 1.62 is very close to a local minimum of the function so the derivative is very close to zero. Therefore the next point from the formula shown below will be far enough away that it will pass the 1.562500 zero and catch the 1.470588 zero.

```
xc = xo - f(x0)/f'(xo)
```