CS1112 Spring 2009 Project 2 due Thursday 2/12 at 11pm

You must work either on your own or with one partner. You may discuss background issues and general solution strategies with others, but the project you submit must be the work of just you (and your partner). If you work with a partner, you and your partner must first register as a group in CMS and then submit your work as a group.

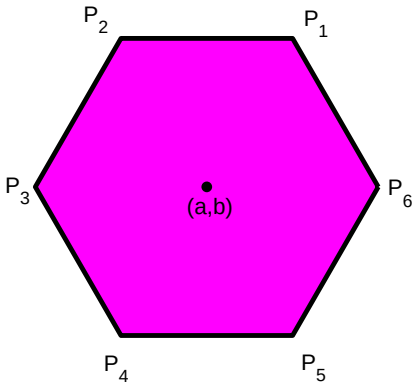
Objectives

Completing this project will help you learn about `for`-loops, `while`-loops, the `if-elseif` construction, boolean expressions, and various built-in functions such as `rem`, `floor`, `ceil`, and `rand`.

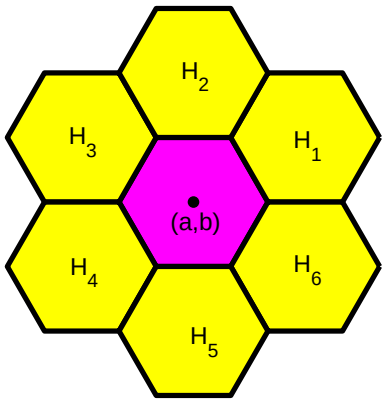
Honeycomb Facts

A *unit hexagon* centered at (a, b) has vertices

$$
\begin{array}{lcl}
P_1 & : & (a + \Delta_x, b + \Delta_y) \\
P_2 & : & (a - \Delta_x, b + \Delta_y) \\
P_3 & : & (a - 1, b) \\
P_4 & : & (a - \Delta_x, b - \Delta_y) \\
P_5 & : & (a + \Delta_x, b - \Delta_y) \\
P_6 & : & (a + 1, b)
\end{array}
$$



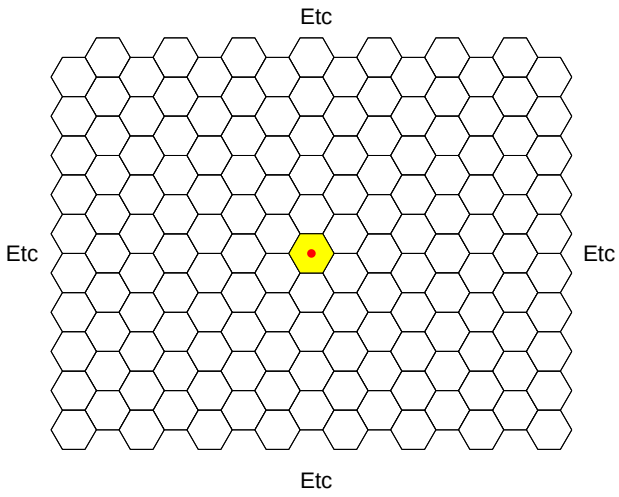
where $\Delta_x = 1/2$ and $\Delta_y = \sqrt{3}/2$. Thus, a unit hexagon has unit edge length and a diameter equal to two. A unit hexagon has six unit hexagon neighbors,



and here are their centers:

$$
\begin{array}{lcl}
H_1 & : & (a + 3\Delta_x, b + \Delta_y) \\
H_2 & : & (a, b + 2\Delta_y) \\
H_3 & : & (a - 3\Delta_x, b + \Delta_y) \\
H_4 & : & (a - 3\Delta_x, b - \Delta_y) \\
H_5 & : & (a, b - 2\Delta_y) \\
H_6 & : & (a + 3\Delta_x, b - \Delta_y)
\end{array}
$$

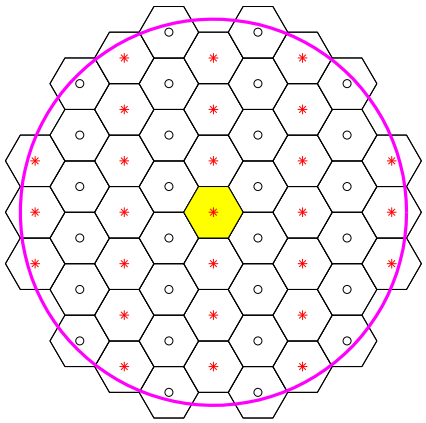
We can tile the xy-plane with unit hexagon tiles:



In order to fix the location of all the tiles, we center one of them over the origin. We refer to that tile as the *home tile* and to the overall pattern as the *infinite honeycomb*.

1 Near to Home

Write a script `Near2Home` that solicits a positive number r via `input` and prints the number of tiles in the infinite honeycomb whose centers are within r of the origin. Thus, a tile with center (a, b) is counted if $a^2 + b^2 \leq r^2$. Here is a picture that illustrates the $r = 6.5$ situation:



In this example, there are 55 tile centers inside the circle.

You are free to count tiles by any method in your solution script. However, we now outline a `for`-loop method that is similar in spirit to the §2.1 example in *Insight*.

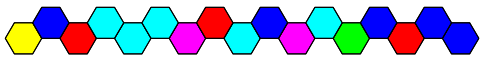
Since tiling with hexagons is a little more complicated than tiling with squares, you should start by understanding how the tiles are laid out in the infinite honeycomb. To help with this we have marked the tile centers in the above example. The vertical spacing between the tile rows is given by Δ_y. Indeed, the y-coordinate of the k-th row is given by $k\Delta_y$. For the $r = 6.5$ case, k ranges from -7 to +7. (The home tile is in row 0.) This suggests a `for`-loop strategy with a count variable that steps through all the rows, e.g., `for k=-7:7`. Of course, your script will have to figure out the beginning and ending values for the loop index. It will depend on the input radius r.

The body of the `for`-loop has to compute the number of tiles in the row "named" by the current value of the count variable and then add the result into a running sum. In the displayed $r = 6.5$ problem, we see that there are five tiles to count in row 0, three tiles to count in row -4, and four tiles to count in row 5. Observe that the calculations are different depending upon whether the row index is even or odd. In the even-indexed rows where the tile centers are marked with "`*`", there is a tile with center on the y-axis. In the odd-indexed rows where the tile centers are marked with "`o`", the tiles are "shifted" horizontally. The tiles in any row have center-to-center spacing equal to three. Your script will have to work with all these facts. Recall that the `rem` function can be used to determine if an integer is divisible by 2. The built-in functions `floor` and/or `ceil` are handy for deciding how may tiles you can "fit" across a given line segment.

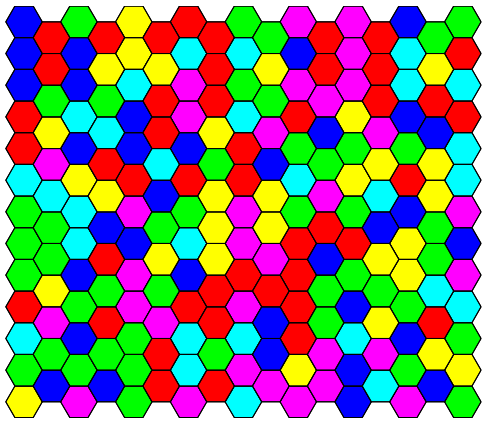
Although you are free to exploit symmetry as in the textbook script `Eg2_1`, it is not required. Organize your script in a way that is most clear to you, making sure to include comments so that we can trace your reasoning. Submit your file `Near2Home.m` in CMS.

2 Bee Hive

A function `DrawHex(a,b)` is available on the course website. This function draws a randomly colored unit hexagon with center (a, b). Don't worry about the code in `DrawHex`; you only need to use it for drawing as demonstrated in the script `BeeHive`, which is also available on the course website. To run the script `BeeHive`, both `BeeHive` and `DrawHex` should be in MATLAB's current directory. `BeeHive` uses `DrawHex` to produce the following image:



Read `BeeHive` carefully and then modify it so that (except for color) it draws in the figure window a honeycomb *exactly* like this:



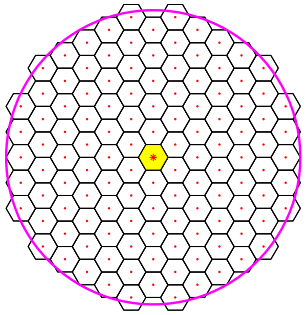
The lower left hexagon should be centered on the origin. One way to organize your solution is to write a nested loop that displays all the rows with nine hexagons and another nested loop that displays all the rows with eight hexagons. Useful facts are (a) the vertical distance between rows is Δ_y, (b) the leftmost hexagon on a nine-hexagon row has its center on the line $x = 0$, and (c) the leftmost hexagon on an eight-hexagon row has its center on the line $x = 3/2$.

Regarding the nesting of loops, one loop is required for the job of drawing all the hexagons in one row. In order to draw multple rows, another loop—an outer loop—is necessary to name all the rows.

You are free to solve the problem any way that you want. However, for full credit your solution script must make effective use of the nested-loop idea. Submit your modified `BeeHive.m` to CMS.

3 Random Exit

A Queen and several thousand worker bees live in a radius-R hive. Here is the $R = 10$ layout:



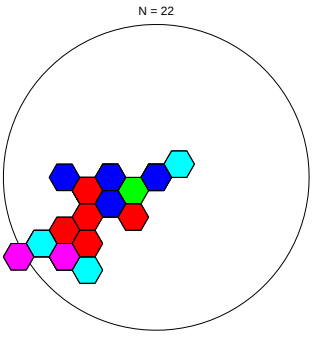
After meeting with the Queen at (0,0) to complain about workload, a worker bee W decides to transfer to another hive. To avoid being followed, W adopts an exit strategy that involves hopping in random directions from one tile to the next. The hopping continues until a tile is reached with center (a, b) that satisfies $a^2 + b^2 > R^2$. The hopping strategy is simple. When W is "standing" on a particular tile, it hops with equal probabilty to one of the neighbor tiles H_1, H_3, H_4 or H_6. (Now is the time to review the notion of "neighbor tiles" presented in the opening section.)

The essential task in this problem is to write a `while`-loop that simulates W's exit and enables you to track the journey graphically. A template script `RandomExit` is provided on the course website. It includes the necessary graphics commands so that you can watch the simulation.

Use a pair of variables `a` and `b` to house the xy coordinates of W's location. At the start, `a = 0` and `b = 0`. The following should happen during each pass through the `while`-loop.

1. Display W's current location by calling `DrawHex`.
2. After the call to `DrawHex` pause the program for 0.1 second so that you can see the current location.
3. Using `rand` and the `if-elseif` construction, determine the next tile. The values in `a` and `b` should be updated accordingly. (Refer to the table above that specifies the centers of the neighbor tiles.)
4. Increment a counter `N` so that when the loop terminates, its value is the number of hops made. Of course, this variable must be properly initialized at the start.
5. Include the statement `title(sprintf('N = %1d',N))` at the bottom of the loop body. This way you can watch the journey's "odometer" at the top of the figure window as the simulation progresses.

Your script should also display the exit tile, i.e., the tile outside of the hive that W eventually reaches. Here is sample output for an $R = 10$ problem:



Notice that the hopping rules make it possible for W to return to a previously visited tile. That is why there are fewer than 22 tiles displayed in the above tracking. You should play with the value of `R`. The average number of required hops grows as the square of R so you do not want R too big! Submit your modified `RandomExit.m` to CMS.