Shannon Nguyen
MATH 2605
4/20/2014

Pinball

Implementation

The Pinball Project required two different files for me to code: PinballCalculator.java and Pinball.java. PinballCalculator was responsible for the actual implementation of the algorithms of this particular project, whereas Pinball acted as a driver file. Java was used due to the JAMA package's Matrix class, which contained the matrix math methods I needed. Though the Pinball project utilized vectors, the vector values needed were simply translated into matrices of one row and however many necessary lengths (typically two). The JAMA package was developed by The MathWorks and the National Institute of Standards and Technology. Further citation and a link to the source can be found in the Citations section.

As previously stated, the PinballCalculator contains all of the necessary algorithms needed to calculate orbits for the project. The intersect() method determines intersection with any of the predefined circles, reflect() calculates the reflected velocity, genRandVelocity() generates a random initially velocity within 0 and 2π, and genSystVelocity() generates equally spaced velocities between 0 and 2π (the divisions depend on the number of test cases). There is also a constructor and numerous getter and setter methods that are used to access and edit certain private variables.

The constructor takes in two parameters: the side length as double s and the radius as double r. The constructor's purpose is to initialize the circles Matrix c1, Matrix c2, and Matrix c3, as well as the position x and the side length and radius as they will be used in the calculations. The position was set to a Matrix of {0, 0} and the three circles are defined by the project specifications. The program itself also has another global variable double initAngle that is not initialized in the constructor, but is defined by calls to the methods genRandVelocity() and genSystVelocity().

intersect() takes in 4 variables: a Matrix c, the circle to calculate an intersection on, a double r, the radius, a Matrix x, the position, and another Matrix v, the velocity. The method returns another matrix, but this matrix represents the time that will be used to determine the next position in the iteration when running the algorithm. The method's variables are:

- Matrix cMinusX, which represents the c – x component of the equation
- double dotVCMX, which is the value of the dot product of v and c-x
- double d, which is the value used to hold D as it is calculated from the equation

The method reflect() takes in 3 parameters: Matrix c, which represents the circle to reflect from, Matrix x, the position to reflect from, and the Matrix v, which is the old velocity. The method returns another Matrix that holds the new reflected velocity. The variables are:

- Matrix cMinusX, which represents the c – x component of the equation
- double dotVCMX, which is the value of the dot product of v and c-x
- Matrix w, which holds the matrix to be returned

genRandVelocity() has variables double angle and double[][] v. getSystVelocity() takes in a double angle that was calculated in a method in the Pinball class. Both return a Matrix v that is the initial velocity to be used. The initial angle generated or taken in depending on the method is stored in the global variable double initAngle.

The Pinball class has the tests() method as well as the main method. Because of the simplicity of the output and required test cases for the program, I thought that it'd be most appropriate to run the program through the console instead of through an applet. The only downsides to this are that for console windows that are too small, the output text may end up being wrapped around and making it more difficult to discern the outputs. However, the issue wasn't severe enough that I though it warranted an applet.

The test() takes in 3 parameters: double rVal, which is the appropriate radius depending on the test case, int tests, the number of tests to be run, and boolean isRandom which is used to determine whether genRandVelocity() or genSystVelocity() will be called. The method uses several variables in order to assist with its output:

- PinballCalculator pb, which is an instance of the PinballCalculator in order to use the appropriate equation methods
- ArrayList<String> circles, which holds the list of circles that are intersected for each iteration
- double c1t, which holds the time when calculating the intersection for circle 1
- double c2t, which holds the time when calculating the intersection for circle 2
- double c3t, which holds the time when calculating the intersection for circle 3
- int[] maxTen, which holds all the possible values in order to decide the maximum ten hits
- double[] tenAngles, which holds the angles of a hit
- ArrayList<String>[] circlesHit, which holds all of the intersected circle sequences for a hit
- int[] freq, which holds all the possible frequencies for hits
- int hits, the actual number of hits that an iteration goes through
- Matrix v, which is the velocity to be used to go through each calculation
- double angle, which is the angle passed into genSystVelocity()
- double spacing, which is the number to increment by when generating a systematic velocity
- double mint, which is the minimum time as determined from circle intersections
- ArrayList<String> tempCircles which is used to hold the circles value in order to prevent the value from being overwritten later on

The method prints out a list of hits and their frequencies, as well as the top ten hits (there will be fewer than ten if there are less than ten types of hits) and the last angle used to get that many hits and the last circle intersection sequence that was generated by that many hits.

The main method is used to simply prompt the user for how many trials they want. Only integer values are allowed. The method will go through iterations for s = 6, r = 1, and s = 6, r = 2 for both randomly generated angles and systematic angles, resulting in a total of 4 different results.

Project Discussion

For a run of one million iterations through the algorithm, I achieved the results below. They are labeled by the number of the report analysis required by the project requirements.

1.) The relative frequency (one million trials)

s = 6, r = 1 – 1,000,000 random angles		
Hits	Frequency	Relative Frequency
0	357,433	0.357433
1	459, 386	0.459386
2	131, 427	0.131427
3	37, 363	0.037363
4	10, 341	0.010342
5	2,886	0.002886
6	829	0.000829
7	235	0.000235
8	77	0.000077
9	17	0.000017
10	4	0.000004
11	3	0.000002

s = 6, r = 1 – 1,000,000 systematic angles		
Hits	Frequency	Relative Frequency
0	536,726	0.536726
1	463,274	0.463274

s = 6, r = 2 – 1,000,000 random angles		
Hits	Frequency	Relative Frequency
0	272,421	0.272421
1	419,992	0.419992
2	221,764	0.221764
3	79,541	0.079541
4	6,104	0.006104
5	178	0.000178

s = 6, r = 2 – 1,000,000 systematic angles		
Hits	Frequency	Relative Frequency
0	408,052	0.408052
1	591,947	0.591947
2	1	0.000001

It is evident by the tables that the method that delivered the longest orbit was using a radius of 2 with randomly generated starting velocities. This is probably attributed to the fact that the random selection of angles allowed for greater variation, but the fact that the radius is smaller provides it will less “room” for intersection. Using targeted angles ultimately gave fewer hits for

both r = 1 and r = 2, and r = 2, though giving a greater proportion of hits, also had less variation in the number of hits and usually did not require numerous hits to exit.

2.) The top 10 hits, with initial velocity and circle hit sequence (one million trials) – Again, if there were fewer than 10 top hit values, fewer than 10 may appear

s = 6, r = 1 – 1,000,000 random angles		
Hits	Initial Angle (Radians)	Sequence
2	3.461724	3,1
3	0.025149	3,1,2
4	0.837544	2,3,2,1
5	5.162778	3,2,3,2,1
6	1.986471	2,3,2,3,2,3
7	3.962122	1,3,2,3,2,3,2
8	0.994322	1,2,3,2,3,2,3,2
9	3.995447	3,2,3,2,1,2,1,2,1
10	0.498108	3,2,1,2,3,2,1,3,1,3
11	2.039589	1,3,2,3,1,3,1,2,3,2,3

s = 6, r = 1 – 1,000,000 systematic angles		
Hits	Initial Angle (Radians)	Sequence
0	3.372343	-
1	6.283179	2

s = 6, r = 2 – 1,000,000 random angles		
Hits	Initial Angle (Radians)	Sequence
0	1.545350	-
1	6.228252	3
2	5.924668	1,2
3	5.470980	3,1,3
4	4.225501	1,2,3,1
5	4.911258	2,3,1,3,2

s = 6, r = 2 – 1,000,000 systematic angles		
Hits	Initial Angle (Radians)	Sequence
0	3.087463	-
1	6.283179	1
2	6.172677	2,1

The best orbits seen here were all determined by brute force. From the looks of it, a greater angle typically gives a higher amount of hits. Lower hits also have a tendency to be closer to π, though more tests would need to be completed in order to reach a definitive conclusion.

Citations

- JAMA: http://math.nist.gov/javanumerics/jama/
    - Specifically for this project, the following files were used:
        - Matrix.java