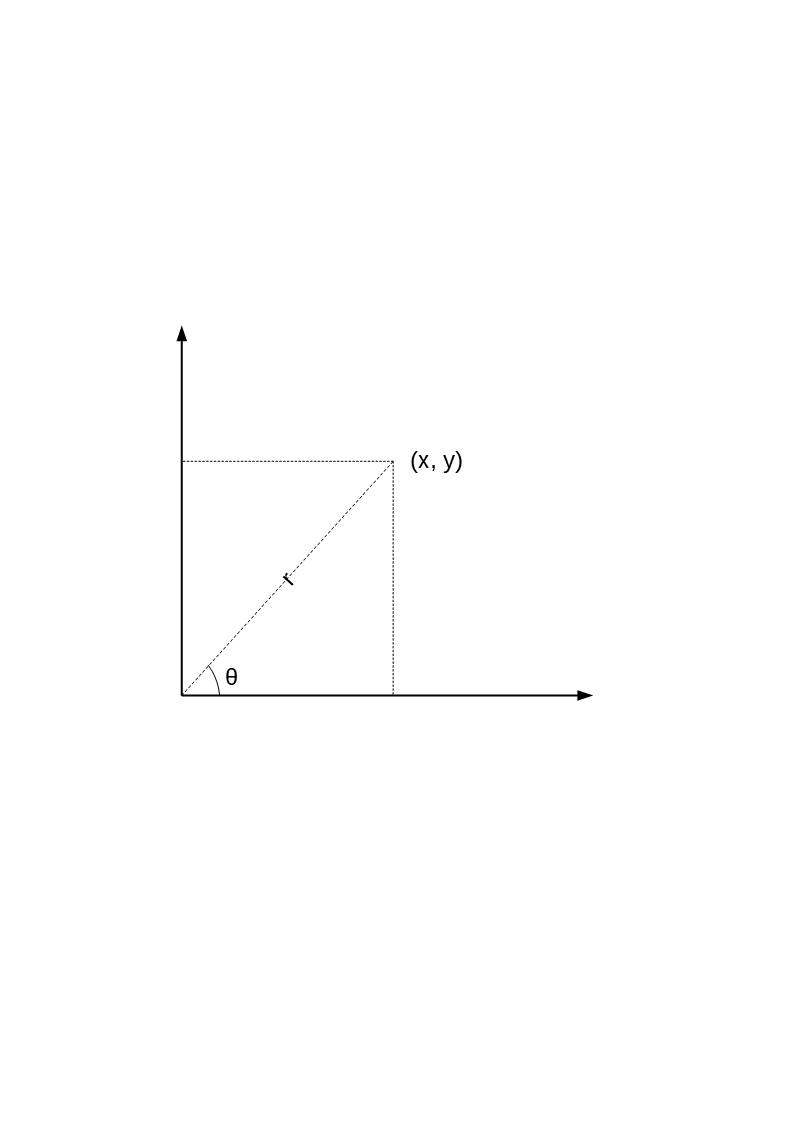
# Motivation

This is a simple yet exciting project idea I came up with in office. Recently, my colleague’s son fell ill with the dengue fever, and on kids the effect is huge. Their immunity is low, and platelets decrease rapidly below danger levels. Dengue is a mosquito spread disease, and mosquitoes are a big issue in India, and its spread and control is quite difficult due to insufficient data on the spread of mosquitoes.

Here, I present an idea that I am working on from the past weeks that could simulate the spread of dengue fever using a mosquito simulation program. So, let’s understand what we have and what we know and build upon it. In the simulation one major aspect is to code the random walk of the mosquitoes on a plane.

I took an approach of performing 2-dimensional random walks on a hypothetical plane using polar coordinates. Those of you who are unaware of polar coordinates, here is a small brief. Instead of using the generic (x, y) notation to locate a point on a plane we use an angle and distance (r,θ) approach.

This makes the simulation a problem. In this article we will discuss on this particular problem and look at some solutions I had come up with. 2-dimensional random walks are already done using an x-y grid in which the particle which is subjected to randomness can move in any 4,8,12 or 24 directions. Mostly people pick 4 directions and keep the step size so small that it seems the particle is moving in all directions including diagonals perfectly. This is a popular digital display trick heavily used today. Display devices are made with higher resolution to show diagonals and curves as accurately as possible but this accuracy is only perceived accuracy and not true accuracy. This is also complemented by anti-aliasing. This is a topic on digital display technologies, which means I’m going off topic.

Mosquitoes or any living insect tend to move away from their source or point of birth, they may visit it back but in-order to spread they must move away. Their spread is equal in all directions. Which means that the random walk must be biased. It can’t be constantly biased as they may find a chance to come back home after a farther travel. So, the distance the mosquito has traveled from home, or lets take displacement of the mosquito from home, which is easier to visualize, affects the biasing of the random walk. So, the closer the mosquito is to home the more biased the random walk would be to push it away from home, and the farther it is from home the lesser biased it would be.

Thus we have 2 problems:

1. 2-dimensional dynamically biased random walks
2. Using polar coordinates

This is only the tip of the iceberg, maybe only a part of it. And in the upcoming articles I will expand the picture and explain the possibilities of this simulation.

# 2-dimensional dynamically biased random walks

Here we use processing library as it is a simple and easy-to-use library written in Java. I am using this library only in this step.

// mosquitto\_sim.pde

int init\_n\_mosquittos = 1000;

ArrayList<Mosquitto> mosquittos = new ArrayList<Mosquitto>();

ArrayList<Mosquitto> next\_mosquittos = new ArrayList<Mosquitto>();

int new\_index = init\_n\_mosquittos;

int show\_after = 1;

int next\_show = 0;

void setup() {

size(960, 720);

frameRate(45);

for (int i = 0 ; i < init\_n\_mosquittos ; i++) {

mosquittos.add(new Mosquitto(i));

}

}

We create 1000 mosquitoes first and assign them to a ArrayList called “mosquittos”. Sorry for the misspelled name as I was unaware of the spelling when I wrote this code. “next\_mosquittos” contains the list of newly added mosquitoes on birth. We shall see how that works. This reproduction is a basic cycle only and will be more sophisticated as we dive deep into the project over the next few blogs.

As I was running this on my laptop which is an old spec machine I stuck to 960x720 resolution screen with framerate of 45.

// mosquitto.pde

class Mosquitto {

private float r = random(0,1);

private int index;

private float angle = random(0,2) \* PI;

private float x = width/2 + r \* cos(angle);

private float y = height/2 + r \* sin(angle);

private float source\_x = x;

private float source\_y = y;

private float speed = random(0.1, 20);

private float spread\_angle = 0;

private int steps = 0;

private int totalSteps = floor(random(1500, 2500));

private int totalDistance = floor(random(350, 450));

private int reproduceSteps = floor(random(950, 1100));

private int reproduceDistance = floor(random(1, 10));

private boolean canReproduce = true;

private boolean kill = false;

Here, we created a class called “Mosquitto”. I contains the following properties:

* r – showing the initial distance of the mosquito from its point ot origin
* index – the index of the mosquito, a unique identifier
* angle – a random angle at which it starts moving [0 - 2π]
* x – the x-coordinate of the mosquito in Cartesian coordinates so that it can be plotted [x = r\*cos(angle)]. Note: here we have added width/2 as the coordinates are made from top-left corner to bottom-right corner and we want the mosquitoes to start from the center.
* y – the y-coordinate of the mosquito in Cartesian coordinates. [y = r\*sin(angle)].
* source\_x, source\_y – to hold the origin position of the mosquito.
* speed – is the number of steps the mosquito covers on each iteration.
* spread\_angle – an angle that adjusts the biasing in the random walk.
* Steps – counts the number of steps taken by the mosquito
* totalSteps – [constant] total number of steps that the mosquito can take before its end of life
* totalDistance – [constant] maximum displacement of the mosquito from its point of origin
* reproduceSteps – [constant] minimum number of steps before the mosquito is eligible for reproduction
* reproduceDistance – [constant] minimum distance between two mosquitoes to allow reproduction
* canReproduce – [boolean] signifying if the mosquito is eligible for reproduction or not
* kill – signifying if the mosquito is to be killed before the next iteration or not

// mosquitto.pde

Mosquitto(int index) {

this.index = index;

}

Mosquitto(int index, float x, float y) {

this.index = index;

this.x = x;

this.y = y;

this.source\_x = x;

this.source\_y = y;

}

The first constructor creates a new Mosquitto instance based on the index. The second constructor creates a new Mosquitto instance based on the given point of origin and new index value.

// mosquitto.pde

void show() {

fill(0,0,0,150);

noStroke();

ellipse(x, y, 10, 10);

}

A member function to display the particular mosquito using the x and y coordinates.

// mosquitto.pde

void update() {

float distance\_from\_source = sqrt(sq(x - source\_x) + sq(y - source\_y));

float spread = 1 / (1 + exp(-0.01 \* distance\_from\_source)) \* 2.0 - 1.0;

this.spread\_angle = spread;

this.angle = this.angle + random(-spread, spread) \* PI;

x = x + speed \* cos(angle);

y = y + speed \* sin(angle);

distance\_from\_source = distance(x, y, source\_x, source\_y);

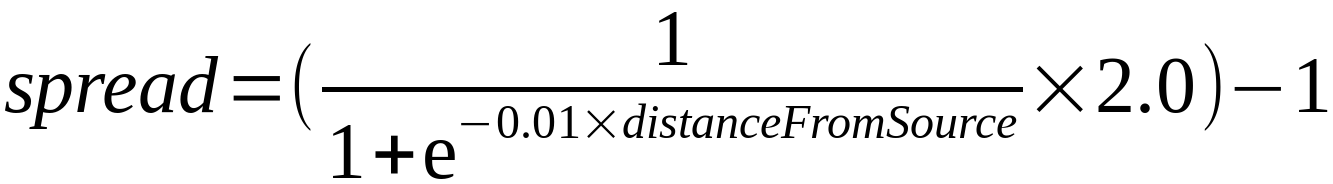
steps++;

reproduce();

die();

}

This update function is called on every mosquito at every iteration. First, distance from source is calculated. The spread\_angle is calculated based on the distance from source. This is mapped to [0,2π].

This is an asymptotic function with a domain of (-∞,∞) and a range of [0, 2).

We then call the reproduce and die methods.

void reproduce() {

if (steps > reproduceSteps && canReproduce) {

for (int i = 0 ; i < mosquittos.size(); i++) {

if (i != index && distance(x, y, mosquittos.get(i).x, mosquittos.get(i).y) < reproduceDistance && partnerCanReproduce(mosquittos.get(i))) {

next\_mosquittos.add(new Mosquitto(new\_index, x, y));

//next\_mosquittos.add(new Mosquitto(new\_index+1, x, y));

new\_index += 1;

canReproduce = false;

}

}

}

}

In reproduce we perform a prilimnary test which proves if the mosquito can reproduce or not. Then, it looks for a mate within the “reproduceDistance” and checks if it can reproduce or not. If the cases pass then a new mosquito is added to the “next\_mosquitto” list so that it can be appended to the primary “mosquitto” list after all the mosquitoes have gone through the analysis. Once a mosquito reproduces then it cannot reproduce. Although, this is not true in a real-world scenario, but to make the proof-of-concept simple I have kept this parameter.

void die() {

if(steps > totalSteps || distance(x, y, source\_x, source\_y) > totalDistance) {

kill = true;

}

}

The die function is very simple.