**12-15-15**

Before 64-bit installed and area scaling added.

Monarch.java

**public** **class** Monarch {

**private** String name;

//Variables that can be changed

//distance monarch moves in each step

**private** **int** steplength = 50;

//directionality parameter constrains possible changes in direction

**private** **double** directionality = 0.75;

//perception distance in meters

**private** **double** perception = 100;

//number of eggs laid per step

**private** **double** eggsperlay = 4;

//probability a monarch jumps any given step

**private** **double** jumpProb = 0.025;

//jump distance

**private** **double** jumpLength = 300;

//parameters for distribution of random jump headings

//sample from normal dist with mean NNE (22.5 deg = 0.3927 rads)

//which in math degrees is 67.5 deg = 1.1781 rads

**private** **double** mean = 1.1781;

**private** **double** sigma = 0.5;

//equations for daily movement, daily eggs, and distance\*probEggs are below

//current angle in radians that Monarch is moving - initialize to random value

**private** **double** currAngle = 2\*Math.***PI***\*Math.*random*();

//magnitude of change in direction

**private** **double** angleChange;

//temp variable needed when angle crosses 360/0 degrees

**private** **double** tempAngle1;

**private** **double** tempAngle2;

//number of steps taken each day

**private** **double** cumSteps = 0;

//the number of times a monarch can lay eggs in a day

**private** **int** numtimeslayeggs;

//number of eggs that can be laid on a given day

**private** **double** dailyeggstolay = 0;

//number of eggs laid per day

**private** **double** dailyeggslaid = 0;

**private** **double** numtimeslayeggs2;

//dist in m between monarch and nearest point of zoneagent, calc'd from lat/long units

**private** **double** dist;

//initialized variables for each step - cumulative distance and maximum dist can move

**private** **double** cumDist = 0;

**private** **double** maxdist = 0;

//current coordinates of Monarch agent

**private** **double** x;

**private** **double** y;

//xy boundaries of the geographic area in lat/long

**static** **double** *xmin* = -93.50925;

**static** **double** *xmax* = -93.45775;

**static** **double** *ymin* = 42.04425;

**static** **double** *ymax* = 42.0964;

//get context and geography

**public** **static** GeometryFactory *fac* = **new** GeometryFactory();

**public** Monarch(String name) {

**this**.name = name;

}

//**TODO** set up these methods for easier to read code

//method to decide which polygon the monarch should move towards

**public** **void** movetoPoly() {

}

**public** **void** getPolyProbs() {

}

**public** **void** layEggs() {

}

//Each step/tick is a day in the life of an egg-laying monarch

@ScheduledMethod(start = 1, interval = 1, priority = ScheduleParameters.***FIRST\_PRIORITY***)

**public** **void** step(){

//System.out.println("Monarch " + this.name());

//initialize these variables to 0 each day

cumDist = 0;

cumSteps = 0;

dailyeggslaid = 0;

//gets current tick

**double** tick = RepastEssentials.*GetTickCount*();

//calculates max distance that can moved during current tick

maxdist = -500\*tick + 10500;

//potential eggs to lay each day - currently starts at 50 and drops to 30

dailyeggstolay = -2\*tick + 52;

//System.out.println("Noeggstolayperday "+ dailyeggstolay);

//number of times a monarch can lay eggs each day if it lays 4 eggs each time

**double** numtimeslayeggs1 = dailyeggstolay/eggsperlay;

//System.out.println("NotimestolayperdayD "+ numtimeslayeggs1);

//number of times a monarch can lay eggs rounded up to nearest integer and

//cast as integer

numtimeslayeggs2 = Math.*ceil*(numtimeslayeggs1);

//System.out.println("NotimestolayperdayCeil "+ numtimeslayeggs2);

numtimeslayeggs = (**int**) numtimeslayeggs2;

//System.out.println("NotimestolayeggsFinal "+ numtimeslayeggs);

//boolean to determine when while loop ends

**boolean** doneMove = **false**;

//failsafe counter to stop while loop

**int** moveCount = 0;

**while** (! doneMove) {

**if** (numtimeslayeggs >= 0) {

moveCount++; //increment emergency shutdown timer

Context context = ContextUtils.*getContext*(**this**);

Geography<Object> geography = (Geography)context.getProjection("Monarchs");

//first coord for network display - Dr. Parry code

Geometry geom = geography.getGeometry(**this**);

Coordinate c1= geom.getCoordinates()[0];

//System.out.println("c1 = mon = " + c1);

Coordinate c2 = **null**;

//decide whether to make random jump or not

**double** toJump = Math.*random*();

//double jumpHeading = 2\*Math.PI\*Math.random();

**double** jumpHeading;

**do** {

jumpHeading = Normal.*staticNextDouble*(mean, sigma);

//System.out.println("first first " + jumpHeadingtest);

} **while** ((jumpHeading > mean + 3.13) && (jumpHeading < mean - 3.13));

//System.out.println("first " + jumpHeadingtest);

**if** (jumpHeading < 0){

jumpHeading = jumpHeading + 2\*3.13;

}

**if** (jumpHeading > 2\*3.13){

jumpHeading = 2\*3.12;

}

//System.out.println("second " + jumpHeading);

//Jump

**if** (toJump < jumpProb) {

cumDist = cumDist + jumpLength;

geography.moveByVector(**this**, jumpLength, jumpHeading);

//Didn't jump

} **else** { //if not, get polygons, decide which to move towards

ArrayList probs = **new** ArrayList(); //normalized pref/p values

ArrayList dists = **new** ArrayList(); //distances in lat/long units

ArrayList destX = **new** ArrayList(); //array for latitude coord of destinations

ArrayList destY = **new** ArrayList();

ArrayList areaPoly = **new** ArrayList(); //area of polygons

**double** runningsum = 0; //sum of pdist's to normalize p's

//retrieve objects within perception distance

GeographyWithin within = **new** GeographyWithin(geography, perception, **this**);

//loops through objects within perception distance

**for** (Object obj : within.query()) {

//if an object is a ZoneAgent, then cast it as a ZoneAgent

**if** (obj **instanceof** ZoneAgent){

ZoneAgent zoneagent = (ZoneAgent)obj;

//System.out.println("ZoneAgentID = " + System.identityHashCode(zoneagent));

//System.out.println("ZoneAgentName = " + zoneagent.getName());

//creates a polygon geometry of this zone

Polygon zonegeom = (Polygon)geography.getGeometry(zoneagent);

//gets the lat/long of closest point in polygon, stores in array

//DistanceOp Op = new DistanceOp(geom, zonegeom, perception);

DistanceOp Op = **new** DistanceOp(geom, zonegeom);

//System.out.println("Op = " + Op);

//Lat/long coordinates of the nearest point of the zoneagent

Coordinate t1 = Op.nearestPoints()[1];

//lat/long coords of monarch agent

Coordinate t2 = Op.nearestPoints()[0];

//System.out.println("t1 = dest = " + t1);

//System.out.println("t2 = mon = " + t2);

//System.out.println("c1 = mon = " + c1);

destX.add(t1.x);

destY.add(t1.y);

//distance between the monarch, poly in lat/long units

**double** distl = Op.distance();

//System.out.println("distl = " + distl);

dists.add(distl);

//convert degrees to meters

CoordinateReferenceSystem crs = geography.getCRS();

**try** {

dist = JTS.*orthodromicDistance*(t1, t2, crs);

} **catch** (TransformException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

//System.out.println("dist = " + dist);

//get probEggs for zoneagent

**double** probEggs = zoneagent.getprobEggs();

//System.out.println("probEggs = " + probEggs);

//p decreases with increasing distance

**double** pdist = probEggs\*(1-(dist/400));

//double pdist = probEggs;

//System.out.println("pdist = " + pdist);

probs.add(pdist);

//total the p's for division later

runningsum = runningsum + pdist;

//System.out.println("runningsum = " + runningsum);

//LAY EGGS

**if**(distl == 0){

**double** r = Math.*random*();

//System.out.println("probEggs = " + probEggs);

//System.out.println("randno = " + r);

//System.out.println("dailyeggstolay1 = " + dailyeggstolay);

//System.out.println("dailyeggslaid1 = " + dailyeggslaid);

**if** (probEggs > r){

//need to account for the fact that sometimes don't have 4 eggs left to lay

**if**(dailyeggstolay > eggsperlay){

zoneagent.cumulativeeggs = zoneagent.cumulativeeggs + eggsperlay;

dailyeggslaid = dailyeggslaid + eggsperlay;

dailyeggstolay = dailyeggstolay - eggsperlay;

//System.out.println("dailyeggstolay2 = " + dailyeggstolay);

//System.out.println("dailyeggslaid2 = " + dailyeggslaid);

} **else**{

zoneagent.cumulativeeggs = zoneagent.cumulativeeggs + dailyeggstolay;

dailyeggslaid = dailyeggslaid + dailyeggstolay;

dailyeggstolay = 0;

//System.out.println("dailyeggstolay3 = " + dailyeggstolay);

//System.out.println("dailyeggslaid3 = " + dailyeggslaid);

}

numtimeslayeggs--;

//System.out.println("egglayingindex = " + numtimeslayeggs);

}

}

}

}

// Continue movement decisions

// if there is more than 1 zone (current zone monarch is in) to choose from

// choose which one to head towards

**if** (probs.size() > 1) {

//logic to determine index of target polygon

**int** whichPoly = -9999;

**double** r = Math.*random*();

//System.out.println("r = " + r);

**double** prevProb = 0;

//System.out.println("prevProb= " + prevProb);

**int** j = 0;

**boolean** done = **false**;

**while** (! done) {

**double** pnorm = (**double**)probs.get(j)/runningsum;

//System.out.println("probs\_j = " + probs.get(j));

//System.out.println("runningsum = " + runningsum);

//System.out.println("pnorm = " + pnorm);

**if** (prevProb < r && r < prevProb + pnorm) {

//System.out.println("prevProb= " + prevProb);

//System.out.println("prevProb+pnorm= " + prevProb+pnorm);

whichPoly = j;

done = **true**;

} **else** {

j++;

//System.out.println("j = " + j);

prevProb = prevProb + pnorm;

//System.out.println("prevProb + pnorm= " + prevProb);

}

}

//System.out.println("whichPolyFinal = " + whichPoly);

//System.out.println("size of probs array is " + probs.size());

//check to see if the target polygon is the polygon currently

//containing the monarch - hyp will be 0 if so

**double** hyp = (**double**)dists.get(whichPoly);

//System.out.println("hyp = " + hyp);

**if** (hyp > 0) {

//target poly is NOT the the poly currently containing the monarch

**double** PolyY = (**double**)destY.get(whichPoly)-c1.y;

**double** PolyX = (**double**)destX.get(whichPoly)-c1.x;

//System.out.println("Moving Towards X = " + destX.get(whichPoly) + " Y = " + destY.get(whichPoly));

//find angle to chosen Polygon

**double** geoRad = Math.*atan2*(PolyY, PolyX); //in radians

//System.out.println("Poly Y = " + PolyY);

//System.out.println("Poly X = " + PolyX);

**if** (geoRad < 0){

geoRad = geoRad + 2\*Math.***PI***;

}

//System.out.println("geo radian heading = " + geoRad);

geography.moveByVector(**this**, steplength, geoRad);

} **else** {

//moves in corr rand walk if it chooses polygon it is already in

corrrandwalk();

}

} **else** {

//moves in correlated random walk if it still has eggs or steps left

//and no other polygons in perception distance

corrrandwalk();

}

cumDist = cumDist + steplength;

} //end of choosing where to move when not random jumping

//check for doneness based on distance moved

**if** (cumDist >= maxdist) {

doneMove = **true**;

}

// done if out of eggs laid

**if** (numtimeslayeggs <= 0) {

doneMove = **true**;

}

//emergency fail-safe while loop halt condition

**if** (moveCount > 500) {

doneMove = **true**;

}

cumSteps++;

//System.out.println("steps2 " + cumSteps);

// record new coordinate - Dr. Parry

geom = geography.getGeometry(**this**);

c2 = geom.getCoordinates()[0];

//get distance moved

**double** moveDist = 0;

CoordinateReferenceSystem crs = geography.getCRS();

**try** {

moveDist = JTS.*orthodromicDistance*(c1, c2, crs);

} **catch** (TransformException e) {

//Auto-generated catch block

e.printStackTrace();

}

//double moveDist = Math.sqrt((c1.x - c2.x)\*(c1.x - c2.x) + (c1.y - c2.y)\*(c1.y - c2.y));

System.***out***.println("distance moved = " + moveDist);

//breaks links when monarch moves around torus

**if** (moveDist < 500) {

// Display path as network

displayNetwork(c1,c2);

}

}

}

}

**private** **void** corrrandwalk(){

//retrieve any Monarchs that go outside the boundaries and return them to opposite side

Context context = ContextUtils.*getContext*(**this**);

Geography<Monarch> geography = (Geography)context.getProjection("Monarchs");

Point loc = (Point)geography.getGeometry(**this**);

x = loc.getCoordinate().x;

y = loc.getCoordinate().y;

**if** (x < *xmin*){

Coordinate tempcoord1 = **new** Coordinate(*xmax* - (*xmin* - x), y);

Point temp1 = *fac*.createPoint(tempcoord1);

geography.move(**this**, temp1);

}

**if** (x > *xmax*){

Coordinate tempcoord2 = **new** Coordinate(*xmin* + (x - *xmax*), y);

Point temp2 = *fac*.createPoint(tempcoord2);

geography.move(**this**, temp2);

}

**if** (y < *ymin*){

Coordinate tempcoord3 = **new** Coordinate(x, *ymax* - (*ymin* - y));

Point temp3 = *fac*.createPoint(tempcoord3);

geography.move(**this**, temp3);

}

**if** (y > *ymax*){

Coordinate tempcoord4 = **new** Coordinate(x, *ymin* + (y - *ymax*));

Point temp4 = *fac*.createPoint(tempcoord4);

geography.move(**this**, temp4);

}

//get new angle for correlated random walk

angleChange = Math.***PI***\*RandomHelper.*nextDouble*()\*(1-directionality);

**if**(Math.*random*() < 0.5){

tempAngle1 = currAngle + angleChange;

**if**(tempAngle1 > 2\*Math.***PI***){

currAngle = tempAngle1 - 2\*Math.***PI***;

}

**else** {

currAngle = tempAngle1;

}

} **else** {

tempAngle2 = currAngle - angleChange;

**if**(tempAngle2 < 0){

currAngle = 2\*Math.***PI*** + tempAngle2;

}

**else**{

currAngle = tempAngle2;

}

}

//Monarch moves

geography.moveByVector(**this**, steplength, currAngle);

}

//Hazel's network code

**public** **void** displayNetwork(Coordinate c1,Coordinate c2){

Context context = ContextUtils.*getContext*(**this**);

Geography<Object> geography = (Geography)context.getProjection("Monarchs");

Network <Object> net = (Network <Object>) context.getProjection("travel");

net.addEdge(c1, c2);

// System.out.println("network " + net + "edge added from " + c1 + " to " + c2);

MonarchPath mp = **new** MonarchPath(net.getEdge(c1, c2),c1,c2);

context.add(mp);

Coordinate carray[] = **new** Coordinate[2];

carray[0] = c1;

carray[1] = c2;

GeometryFactory fac = **new** GeometryFactory();

LineString ls = fac.createLineString(carray);

geography.move(mp, ls);

}

**public** String getName() {

**return** name;

}

**public** **double** getEggsToLay(){

**return** dailyeggstolay;

}

**public** **double** getEggsLaid(){

**return** dailyeggslaid;

}

**public** **double** getTimesLaidEggs(){

**double** timeslaideggs = numtimeslayeggs2 - numtimeslayeggs;

**return** timeslaideggs;

}

**public** **double** getcumSteps(){

**return** cumSteps;

}

**public** **double** getMaxDist(){

**return** maxdist;

}

**public** **double** cumDist(){

**return** cumDist;

}

**public** String name() {

**return** name;

}

@Override

**public** String toString() {

**return** name;

}

}

ContextBuild.java

**public** **class** ContextBuild **implements** ContextBuilder{

**int** numAgents;

**public** Context build(Context context) {

System.***out***.println("Monarchs ContextBuild.build()");

Parameters parm = RunEnvironment.*getInstance*().getParameters();

numAgents = (Integer)parm.getValue("numAgents");

//set up GIS projection

GeographyParameters geoParams = **new** GeographyParameters();

Geography geography = GeographyFactoryFinder.*createGeographyFactory*(**null**).

createGeography("Monarchs", context, geoParams);

/\*\*\*\* Add Travel Network - Hazel's code\*/

Network network = NetworkFactoryFinder.*createNetworkFactory*(**null**)

.createNetwork("travel", context, **true**);

GeometryFactory fac = **new** GeometryFactory();

// FUDGE WORKAROUND FOR REPAST BUG TO VISUALISE NETWORK!:

//init coords are in a field in middle of test shapefile

Coordinate c = **new** Coordinate(-93.49, 42.065);

Coordinate c1 = **new** Coordinate(-93.490000001, 42.065);

Network <Object> net = (Network <Object>) context.getProjection("travel");

net.addEdge(c, c1);

Coordinate carray[] = **new** Coordinate[2];

carray[0]= **new** Coordinate(-93.49, 42.065);

carray[1]= **new** Coordinate(-93.490000001, 42.065);

LineString ls = fac.createLineString(carray);

MonarchPath mp = **new** MonarchPath(net.getEdge(c, c1),c,c1);

context.add(mp);

geography.move(mp, ls);

// Generate agents

**for** (**int** i = 0; i < numAgents; i++){

Monarch monarch = **new** Monarch("M" + i);//, Monarch.initeggstolay);

context.add(monarch);

Coordinate coord = **new** Coordinate(-93.50925 + 0.0515\* Math.*random*(),

42.04425 + 0.05215 \* Math.*random*());

Point geom = fac.createPoint(coord);

geography.move(monarch, geom);

}

//Load features from shapefiles

loadFeatures( "data/Testshapefile3.shp", context, geography);

**return** context;

}

**private** **void** loadFeatures (String filename, Context context, Geography geography){

URL url = **null**;

**try** {

url = **new** File(filename).~~toURL~~();

} **catch** (MalformedURLException e1) {

e1.printStackTrace();

}

List<SimpleFeature> features = **new** ArrayList<SimpleFeature>();

SimpleFeatureIterator fiter = **null**;

ShapefileDataStore store = **null**;

store = **new** ShapefileDataStore(url);

**try** {

fiter = store.getFeatureSource().getFeatures().features();

**while**(fiter.hasNext()){

features.add(fiter.next());

}

} **catch** (IOException e) {

e.printStackTrace();

}

**finally**{

fiter.close();

store.dispose();

}

// For each features in the file

**for** (SimpleFeature feature : features){

Geometry geom = (Geometry)feature.getDefaultGeometry();

Object agent = **null**;

// For Polygons, create ZoneAgents

**if** (geom **instanceof** MultiPolygon){

MultiPolygon mp = (MultiPolygon)feature.getDefaultGeometry();

geom = (Polygon)mp.getGeometryN(0);

// Read the feature attributes an assign to the ZoneAgent

String name = (String)feature.getAttribute("CLASS\_NAME");

//double pref = (double)feature.getAttribute("Pref");

**double** probEggs = (**double**)feature.getAttribute("ProbEggs");

**int** cumulativeeggs = 0;

agent = **new** ZoneAgent(name,cumulativeeggs,probEggs);

// I didn't create buffer like GIS example

// also didn't have else's for this if statement

}

**if** (agent != **null**){

context.add(agent);

geography.move(agent, geom);

}

**else**{

System.***out***.println("Error creating agent for " + geom);

}

}

}

}

Zoneagent.java

**public** **class** ZoneAgent {

**private** String name;

**public** **double** cumulativeeggs;

**private** **double** probEggs;

**public** ZoneAgent(String name, **int** cumulativeeggs, **double** probEggs){

**this**.name = name;

**this**.cumulativeeggs = cumulativeeggs;

**this**.probEggs = probEggs;

}

// @ScheduledMethod(start = 1, interval = 1)

// public void step() {

// collEggs();

// }

// private void collEggs(){

// Context context = ContextUtils.getContext(this);

// Geography geography = (Geography)context.getProjection("Monarchs");

//Checks if any monarchs in polygon and if so has probability to collect 4 eggs

//that contribute to total cumulative eggs in that polygon

//Parameters parm = RunEnvironment.getInstance().getParameters();

// GeographyWithin within = new GeographyWithin(geography, 0, this);

// for (Object obj : within.query()) {

// if (obj instanceof Monarch) {

// Monarch monarch = (Monarch)obj;

// if (this.probEggs > Math.random()){

//need to account for the fact that sometimes don't have 4 eggs left to lay

// if(monarch.dailyeggstolay > 3){

// cumulativeeggs = cumulativeeggs + Monarch.eggsperlay;

// } else{

// cumulativeeggs = cumulativeeggs + monarch.dailyeggstolay;

// }

// monarch.eggsleft();

// monarch.laideggs();

// }

// }

// }

// }

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **double** getEggs() {

**return** cumulativeeggs;

}

**public** **double** getprobEggs() {

**return** probEggs;

}

}

MonarchPath.java

**public** **class** MonarchPath {

// The NetworkEdge which represents this Path in the bat path network

**public** RepastEdge edge;

GeneralPath path;

**public** MonarchPath(RepastEdge e, Coordinate c1, Coordinate c2) {

**this**.edge = e;

setPath(c1,c2);

}

**public** **void** setPath(Coordinate c1, Coordinate c2){

//Create a general path which will go through all our vertices.

path = **new** GeneralPath();

//Move to the first vertex.

path.moveTo(c1.x, c1.y);

path.lineTo(c2.x,c2.y);

//Close the path.

path.closePath();

}

**public** GeneralPath getPath(){

**return** path;

}

/\*\*

\* Get the RepastEdge which represents this path object

\*

\* **@return** the edge

\*/

**public** RepastEdge getEdge() {

**return** edge;

}

/\*\*

\* **@param** edge

\* the edge to set

\*/

**public** **void** setEdge(RepastEdge edge) {

**this**.edge = edge;

}

}

MonarchStyle.java

**public** **class** MonarchStyle **implements** MarkStyle<Monarch>{

**private** Offset labelOffset;

**public** MonarchStyle(){

/\*\*

\* The gov.nasa.worldwind.render.Offset is used to position the label from

\* the mark point location. The first two arguments in the Offset

\* constructor are the x and y offset values. The third and fourth

\* arguments are the x and y units for the offset. AVKey.FRACTION

\* represents units of the image texture size, with 1.0 being one image

\* width/height. AVKey.PIXELS can be used to specify the offset in pixels.

\*/

labelOffset = **new** Offset(1.2d, 0.6d, AVKey.***FRACTION***, AVKey.***FRACTION***);

}

/\*\*

\* The PlaceMark is a WWJ PointPlacemark implementation with a different

\* texture handling mechanism. All other standard WWJ PointPlacemark

\* attributes can be changed here. PointPlacemark label attributes could be

\* set here, but are also available through the MarkStyle interface.

\*

\* **@see** gov.nasa.worldwind.render.PointPlacemark for more info.

\*/

@Override

**public** PlaceMark getPlaceMark(Monarch monarch, PlaceMark mark) {

// PlaceMark is null on first call.

**if** (mark == **null**)

mark = **new** PlaceMark();

/\*\*

\* The Altitude mode determines how the mark appears using the elevation.

\* WorldWind.ABSOLUTE places the mark at elevation relative to sea level

\* WorldWind.RELATIVE\_TO\_GROUND places the mark at elevation relative to ground elevation

\* WorldWind.CLAMP\_TO\_GROUND places the mark at ground elevation

\*/

mark.setAltitudeMode(WorldWind.***RELATIVE\_TO\_GROUND***);

mark.setLineEnabled(**false**);

**return** mark;

}

/\*\*

\* Get the mark elevation in meters. The elevation is used to visually offset

\* the mark from the surface and is not an inherent property of the agent's

\* location in the geography.

\*/

@Override

**public** **double** getElevation(Monarch monarch) {

**return** 0;

}

/\*\*

\* Here we set the appearance of the GisAgent. In this style implementation,

\* the style class creates a new BufferedImage each time getTexture is

\* called. If the texture never changes, the texture argument can just be

\* checked for null value, created once, and then just returned every time

\* thereafter. If there is a small set of possible values for the texture,

\* eg. blue circle, and yellow circle, those BufferedImages could

\* be stored here and re-used by returning the appropriate image based on

\* the agent properties.

\*/

@Override

**public** WWTexture getTexture(Monarch monarch, WWTexture texture) {

// WWTexture is null on first call.

Color color = **null**;

color = Color.***red***;

BufferedImage image = PatternFactory.*createPattern*(PatternFactory.***PATTERN\_CIRCLE***,

**new** Dimension(10, 10), 0.7f, color);

**return** **new** BufferedImageTexture(image);

}

/\*\*

\* Scale factor for the mark size.

\*/

@Override

**public** **double** getScale(Monarch monarch) {

**return** 1;

}

@Override

**public** **double** getHeading(Monarch monarch) {

**return** 0;

}

/\*\*

\* The agent on-screen label. Return null instead of empty string "" for better

\* performance.

\*/

@Override

**public** String getLabel(Monarch monarch) {

**return** "";

}

@Override

**public** Color getLabelColor(Monarch monarch) {

**return** Color.***ORANGE***;

}

/\*\*

\* Return an Offset that determines the label position relative to the mark

\* position. **@see** gov.nasa.worldwind.render.Offset

\*

\*/

@Override

**public** Offset getLabelOffset(Monarch monarch) {

**return** labelOffset;

}

@Override

**public** Font getLabelFont(Monarch obj) {

**return** **null**;

}

/\*\* Width of the line that connects an elevated mark with the surface. Use

\* a value of 0 to disable line drawing.

\*

\*/

@Override

**public** **double** getLineWidth(Monarch monarch) {

**return** 0;

}

@Override

**public** Material getLineMaterial(Monarch obj, Material lineMaterial) {

**if** (lineMaterial == **null**){

lineMaterial = **new** Material(Color.***RED***);

}

**return** lineMaterial;

}

@Override

**public** Offset getIconOffset(Monarch obj) {

// **TODO** Auto-generated method stub

**return** **null**;

}

}

PathStyle.java

**public** **class** PathStyle **implements** SurfaceShapeStyle<MonarchPath>{

@Override

**public** SurfaceShape getSurfaceShape(MonarchPath object, SurfaceShape shape) {

**return** **new** SurfacePolygon();

}

@Override

**public** Color getFillColor(MonarchPath zone) {

**return** Color.***CYAN***;

}

@Override

**public** **double** getFillOpacity(MonarchPath obj) {

**return** 0.25;

}

@Override

**public** Color getLineColor(MonarchPath zone){

**return** Color.***yellow***;

}

@Override

**public** **double** getLineOpacity(MonarchPath obj) {

**return** 1.0;

}

@Override

**public** **double** getLineWidth(MonarchPath obj) {

**return** 3;

}

}

**1-5-16**

Before cleaning code

**public** **class** Monarch {

**private** String name;

//Variables that can be changed

//distance monarch moves in each step

**private** **int** steplength = 50;

//directionality parameter constrains possible changes in direction

**private** **double** directionality = 0.75;

//perception distance in meters

**private** **double** perception = 100;

//number of eggs laid per step

**private** **double** eggsperlay = 4;

//probability a monarch jumps any given step

**private** **double** jumpProb = 0.0;

//jump distance

**private** **double** jumpLength = 300;

//parameters for distribution of random jump headings

//sample from normal dist with mean NNE (22.5 deg = 0.3927 rads)

//which in math degrees is 67.5 deg = 1.1781 rads

**private** **double** mean = 1.1781;

**private** **double** sigma = 0.5;

//number of polygons to remember visiting

**private** **int** remembered = 10;

//multiplier for polygons remembered

**private** **double** memscale = 0.5;

//equations for daily movement, daily eggs, distance\*probEggs,

//and area\*probEggs are below

//current angle in radians that Monarch is moving - initialize to random value

**private** **double** currAngle = 2\*Math.***PI***\*Math.*random*();

//magnitude of change in direction

**private** **double** angleChange;

//temp variable needed when angle crosses 360/0 degrees

**private** **double** tempAngle1;

**private** **double** tempAngle2;

//number of steps taken each day

**private** **double** cumSteps = 0;

//the number of times a monarch can lay eggs in a day

**private** **int** numtimeslayeggs;

**private** **double** numtimeslayeggs2;

//number of eggs that can be laid on a given day

**private** **double** dailyeggstolay = 0;

//number of eggs laid per day

**private** **double** dailyeggslaid = 0;

//dist in m between monarch and nearest point of zoneagent, calc'd from lat/long units

**private** **double** dist;

//cumulative distance and maximum dist can move

**private** **double** cumDist = 0;

**private** **double** maxdist = 0;

//number of steps taken to lay all eggs

**private** **double** eggslaidsteps = 0;

//current coordinates of Monarch agent

**private** **double** x;

**private** **double** y;

//xy boundaries of the geographic area in lat/long

**static** **double** *xmin* = -93.50925;

**static** **double** *xmax* = -93.45775;

**static** **double** *ymin* = 42.04425;

**static** **double** *ymax* = 42.0964;

//get context and geography

**public** **static** GeometryFactory *fac* = **new** GeometryFactory();

CoordinateReferenceSystem equalAreaCRS;

MathTransform transform;

Geometry transformedIntersection;

**public** Monarch(String name) {

**this**.name = name;

}

//Each step/tick is a day in the life of an egg-laying monarch

@ScheduledMethod(start = 1, interval = 1, priority = ScheduleParameters.***FIRST\_PRIORITY***)

**public** **void** step(){

//System.out.println("Monarch " + this.name());

//initialize these variables to 0 each day

cumDist = 0;

cumSteps = 0;

dailyeggslaid = 0;

eggslaidsteps = 0;

//Array to hold polygons previously visited

String[] memories = **new** String[remembered];

//gets current tick

**double** tick = RepastEssentials.*GetTickCount*();

//calculates max distance that can moved during current tick

maxdist = -500\*tick + 10500;

//maxdist for debugging

//maxdist = 1000;

//potential eggs to lay each day - currently starts at 50 and drops to 30

dailyeggstolay = -2\*tick + 52;

//System.out.println("Noeggstolayperday "+ dailyeggstolay);

//number of times a monarch can lay eggs each day if it lays 4 eggs each time

**double** numtimeslayeggs1 = dailyeggstolay/eggsperlay;

//System.out.println("NotimestolayperdayD "+ numtimeslayeggs1);

//number of times a monarch can lay eggs rounded up to nearest integer

numtimeslayeggs2 = Math.*ceil*(numtimeslayeggs1);

//System.out.println("NotimestolayperdayCeil "+ numtimeslayeggs2);

numtimeslayeggs = (**int**) numtimeslayeggs2;

//System.out.println("NotimestolayeggsFinal "+ numtimeslayeggs);

//boolean to determine when while loop ends

**boolean** doneMove = **false**;

//failsafe counter to stop while loop

**int** moveCount = 0;

**while** (! doneMove) {

//if (numtimeslayeggs >= 0) { //now keep moving even if run out of eggs

moveCount++; //increment emergency shutdown timer

//System.out.println("steps2 " + cumSteps);

Context context = ContextUtils.*getContext*(**this**);

Geography<Object> geography = (Geography)context.getProjection("Monarchs");

//first coord for network display - Dr. Parry code

Geometry geom = geography.getGeometry(**this**);

Coordinate c1= geom.getCoordinates()[0];

//System.out.println("c1 = mon = " + c1);

Coordinate c2 = **null**;

//step number when eggs run out

**if** (numtimeslayeggs > 0) {

eggslaidsteps++;

}

//decide whether to make random jump or not

**double** toJump = Math.*random*();

//double jumpHeading = 2\*Math.PI\*Math.random();

**double** jumpHeading;

**do** {

jumpHeading = Normal.*staticNextDouble*(mean, sigma);

//System.out.println("first first " + jumpHeadingtest);

} **while** ((jumpHeading > mean + 3.13) && (jumpHeading < mean - 3.13));

//System.out.println("first " + jumpHeadingtest);

**if** (jumpHeading < 0){

jumpHeading = jumpHeading + 2\*3.13;

}

**if** (jumpHeading > 2\*3.13){

jumpHeading = 2\*3.12;

}

//System.out.println("second " + jumpHeading);

//Jump

**if** (toJump < jumpProb) {

cumDist = cumDist + jumpLength;

geography.moveByVector(**this**, jumpLength, jumpHeading);

//Didn't jump

} **else** { //if not, get polygons, decide which to move towards

ArrayList probs = **new** ArrayList(); //normalized pref/p values

ArrayList dists = **new** ArrayList(); //distances in lat/long units

ArrayList destX = **new** ArrayList(); //array for latitude coord of destinations

ArrayList destY = **new** ArrayList();

ArrayList areaPoly = **new** ArrayList(); //area of polygons

**double** runningsum = 0; //sum of fprobs to normalize p's

//retrieve objects within perception distance

GeographyWithin within = **new** GeographyWithin(geography, perception, **this**);

//loops through objects within perception distance

**for** (Object obj : within.query()) {

//if an object is a ZoneAgent, then cast it as a ZoneAgent

**if** (obj **instanceof** ZoneAgent){

ZoneAgent zoneagent = (ZoneAgent)obj;

//System.out.println("ZoneAgenthexID = " + Integer.toHexString(System.identityHashCode(zoneagent)));

//System.out.println("ZoneAgentID = " + zoneagent.getID());

//System.out.println("ZoneAgentType = " + zoneagent.getName());

//creates a polygon geometry of this zone

Polygon zonegeom = (Polygon)geography.getGeometry(zoneagent);

//gets the lat/long of closest point in polygon, stores in array

//DistanceOp Op = new DistanceOp(geom, zonegeom, perception);

DistanceOp Op = **new** DistanceOp(geom, zonegeom);

//System.out.println("Op = " + Op);

//Lat/long coordinates of the nearest point of the zoneagent

Coordinate t1 = Op.nearestPoints()[1];

//lat/long coords of monarch agent

Coordinate t2 = Op.nearestPoints()[0];

//System.out.println("t1 = dest = " + t1);

//System.out.println("t2 = mon = " + t2);

//System.out.println("c1 = mon = " + c1);

destX.add(t1.x);

destY.add(t1.y);

//distance between the monarch, poly in lat/long units

**double** distl = Op.distance();

//System.out.println("distl = " + distl);

dists.add(distl);

//convert distance units from degrees to meters

CoordinateReferenceSystem crs = geography.getCRS();

**try** {

dist = JTS.*orthodromicDistance*(t1, t2, crs);

} **catch** (TransformException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

//System.out.println("dist = " + dist);

//get probEggs for zoneagent

**double** probEggs = zoneagent.getprobEggs();

//System.out.println("probEggs = " + probEggs);

//p decreases with increasing distance

//double pdist = probEggs\*(1-(dist/400));

**double** pdist = probEggs;

//System.out.println("pdist = " + pdist);

//p scaled by area

//Define an equal area CRS

**try** {

equalAreaCRS = CRS.*decode*("EPSG:2163", **true**);

} **catch** (NoSuchAuthorityCodeException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (FactoryException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

//create a buffer around the point of perception distance

Geometry pointBuffer = GeometryUtils.*generateBuffer*(geography, geom, perception);

//calculate area of intersection of buffer with zoneagent

Geometry intersection = pointBuffer.intersection(zonegeom);

//System.out.println("intersection = " + intersection);

//area of intersection in lat/long units

**double** degreeArea = intersection.getArea();

//System.out.println("degree area = " + degreeArea);

//transform from lat/long units to meters

**try** {

transform = CRS.*findMathTransform*(crs, equalAreaCRS, **true**);

} **catch** (FactoryException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

//Geometry transformedIntersection;

**try** {

transformedIntersection = JTS.*transform*(intersection, transform);

} **catch** (MismatchedDimensionException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (TransformException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**double** meterarea = transformedIntersection.getArea();

//System.out.println("meter area = " + meterarea);

//probmove declines linearly by area of buffer

//could save some space by doing this in lat/long units

// so don't need to convert area to meters above

//parea depends only on meterarea

**double** parea = meterarea/(Math.***PI***\*perception\*perception);

//parea depends on area and probEggs - see Excel spreadsheet for details

//double parea = probEggs + meterarea\*((1-probEggs)/(Math.PI\*perception\*perception));

//System.out.println("parea = " + parea);

//combined effect of distance and area

**double** pareadist = parea\*pdist;

//System.out.println("parea\*dist = " + pareadist);

//check if zoneagent is "remembered" and scale p accordingly

//according to some purists on stackflow, i might be better off

//writing a loop for this instead of coercing to List

**boolean** check = Arrays.*asList*(memories).contains(zoneagent.ID);

//System.out.println("check = " + check);

//final prob of moving to polygon after all adjustments

**double** fprob;

//if monarch has been there before, cut probmove in half

**if** (check == **true**){

//logistic equation to scale memory by area

//large areas have virtually no area effect

//I used lat/long area because that is what is in shapefile

**double** memareascale = 1/(1-Math.*exp*(-2\*(zoneagent.getArea()-1)));

//fprob = pareadist\*memscale;

fprob = pareadist\*memareascale;

//System.out.println("fprob1 = " + fprob);

}**else**{

fprob = pareadist;

//System.out.println("fprob2 = " + fprob);

}

probs.add(fprob);

//System.out.println("size of probs array is " + probs.size());

//make loop to print all probs for debugging

//total the p's for normalization below

runningsum = runningsum + fprob;

//System.out.println("runningsum = " + runningsum);

//for current polygon

**if**(distl == 0){

//lay eggs

//layeggs(); create this method

**double** r = Math.*random*();

//System.out.println("probEggs = " + probEggs);

//System.out.println("randno = " + r);

//System.out.println("dailyeggstolay1 = " + dailyeggstolay);

//System.out.println("dailyeggslaid1 = " + dailyeggslaid);

**if** (probEggs > r && dailyeggstolay > 0){

//need to account for the fact that sometimes don't have 4 eggs left to lay

**if**(dailyeggstolay > eggsperlay){

zoneagent.cumulativeeggs = zoneagent.cumulativeeggs + eggsperlay;

dailyeggslaid = dailyeggslaid + eggsperlay;

dailyeggstolay = dailyeggstolay - eggsperlay;

numtimeslayeggs--;

//System.out.println("egglayingindex = " + numtimeslayeggs);

//System.out.println("dailyeggstolay2 = " + dailyeggstolay);

//System.out.println("dailyeggslaid2 = " + dailyeggslaid);

} **else**{

zoneagent.cumulativeeggs = zoneagent.cumulativeeggs + dailyeggstolay;

dailyeggslaid = dailyeggslaid + dailyeggstolay;

dailyeggstolay = 0;

//System.out.println("dailyeggstolay3 = " + dailyeggstolay);

//System.out.println("dailyeggslaid3 = " + dailyeggslaid);

numtimeslayeggs--;

//System.out.println("egglayingindex = " + numtimeslayeggs);

}

}

//remember this polygon

**for** (**int** i = remembered-1; i > 0; i--){

memories[i] = memories[i-1];

}

memories[0] = zoneagent.ID;

//System.out.println("Memories[0]= " + memories[0]);

//System.out.println("Memories[1]= " + memories[1]);

//System.out.println("Memories[2]= " + memories[2]);

//System.out.println("Memories[3]= " + memories[3]);

//System.out.println("Memories[4]= " + memories[4]);

//System.out.println("Memories[5]= " + memories[5]);

//System.out.println("Memories[6]= " + memories[6]);

//System.out.println("Memories[7]= " + memories[7]);

//System.out.println("Memories[8]= " + memories[8]);

//System.out.println("Memories[9]= " + memories[9]);

}

}

}

// Continue movement decisions

// if there is more than 1 zone (current zone monarch is in) to choose from

// choose which one to head towards

**if** (probs.size() > 1) {

//logic to determine index of target polygon

**int** whichPoly = -9999;

**double** r = Math.*random*();

//System.out.println("r = " + r);

**double** prevProb = 0;

//System.out.println("prevProb= " + prevProb);

**int** j = 0;

**boolean** done2 = **false**;

**while** (! done2) {

**double** pnorm = (**double**)probs.get(j)/runningsum;

//System.out.println("probs\_j = " + probs.get(j));

//System.out.println("runningsum = " + runningsum);

//System.out.println("pnorm = " + pnorm);

**if** (prevProb < r && r < prevProb + pnorm) {

//System.out.println("prevProb= " + prevProb);

//the statement below is not printing the sum

//System.out.println("prevProb+pnorm= " + prevProb+pnorm);

whichPoly = j;

done2 = **true**;

} **else** {

j++;

//System.out.println("j = " + j);

prevProb = prevProb + pnorm;

//System.out.println("prevProb= " + prevProb);

}

}

//System.out.println("whichPolyFinal = " + whichPoly);

//System.out.println("size of probs array is " + probs.size());

//check to see if the target polygon is the polygon currently

//containing the monarch - hyp will be 0 if so

**double** hyp = (**double**)dists.get(whichPoly);

//System.out.println("hyp = " + hyp);

**if** (hyp > 0) {

//target poly is NOT the the poly currently containing the monarch

**double** PolyY = (**double**)destY.get(whichPoly)-c1.y;

**double** PolyX = (**double**)destX.get(whichPoly)-c1.x;

//System.out.println("Moving Towards X = " + destX.get(whichPoly) + " Y = " + destY.get(whichPoly));

//find angle to chosen Polygon

**double** geoRad = Math.*atan2*(PolyY, PolyX); //in radians

//System.out.println("Poly Y = " + PolyY);

//System.out.println("Poly X = " + PolyX);

**if** (geoRad < 0){

geoRad = geoRad + 2\*Math.***PI***;

}

//System.out.println("geo radian heading = " + geoRad);

geography.moveByVector(**this**, steplength, geoRad);

} **else** {

//moves in corr rand walk if it chooses polygon it is already in

corrrandwalk();

}

} **else** {

//moves in correlated random walk if it still has eggs or steps left

//and no other polygons in perception distance

corrrandwalk();

}

cumDist = cumDist + steplength;

} //end of choosing where to move when not random jumping

//check for doneness based on distance moved

**if** (cumDist >= maxdist) {

doneMove = **true**;

}

// done if out of eggs laid

//if (numtimeslayeggs <= 0) {

// doneMove = true;

//}

//emergency fail-safe while loop halt condition

**if** (moveCount > 500) {

doneMove = **true**;

}

cumSteps++;

//System.out.println("steps2 " + cumSteps);

/\*\*

// record new coordinate - Dr. Parry

geom = geography.getGeometry(this);

c2 = geom.getCoordinates()[0];

//get distance moved

double moveDist = 0;

CoordinateReferenceSystem crs = geography.getCRS();

try {

moveDist = JTS.orthodromicDistance(c1, c2, crs);

} catch (TransformException e) {

//Auto-generated catch block

e.printStackTrace();

}

//double moveDist = Math.sqrt((c1.x - c2.x)\*(c1.x - c2.x) + (c1.y - c2.y)\*(c1.y - c2.y));

//System.out.println("distance moved = " + moveDist);

//breaks links when monarch moves around torus

if (moveDist < 500) {

// Display path as network

displayNetwork(c1,c2);

}

\*\*/

//} //if statement for if eggs left to lay

}

}

**private** **void** layeggs(){

//fill this

}

**private** **void** corrrandwalk(){

//retrieve any Monarchs that go outside the boundaries and return them to opposite side

Context context = ContextUtils.*getContext*(**this**);

Geography<Monarch> geography = (Geography)context.getProjection("Monarchs");

Point loc = (Point)geography.getGeometry(**this**);

x = loc.getCoordinate().x;

y = loc.getCoordinate().y;

**if** (x < *xmin*){

Coordinate tempcoord1 = **new** Coordinate(*xmax* - (*xmin* - x), y);

Point temp1 = *fac*.createPoint(tempcoord1);

geography.move(**this**, temp1);

}

**if** (x > *xmax*){

Coordinate tempcoord2 = **new** Coordinate(*xmin* + (x - *xmax*), y);

Point temp2 = *fac*.createPoint(tempcoord2);

geography.move(**this**, temp2);

}

**if** (y < *ymin*){

Coordinate tempcoord3 = **new** Coordinate(x, *ymax* - (*ymin* - y));

Point temp3 = *fac*.createPoint(tempcoord3);

geography.move(**this**, temp3);

}

**if** (y > *ymax*){

Coordinate tempcoord4 = **new** Coordinate(x, *ymin* + (y - *ymax*));

Point temp4 = *fac*.createPoint(tempcoord4);

geography.move(**this**, temp4);

}

//get new angle for correlated random walk

angleChange = Math.***PI***\*RandomHelper.*nextDouble*()\*(1-directionality);

**if**(Math.*random*() < 0.5){

tempAngle1 = currAngle + angleChange;

**if**(tempAngle1 > 2\*Math.***PI***){

currAngle = tempAngle1 - 2\*Math.***PI***;

}

**else** {

currAngle = tempAngle1;

}

} **else** {

tempAngle2 = currAngle - angleChange;

**if**(tempAngle2 < 0){

currAngle = 2\*Math.***PI*** + tempAngle2;

}

**else**{

currAngle = tempAngle2;

}

}

//Monarch moves

geography.moveByVector(**this**, steplength, currAngle);

}

/\*\*

//Hazel's network code

public void displayNetwork(Coordinate c1,Coordinate c2){

Context context = ContextUtils.getContext(this);

Geography<Object> geography = (Geography)context.getProjection("Monarchs");

Network <Object> net = (Network <Object>) context.getProjection("travel");

net.addEdge(c1, c2);

// System.out.println("network " + net + "edge added from " + c1 + " to " + c2);

MonarchPath mp = new MonarchPath(net.getEdge(c1, c2),c1,c2);

context.add(mp);

Coordinate carray[] = new Coordinate[2];

carray[0] = c1;

carray[1] = c2;

GeometryFactory fac = new GeometryFactory();

LineString ls = fac.createLineString(carray);

geography.move(mp, ls);

}

\*\*/

**public** String getName() {

**return** name;

}

**public** **double** getEggsToLay(){

**return** dailyeggstolay;

}

**public** **double** getEggsLaid(){

**return** dailyeggslaid;

}

**public** **double** getTimesLaidEggs(){

**double** timeslaideggs = numtimeslayeggs2 - numtimeslayeggs;

**return** timeslaideggs;

}

**public** **double** getcumSteps(){

**return** cumSteps;

}

**public** **double** getMaxDist(){

**return** maxdist;

}

**public** **double** getcumDist(){

**return** cumDist;

}

**public** **double** geteggslaidsteps(){

**return** eggslaidsteps;

}

**public** String name() {

**return** name;

}

@Override

**public** String toString() {

**return** name;

}

}