```
CS4610: Robotic Science and Systems
Lab 5: Fiducial Tag Localization and EKF
Nicholas Jones, Zach Ferland
-----------------------------------

How to compile
--------------
hopefully the code will run without installing anything new

run: make

If this is not the case you may need to follow the installation instructions
on http://april.eecs.umich.edu/wiki/index.php/Download_and_Installation

How to Execute
--------------
Unfortunately we don't have a cohesive product.

to run the tag localization:
  ./run-class FiducialTagFinder

The tag used for testing was tag 0 of tag family tag36h11, which is available
here: http://april.eecs.umich.edu/software/tag36h11.tgz
The size of the tag will effect the accuracy of any 3d readings (were we making
them correctly.)  For quick demonstration purposes this shouldn't matter.

to run the EKF:
  ./run-class EKF

EKF can also be run in another class in a seperate thread as such
       EKF ekf = new EKF(ohmm);
       ekf.run();

Known Bugs/ Incomplete aspects
------------------------------
AprilTag Localization:
       As far as I can tell, the only pieces that prevent the localization from
       working perfectly are the two coordinate tranformations.  In
       FiducialTagFinder.getVisibleTagLocs(), the homography needs to be converted
  into the location of the tag in reference to the robot.  We were unable to
       get this to work.  In WorldMap.findRobotGlobal(), the location of the tag
       in the robot frame needs to be compared to its location in the world frame
       to find the pose of the robot.  We were unable to get this to work either.

  Other than these 6 lines of code, I'm pretty sure everything works.

EKF:
EKF runs quite well and accurate. It has to be tuned more properly though with more
accurate error estimates to run well for this specific robot. The current EKF class fakes
input (pose) from an extoreceptive sensor(camera) since we were not able
incorporate AprilTags properly and get accurate readings from it. There is a few minor
things noted in the code that could be improved and written more accurately as well. I would
like to improve it in the future and provide a visualization of the error ellipse.


Extra Credit
------------
None

Contributions
-------------
Nick wrote all the code in WorldMap, TagLocation, and FiducialTagFinder.

Zach wrote the code in EKF
```

Gentlemen - In general I like what you've done. The most critical failure is that you seem to have discovered that you needed help too late. Knowing approximately what needs to be done in code is one thing, but debugging it can be massively time consuming and can require outside help and discussion to truly understand the details. If everything was working and integrated this would be "A" quality work. I believe if you could have gotten to that point if you had left about 3 or 4 more days in your development schedule at the end, including time to consult with me in detail.

As it is I am assigning you a B (85/100).

```
/*******************************************************************************
* CS4610: Lab 5
* Group 1: Nicholas Jones, Zach Ferland
*******************************************************************************/
package l5;

import java.io.IOException;

import ohmm.*;
import ohmm.Grasp.*;
import Jama.*;
import april.tag.*;
import Jama.EigenvalueDecomposition;
import java.lang.Math;

public class EKF {
    //semantics, array access
    static final int X = 0;
    static final int Y = 1;
    static final int T = 2;
    static final int LEFT = 0;
    static final int RIGHT = 1;

    //robot baseline (mm)
    public static final double b = 205;
    //radius of robot wheels (mm)
    public static final double r = 39;

    // Robot Pose
    protected float[] pose = new float[3];
    // Wheel positions (rotations) (total)
    protected float[] motPos = new float [2];
    // covariance matrix of pose - initialized as a zero matrix
    Matrix covariancePose = new Matrix(3,3);

    // EKF needs an instance of OHMMDrive
    OHMMDrive ohmm = null;

    //initialize EKF
    public EKF(OHMMDrive ohmm) {
        this.ohmm = ohmm;
    }

    // Main run function when used in other objects or classes, run in a seperate thread
    public void run() {
        long taskTime = 0;
        long sleepTime = 1000/1;
        while (true) {
            taskTime = System.currentTimeMillis();

            predictionProcess();
            //errorEllipse();
            // poseUncertainty(1);

            // if the area of the error elipse gets too large from odometry, run a update process
            double areaUncert = 600;
            if (errorSize() > areaUncert) {
                updateProcess();
            }

            taskTime = System.currentTimeMillis()-taskTime;
            if (sleepTime-taskTime > 0 ) {
            try {
                    Thread.sleep(sleepTime-taskTime);
            } catch(InterruptedException e) { }
          }
        }
    }

    // runs at (times per second) determines odometry with uncertainty
    public void predictionProcess() {
        System.out.println("////////////////////////////////////////////////////////////////////
/");

        //get current robot pose - x y t
        float[] newPose = new float[3];
        ohmm.driveGetPose(newPose);
```

```
        System.out.println("NEW POSE = X - " + newPose[X] + " Y - " + newPose[T] + " T - " + newPo
se[T]);

        //get wheel rotations since last update - 1 = left, 2 = right
        float[] newMotPos = new float[2];
        ohmm.motGetPos(newMotPos);

        //calculate difference since last time
        float motPosLDiff = newMotPos[LEFT] - motPos[LEFT];
        float motPosRDiff = newMotPos[RIGHT] - motPos[RIGHT];
        float wl = motPosLDiff;
        float wr = motPosRDiff;

        //gurantee difference is not zero, zero difference causes division/multiplications issues
        //when calculating pose covariance
        if (wl == wr) {
             wr = wr + 0.00000001f;
        }

        //Create diagnol covariance matrix for wheels
        Matrix covarianceWheels = new Matrix(2,2);

        //set standard deviation of wheels (will have to determine this with tests)
        float sdwheel = 0.01f;

        //set proportional standard deviation or fixed constant as diagonal matrix
        covarianceWheels.set(0,0,sdwheel * wl);
        covarianceWheels.set(1,1,sdwheel * wr);

        // Print Pose Covariance
        System.out.println("Pose Covariance");
        covariancePose.print(2,2);

        //Initialize 2D Jacobian Array of Pose ()
        double[][] jacobianPoseArray = new double[3][3];

        //calculate jacobian array of partial derivatives of F over p
        jacobianPoseArray[0][0] = 1;
        jacobianPoseArray[0][1] = 0;
        jacobianPoseArray[0][2] = ((b*(wr+wl))/(2*(wr-wl)))*(-Math.cos(pose[T])+Math.cos(pose[T]+(
r/b)*(wr-wl)));
        jacobianPoseArray[1][0] = 0;
        jacobianPoseArray[1][1] = 1;
        jacobianPoseArray[1][2] = ((b*(wr+wl))/(2*(wr-wl)))*(-Math.sin(pose[T])+Math.sin(pose[T]+(
r/b)*(wr-wl)));
        jacobianPoseArray[2][0] = 0;
        jacobianPoseArray[2][1] = 0;
        jacobianPoseArray[2][2] = 1;

        //create matrix from 2D array
        Matrix jacobianPose = new Matrix(jacobianPoseArray);

        // System.out.println("Jacobian Pose");
        // jacobianPose.print(3,2);

        //transpose of jacobian pose
        Matrix jacobianPoseT = jacobianPose.transpose();

       // System.out.println("Jacobian Pose Transpose");
       // jacobianPoseT.print(3,2);

       //Initialize 2D Jacobian Array of wheels ()
       double[][] jacobianWheelArray = new double[3][2];

        //calculate jacobian array of partial derivatives of F over w
        jacobianWheelArray[0][0] = ((b/2)*((wr+wl)/Math.pow(wr-wl, 2))*(-Math.sin(pose[T])+Math.si
n(pose[T]+(r/b)*(wr-wl)))) + ((b/2)*(1/(wr-wl))*(-Math.sin(pose[T])+Math.sin(pose[T]+(r/b)*(wr-wl)
))) - ((r/2)*((wr+wl)/(wr-wl))*(Math.cos(pose[T]+(r/b)*(wr-wl))));
        jacobianWheelArray[0][1] = ((-b/2)*((wr+wl)/Math.pow(wr-wl, 2))*(-Math.sin(pose[T])+Math.s
in(pose[T]+(r/b)*(wr-wl)))) + ((b/2)*(1/(wr-wl))*(-Math.sin(pose[T])+Math.sin(pose[T]+(r/b)*(wr-wl
)))) + ((r/2)*((wr+wl)/(wr-wl))*(Math.cos(pose[T]+(r/b)*(wr-wl))));
        jacobianWheelArray[1][0] = ((r/2)*((wr+wl)/(wr-wl))*(Math.sin(pose[T]+(r/b)*(wr-wl))))-((b
/2)*((wr+wl)/Math.pow(wr-wl, 2))*(Math.cos(pose[T])-Math.cos(pose[T]+(r/b)*(wr-wl)))) + ((b/2)*(1/
(wr-wl))*(Math.cos(pose[T])-Math.cos(pose[T]+(r/b)*(wr-wl))));
        jacobianWheelArray[1][1] = ((-r/2)*((wr+wl)/(wr-wl))*(Math.sin(pose[T]+(r/b)*(wr-wl)))) +
((b/2)*((wr+wl)/Math.pow(wr-wl, 2))*(Math.cos(pose[T])-Math.cos(pose[T]+(r/b)*(wr-wl)))) + ((b/2)*
(1/(wr-wl))*(Math.cos(pose[T])-Math.cos(pose[T]+(r/b)*(wr-wl))));
        jacobianWheelArray[2][0] = -r/b;
```

```
        jacobianWheelArray[2][0] = r/b;

        //create matrix from array
        Matrix jacobianWheel = new Matrix(jacobianWheelArray);

        //  System.out.println("Jacobian Wheels");
        // jacobianWheel.print(2,2);

        //transpose of jacobian wheel
        Matrix jacobianWheelT = jacobianWheel.transpose();

        // System.out.println("Jacobian Wheels Transpose");
        //  jacobianWheelT.print(3,2);

        //Calculate new covariance matrix of pose
        // newsigmaP = (Jp * sigmaP * Jpt) + (Jw * sigmaW * Jwt)
        Matrix newCovariancePose = new Matrix(3,3);

        //pose prediction error (Jp * sigmaP * Jpt)
        Matrix tempP = jacobianPose.times(covariancePose);
        Matrix errorPose = tempP.times(jacobianPoseT);

        //wheel error
        Matrix tempW = jacobianWheel.times(covarianceWheels);
        Matrix errorWheel = tempW.times(jacobianWheelT);

        newCovariancePose = errorPose.plus(errorWheel);

        //Update values with new calculations
        pose = newPose;
        motPos = newMotPos;
        covariancePose = newCovariancePose;
    }

    public void updateProcess() {
        // System.out.println("Covariance Before");
        // covariancePose.print(3,2);

        System.out.println("UPDATE PROCESS ----------------------------------------");

        // Create diagonal uncertainty matrix from exteroceptive sensing (q)
        Matrix covarianceVisQ = new Matrix(3,3);
        // accuracy here is unknown and unmeasured, used below is a best guess
        // 10 mm
        double sdpose = 2;
        double sdtheta = 0;
        //set  standard deviation diagonal matrix
        covarianceVisQ.set(0,0,sdpose);
        covarianceVisQ.set(1,1,sdpose);
        covarianceVisQ.set(2,2,sdtheta);

        //get visual sensing pose
        float[] poseQ = new float[3];
        // SHOULD GET POSE FROM EXTROCEPTIVE SENSING CLASS HERE
        // FOR THE TIME BEING THIS IS FAKED
        // MarkerDetection.getPose(poseQ);
         ohmm.drivePause();
        // messy fake pause
        try {
            Thread.sleep(3000);

        } catch (InterruptedException ie) {

        }

        ohmm.driveUnPause();

        // Set given pose of extroceptive sensors (this is fake for the time being as stated above
)
        Matrix mPoseQ = new Matrix(3,3);
        mPoseQ.set(0,0, pose[X] + 1);
        mPoseQ.set(1,1, pose[Y] - 1);
        mPoseQ.set(2,2, pose[T]);

        //the addition of covariance from vis sensing and odometry
        Matrix covarianceV =  covarianceVisQ.plus(covariancePose);

        // System.out.println("COVARIANCE V");
```

```
// covarianceV.print(3,2);

//instantiate corrected pose matrix 3 X 3 diagnol
Matrix newPose = new Matrix(3,3);

//inverse of matrix covarianceV
Matrix covarianceVI = covarianceV.inverse();

//kalman gain
Matrix kgain = covariancePose.times(covarianceVI);

// System.out.println("KGAIN");
// kgain.print(3,2);

// covert current pose to matrix - complete explicitly since it is not a double array
// 3x3 diagonal matrix
Matrix poseM = new Matrix(3, 3);
poseM.set(0,0,pose[X]);
poseM.set(1,1,pose[Y]);
poseM.set(2,2,pose[T]);

// v = q - p
Matrix poseV = mPoseQ.minus(poseM);

// System.out.println("POSE V");
// poseV.print(3,2);

//update pose by weighted uncertainty from both vis and odometry
// p1 = p + Kv
newPose = poseM.plus(kgain.times(poseV));

// System.out.println("NEW POSE");
// newPose.print(3,2);

//update pose covariance, decrease uncertainty
Matrix newCovariancePose = new Matrix(3,3);

//tranpose kalman gain
 Matrix kgainT = kgain.transpose();

// System.out.println("KGAIN TRANSPOSE");
// kgainT.print(3,2);

 //ksigma * Vsigma * ksigmaT
 Matrix temp = kgainT.times(kgain.times(covarianceV));

// System.out.println("MINUS COVARIANCE");
// temp.print(3,2);

newCovariancePose = covariancePose.minus(temp);

// System.out.println("NEW POSE COVARIANCE");
// newCovariancePose.print(3,2);

 //update pose and covariance
 pose[X] = (float) newPose.get(0,0);
 pose[Y] = (float) newPose.get(1,1);
 pose[T] = (float) newPose.get(2,2);
  System.out.println("NEW POSE = X - " + pose[X] + " Y - " + pose[T] + " T - " + pose[T]);

 // SHOULD ONLY THE DIAGNOLS BE PASSED ALONG?
 // had many issues here with the covariance growing too large too fast, after this
//calculation, and negatives. Also turning error grew to large, set fixed for now
 newCovariancePose.set(0,1,0);
 newCovariancePose.set(0,2,0);
 newCovariancePose.set(1,0,0);
 newCovariancePose.set(1,2,0);
 newCovariancePose.set(2,0,0);
 newCovariancePose.set(2,1,0);
 // newCovariancePose.set(2,2,Math.abs(newCovariancePose.get(2,2)));
 newCovariancePose.set(2,2,0.01);

 covariancePose = newCovariancePose;
 //update pose when sensing data is more accurate
 // ohmm.driveSetPose(pose[X], pose[Y], pose[T])

System.out.println("New Pose Covariance");
covariancePose.print(3,2);
```

```
        System.out.println("END UPDATE PROCESS ----------------------------------------");
    }

    //get error ellipse (returns radi r0 and r1 in array)
    //TODO - add functionality to return all need information to draw ellipse
    public double[] errorEllipse(){
        //o (will need these to draw ellipse, the xy of pose)
        // maybe return these after
        float x = pose[X];
        float y = pose[Y];

        //get eigenValues
        double[] eVal = getEigenVal();

        //desired uncertainty
        double p = 95;

        double r0 = Math.sqrt(-2*Math.log(1-p/100)*eVal[0]);
        double r1 = Math.sqrt(-2*Math.log(1-p/100)*eVal[1]);

        double[] radi = new double[2];
        radi[0] = r0;
        radi[1] = r1;

        double area = radi[0] * radi[1] * Math.PI;

        System.out.println("----------------------------------------");
        System.out.println("ERROR ELLIPSE AREA - " + area + " R1 = " + radi[0] + " R2 = " + radi[1
]);
        System.out.println("----------------------------------------");

        return radi;
    }

    //area of current error ellipse, and singe value that can be used to compare errors
    public double errorSize() {
        double[] ellipseRadi = errorEllipse();
        double area = ellipseRadi[0] * ellipseRadi[1] * Math.PI;
        return area;
    }

    //get pose uncertainty based on a fixed ellipse area
    // does not work well yet
    public double poseUncertainty(double area){
        //get eigenValues
        double[] eVal = getEigenVal();

        double raised = Math.sqrt(Math.pow((area/Math.PI),2)/(4*eVal[0]*eVal[1]));
        double p = 100 * (Math.pow(Math.E, raised) - 1);

        //1 or 100?
        if (p > 100) {
            p = 100;
        }

        System.out.println("Uncertainty Percent - " + p);
        return p;
    }

    //returns two eigen values in double array based on current pose
    public double[] getEigenVal(){
        //take upper left 2x2 submatrix from the 3x3 pose covariance matrix
        Matrix e = covariancePose.getMatrix(0,1,0,1);

        //compute an Eigendecomposition of E
        EigenvalueDecomposition eigenDecomp = new EigenvalueDecomposition(e);

        // diagonal matrix, two values
        Matrix eigenValues = eigenDecomp.getD();

        //EigenValues l0 and l1
        double l0 = eigenValues.get(0,0);
        double l1 = eigenValues.get(1,1);

        //create return array
        double[] eVal = new double[2];
        eVal[0] = l0;
```

```
        eVal[1] = l1;

        return eVal;
    }

    //Main run class for EKF if not used with another class or object, simple a demonstration
    //of EKF while the robot continously drives in a straight line
    public static void main(final String[] args) throws IOException, InterruptedException {
       OHMMDrive ohmm =
                (OHMMDrive) OHMM.makeOHMM(new String[]{"-r", "/dev/ttyACM1"});
        if (ohmm == null) System.exit(0);

        // reset pose
        ohmm.motReinit();

        EKF ekf = new EKF(ohmm);

        ohmm.driveResetPose();

        ohmm.driveSetVW(40f, 0.0f);

        //run n times per second (note - handle this better)
        long n = 1;
        long taskTime = 0;
        long sleepTime = 1000/n;
        while (true) {
            taskTime = System.currentTimeMillis();

            ekf.predictionProcess();
            // ekf.errorEllipse();
            ekf.poseUncertainty(100);

            // if error ellipse area get too large, run an update process
            double areaUncert = 600;
            if (ekf.errorSize() > areaUncert) {
                ekf.updateProcess();
            }

          taskTime = System.currentTimeMillis()-taskTime;
          if (sleepTime-taskTime > 0 ) {
            Thread.sleep(sleepTime-taskTime);
          }
        }

        //ohmm.close();
    }
}
```

```
/*****************************************************************************
*
* TagLocation
* CS4610: Robotics Science and Systems
* Lab 5
* 4/14/2014
* Nicholas Jones, Zach Ferland
*
*****************************************************************************/

package l5;

import ohmm.*;

import com.googlecode.javacpp.*;
import com.googlecode.javacv.*;
import com.googlecode.javacv.cpp.*;
import static com.googlecode.javacv.cpp.opencv_calib3d.*;
import static com.googlecode.javacv.cpp.opencv_contrib.*;
import static com.googlecode.javacv.cpp.opencv_core.*;
import static com.googlecode.javacv.cpp.opencv_features2d.*;
import static com.googlecode.javacv.cpp.opencv_flann.*;
import static com.googlecode.javacv.cpp.opencv_highgui.*;
import static com.googlecode.javacv.cpp.opencv_imgproc.*;
import static com.googlecode.javacv.cpp.opencv_legacy.*;
import static com.googlecode.javacv.cpp.opencv_ml.*;
import static com.googlecode.javacv.cpp.opencv_objdetect.*;
import static com.googlecode.javacv.cpp.opencv_video.*;

import java.awt.image.BufferedImage;
import java.awt.Graphics2D;
import java.awt.Color;

import java.io.IOException;

import april.tag.TagDetector;
import april.tag.TagDetection;
import april.tag.CameraUtil;
import april.tag.Tag36h11;
import april.tag.TagFamily;

import april.jmat.LinAlg;

import java.util.ArrayList;
import java.util.Arrays;

/**
* FiducialTagFinder runs an image server that serves the images from the video
* camera.  It also takes the images from the camera and processes them with
* AprilTags to find the 3D location of the tags in the picure.  This information
* can be used to obtain the location of the robot in the world frame.
* @author Nicholas Jones
* @version 4/14/2014
*/
public class FiducialTagFinder extends CvBase implements Runnable {
    // CAMERA CONSTANTS
    public static final double[] CAMERA_F
        = new double[] {534.816842, 531.883274}; // Camera focal length

    public static final double[] CAMERA_C
        = new double[] {304.638279, 224.403908}; // Camera Center

    public static final double[] CAMERA_K
        = new double[] {0.0403554, -0.022707, -0.014288, -0.263265}; // Camera K

    public static final double TAG_SIZE = 0.125; // Size of tag in meters

    // IMAGE SERVER FIELDS
    public static final int DEF_MAX_FPS = 5; // Max frames per second
    public static final String APPNAME = "TagFinder"; // Name of APP


    private ArrayList<TagDetection> currentlySeenTags;
    private TagDetector detector;
    private WorldMap map;
    protected IplImage procImg = null;
```

```
/**
* Constructor
* @param family - The TagFamily which all tags seen will belong to.
* @param map - The path of the file containing the map of the tags in the
*              world frame.
* @throws IOException
*/
public FiducialTagFinder(TagFamily family, String map) throws IOException {
    super(APPNAME);
    maxFPS = DEF_MAX_FPS;

    this.map = new WorldMap(map);

    System.out.println(this.map);

    this.currentlySeenTags = new ArrayList<TagDetection>();
    this.detector =  new TagDetector(family);
}


/**
* Start the thread.
*/
public void run() {
    mainLoop();
    release();
}

/**
* Process the frame from the camera to find the TagDetections.
* @param frame - The image from the camera
* @return The image from the camera unchanged
*/
public IplImage process(IplImage frame) {

    if (frame != null)
        this.currentlySeenTags = this.detector.process(frame.getBufferedImage(), CAMERA_C);

    return frame;
}

/**
* Releases allocated memory.
*/
public void release() {
    if (procImg != null) { procImg.release(); procImg = null; }
        super.release();
}


/**
* Get the locations of the visible tags.
* @return A list of TagLocations containing the 3D location of the tags in
*         the robot frame.
*/
private ArrayList<TagLocation> getVisibleTagLocs() {
    ArrayList<TagLocation> list = new ArrayList<TagLocation>();

    for (TagDetection td : this.currentlySeenTags) {
        double m[][] = CameraUtil.homographyToPose(CAMERA_F[0],
                                                   CAMERA_F[1],
                                                   CAMERA_C[0],
                                                   CAMERA_C[1],
                                                   td.homography);
        m = CameraUtil.scalePose(m, 2.0, TAG_SIZE);

        for (int i = 0; i < m.length; i++) {
            for (int j = 0; j < m[0].length; j++)
                System.out.printf("%8.3f ", m[i][j]);
            System.out.println();
        }

        //System.out.println();
        list.add(new TagLocation(td.id, m[0][3], m[1][3], m[2][3], 0.0 /* TODO */));
    }

    System.out.println("------------------------");
```

```
        return list;
    }

    /**
    * Find the robot location in the world frame using the tags visible together
    * with the world map.
    * @return {x, y, z, rotation} of the robot in meters and radians
    */
    public double[] getRobotLocation() {
        ArrayList<TagLocation> tags = getVisibleTagLocs();

        if (tags.size() == 0)
            return null;
        else
            return this.map.findRobotGlobal(tags.get(0));
    }


    /**
    * ENTRY POINT
    */
    public static void main(String[] args) throws IOException {
        FiducialTagFinder finder = new FiducialTagFinder(new Tag36h11(), "tagmap.map");
        finder.init(args.length, args);

        new Thread(finder).start();

        for (int i = 0; i < 6000; i++) {
            double[] robot = finder.getRobotLocation();
            if (robot != null)
                System.out.println(Arrays.toString(robot));
            try {Thread.sleep(1000);}
            catch (InterruptedException e) {}
        }
    }
}
```

```
package l5;

import ohmm.*;
import java.io.IOException;

public class Stop {
  public static void main(String[] args) throws IOException, InterruptedException {
        OHMMDrive ohmm =
          (OHMMDrive) OHMM.makeOHMM(new String[]{"-r", "/dev/ttyACM1"});
        if (ohmm == null) System.exit(0);

    ohmm.driveSetVW(0f, 0f);

        ohmm.close();
  }
}
```

```
/*******************************************************************************
*
* TagLocation
* CS4610: Robotics Science and Systems
* Lab 5
* 4/14/2014
* Nicholas Jones, Zach Ferland
*
*******************************************************************************/

package l5;

/**
* TagLocation is a class to contain information about the 3D location and
* rotation (around the axis orthogonal to the face of the tag.)
* @author Nicholas Jones
* @version 4/14/2014
*/
public class TagLocation {
    /** The id of the tag */
    public int id;

    /** The x coordinate of the tag in meters. */
    public double x;

    /** The y coordinate of the tag in meters. */
    public double y;

    /** The z coordinate of the tag in meters. */
    public double z;

    /** The rotation of the tag around the axis orthogonal to its face in rad */
    public double rotation;

    /**
    * Constructor
    * @param id - The id of the tag
    * @param x - The x coordinate of the tag in meters
    * @param y - The y coordinate of the tag in meters
    * @param z - The z coorfinate of the tag in meters
    * @param rotation - The rotation of the tag around the axis orthogonal to
    *                   its face in rad
    */
    public TagLocation(int id, double x, double y, double z, double rotation) {
        this.id = id;
        this.x = x;
        this.y = y;
        this.z = z;
        this.rotation = rotation;
    }


    /**
    * @return A String representing the TagLocation
    */
    @Override
    public String toString() {
        return  "id: "   + this.id +
                " x: "   + this.x  +
                " y: "   + this.y  +
                " z: "   + this.z  +
                " rot: " + this.rotation;
    }
}
```

```
/*******************************************************************************
*
* WorldMap
* CS4610: Robotics Science and Systems
* Lab 5
* 4/14/2014
* Nicholas Jones, Zach Ferland
*
*******************************************************************************/

package l5;

import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;

/**
* WorldMaps reads a file containing a list of tag locations and provides a
* method for finding the location of the robot.
* @author Nicholas Jones
* @version 4/14/2014
*/
public class WorldMap {
    private ArrayList<TagLocation> tagList;

    /**
    * Constructor
    * Reads in the Tag information from the file
    * @param url - The path of the file containing the map information.
    */
    public WorldMap(String url) throws IOException {
        this.tagList = new ArrayList<TagLocation>();

        Scanner scnr = new Scanner(new File(url));

        while (scnr.hasNext()) {
            Scanner line = new Scanner(scnr.nextLine());
            // TAG_ID TAG_X TAG_Y TAG ROTATION
            this.tagList.add(new TagLocation(line.nextInt(),
                                             line.nextDouble(),
                                             line.nextDouble(),
                                             0.0,
                                             line.nextDouble()));
        }
    }


    /**
    * Find the robot location in the world frame by comparing a tag location in
    * the robot frame to its location in the world frame.
    * @param tl - The TagLocation for a tag seen by the robot
    * @return {x, y, z, theta} of robot in world frame in meters and radians
    */
    public double[] findRobotGlobal(TagLocation tl) {

        for (TagLocation info : this.tagList) {
            if (info.id == tl.id) {
                System.out.printf("Found %d in map\n", info.id);
                double[] robot = new double[3];

                // TODO
                robot[0] = tl.x / .125;
                robot[1] = tl.y / .125;
                robot[2] = tl.rotation / .125;

                return robot;
            }
        }

        return null;
    }

    /**
    * @return A String representing the World Map
    */
    public String toString() {
        String s = "";
```

```
        for (TagLocation info : this.tagList) {
            s += info + "\n";
        }

        return s;
    }
}
```

```
0 1 0 0
```

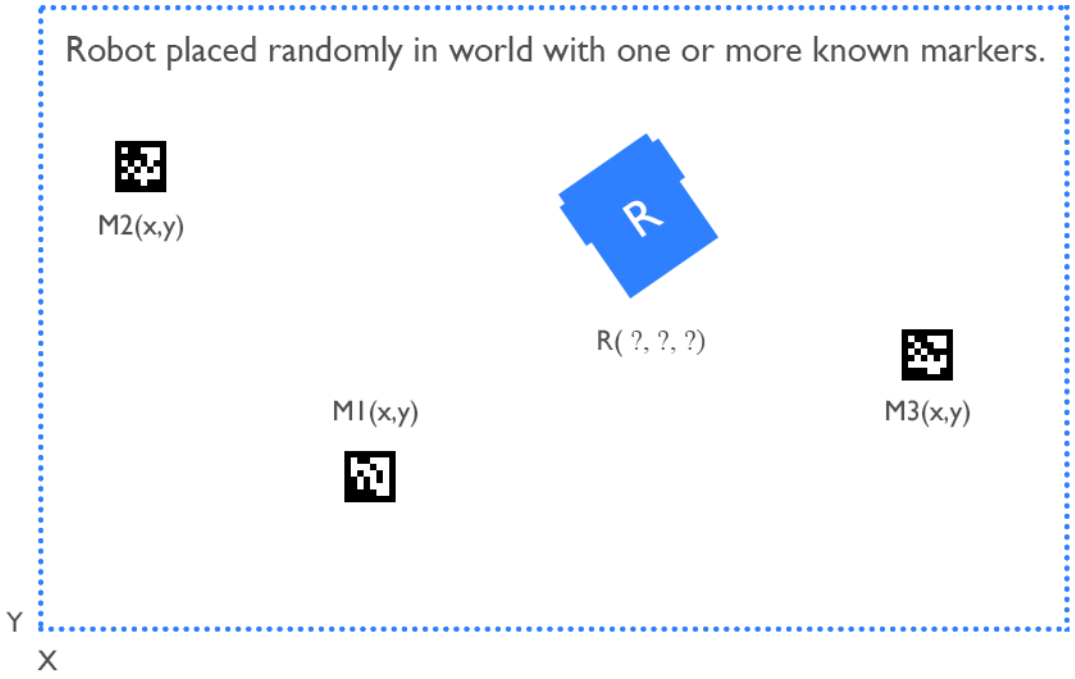
LAB 5 PROJECT PROPOSAL

Both Zach and Nick would have liked to do SLAM (Simultaneous Localization and Mapping) for the final project. Due to time and difficulty constraints, this is not possible, so we have decided to work on a smaller subsection of SLAM, localization.

The premise of the project is that the robot has been kidnapped, and it doesn't know where it is located in the world frame. Using a previously held map of its environment it must calculate where it is located and head to a location that we give it at run time.

The features that we will use to localize the robot will be either a series of colored strips located throughout the environment, or black and white markers from AprilTags or ARToolKit.

In order to calculate our location using these features, we will use a camera attached to the robot. This camera will either be the monocular camera used in previous labs or a Kinect camera given to us by Professor Vona for the purpose of this project. In order for the Kinect to work with the robot, we may need to swap out the PandaBoard used in previous labs with a more powerful netbook. This is because the PandaBoard may not support high enough usb bandwidth.


Robot placed randomly in world with one or more known markers.
M2(x,y)
R
R(?, ?, ?)
M1(x,y)
M3(x,y)
Y
X