```
#use "cmucamlib.ic"

/*
 * define the ports used
 */
#define RADAR_SERVO servo0
#define LEFT_MOTOR_PORT 2
#define RIGHT_MOTOR_PORT 3
#define GRIPPER_MOTOR_PORT 1
#define LEFT_REFLECTIVE_PORT 3
#define RIGHT_REFLECTIVE_PORT 2
#define ET_SENSOR_PORT 6

#define GRIPPER_LENGTH 0.4 * CLICKS_PER_FOOT


/*
 * The value above which a reading from a reflective
 * sensor would be considered black.
 */
#define BLACK_LEVEL 150

/*
 * radar servo values
 */
#define RADAR_CENTER 2100
#define RADAR_FULL_RIGHT 3650
#define RADAR_FULL_LEFT 700
#define RADAR_FULL_DEGREES 90.0

/*
 * Distance measurements
 */
#define CLICKS_PER_FOOT 47.0
#define CLICKS_PER_RIGHT_ANGLE 22.5

/*
 * STATE OF THE ROBOT
 ****************************************************************/
int radarDegrees = 0;      // direction of radar
int robotDegrees = 0;      // direction of robot
float robotXClicks = 0.0;  // x-coordinate of robot
float robotYClicks = 0.0;  // y-coordinate of robot
int gripperOpen = 0;       // gripper open or not
int hasCube = 0;           // has a cube or not in gripper
int leftIsBlack = 0;       // reflectivity sensor
int rightIsBlack = 0;      // reflectivity sensor
int et = 0;                // whether et sensor detects something...
int lost = 0;              // if robot is lost or not...

/*
 * RADAR CODE
 *********************************************************************/

void radarTo(int degrees) {
    if (degrees > 90) {
        degrees = 90;
    }
    if (degrees < -90) {
        degrees = -90;
    }
    radarDegrees = degrees;
    if (degrees == 0) {
        RADAR_SERVO = RADAR_CENTER;
        return;
    }
    if (degrees < 0) {
        RADAR_SERVO = RADAR_CENTER + (int)( ((float)(RADAR_FULL_LEFT -
RADAR_CENTER)) * (((float)(-degrees)) / RADAR_FULL_DEGREES));
    } else {
        RADAR_SERVO = RADAR_CENTER + (int)( ((float)(RADAR_FULL_RIGHT -
RADAR_CENTER)) * (((float)(degrees)) / RADAR_FULL_DEGREES));
    }
}

/*
 * Returns the difference between voo and doo
 */
int diff(int voo, int doo) {
    if (voo > doo) {
        return voo - doo;
    }
    return doo - voo;
}

/*
 * CMUCAM CODE
 *******************************************************************/

#define MIN_BLUE_R 140
#define MAX_BLUE_R 240
#define MIN_BLUE_G 190
#define MAX_BLUE_G 230
#define MIN_BLUE_B 60
#define MAX_BLUE_B 110

#define MIN_ORANGE_R 200
#define MAX_ORANGE_R 255
#define MIN_ORANGE_G 50
#define MAX_ORANGE_G 255
#define MIN_ORANGE_B 0
#define MAX_ORANGE_B 20

/*
 * The orange confidence above which it is orange...
 */
#define MIN_ORANGE_CONFIDENCE 20

/*
 * Better version of track_orange()...
 */
int trackOrange() {
    return trackRaw(MIN_ORANGE_R, MAX_ORANGE_R, MIN_ORANGE_G, MAX_ORANGE_G,
MIN_ORANGE_B, MAX_ORANGE_B);
```

```
}

/*
 * Better version of track_blue()...
 */
int trackBlue() {
    return trackRaw(MIN_BLUE_R, MAX_BLUE_R, MIN_BLUE_G, MAX_BLUE_G, MIN_BLUE_B,
MAX_BLUE_B);
}

/*
 * Centers the camera on orange.  The angle is kept track of
 * by the radarTo() function, which updates a variable, radarDegrees.
 */
int centerOnOrange() {
    int deathCount = 20;
    while (deathCount > 0 && trackOrange() > MIN_ORANGE_CONFIDENCE) {
        deathCount--;
        if (track_x > 0) {
            radarTo(radarDegrees + 1);
        }
        if (track_x < 0) {
            radarTo(radarDegrees - 1);
        }
        if (track_x == 0) {
            return 1;
        }
        if (radarDegrees < -90 || radarDegrees > 90) {
            return 0;
        }
    }
    return 0;
}

/*
 * Function that was never used for following blue blobs, like the sky...
 */
void followBlue() {
    while (trackBlue() > 50) {
        goStraight(10, 0, 0);
    }
}

/*
 * Checks if there is orange within the range min - max.  Where min and
 * max are degrees between -90 and 90, and min is less than max.
 */
int scanForOrange(int min, int max) {

    int degrees = min;
    int go = 1;

    radarTo(degrees);
    printf("Hello\n");
    beep();
    beep();
    tone(1000.0, 0.1);
    tone(700.0, 0.1);
    tone(300.0, 0.5);

    while (go && degrees < max) {
        radarTo(degrees);
        if (trackOrange() > MIN_ORANGE_CONFIDENCE) {
            while (trackOrange() > MIN_ORANGE_CONFIDENCE) {
                if (track_x > 0) {
                    degrees++;
                    radarTo(degrees);
                }
                if (track_x < 0) {
                    degrees--;
                    radarTo(degrees);
                }
                if (track_x == 0) {
                    go = 0;
                    break;
                }
                if (degrees < -90 || degrees > 90) {
                    return 0;
                }
            }
        }
        degrees += 5;
    }
    return !go;
}


/*
 * GRIPPER CODE
 *******************************************************************/

/*
 * Opens the gripper
 */
void openGripper() {
    if (!gripperOpen) {
        int shakePower = 20;

        motor(GRIPPER_MOTOR_PORT, 100);

        motor(LEFT_MOTOR_PORT, shakePower);
        motor(RIGHT_MOTOR_PORT, -shakePower);
        tone(600.0, 0.2);
        motor(LEFT_MOTOR_PORT, -shakePower);
        motor(RIGHT_MOTOR_PORT, shakePower);
        tone(670.0, 0.18);
        motor(LEFT_MOTOR_PORT, shakePower);
        motor(RIGHT_MOTOR_PORT, -shakePower);
        tone(720.0, 0.16);
        motor(LEFT_MOTOR_PORT, -shakePower);
        motor(RIGHT_MOTOR_PORT, shakePower);
        tone(760.0, 0.14);
        motor(LEFT_MOTOR_PORT, shakePower);
        motor(RIGHT_MOTOR_PORT, -shakePower);
        tone(790.0, 0.13);
        motor(LEFT_MOTOR_PORT, -shakePower);
        motor(RIGHT_MOTOR_PORT, shakePower);
        tone(800.0, 0.12);
        motor(LEFT_MOTOR_PORT, shakePower);
        motor(RIGHT_MOTOR_PORT, -shakePower);
        tone(807.0, 0.11);
        motor(LEFT_MOTOR_PORT, -shakePower);
        motor(RIGHT_MOTOR_PORT, shakePower);
        tone(810.0, 0.1);

        motor(LEFT_MOTOR_PORT, 0);
        motor(RIGHT_MOTOR_PORT, 0);

        sleep(1.0);

        ao();

        gripperOpen = 1;
    }
}

/*
 * Closes the gripper
 */
void closeGripper() {
    if (gripperOpen) {
        motor(GRIPPER_MOTOR_PORT, -100);
        tone(500.0, 0.50);
        tone(570.0, 0.50);
        tone(500.0, 0.25);
        tone(600.0, 0.25);
        tone(700.0, 0.25);
        tone(670.0, 0.25);
        motor(GRIPPER_MOTOR_PORT, 0);
        gripperOpen = 0;
    } else {
        motor(GRIPPER_MOTOR_PORT, -100);
        sleep(1.0);
        motor(GRIPPER_MOTOR_PORT, 0);
    }
}

/*
 * Grabs orange thing in front of robot, going forward while
 * orange thing is too far away.
 */
int grab() {

    int clicks = 0;

    radarTo(0);
    sleep(1.0);

    reset_encoder(0);
    reset_encoder(1);
    openGripper();

    while (clicksFromWall(robotXClicks, robotYClicks) > GRIPPER_LENGTH * 1.5 &&
trackOrange() > MIN_ORANGE_CONFIDENCE) {
        if (track_x > -10 && track_x < 10) {
            clicks += 5;
            goStraight2(clicks, 10, 0, 0);
            if (track_y < 20) {
                break;
            }
        } else {
            break;
        }
    }
    updateRobotClicks(clicks);
    ao();
    clicks = (int)clicksFromWall(robotXClicks, robotYClicks);
    if ((float)clicks < GRIPPER_LENGTH * 1.5) {
        reset_encoder(0);
        reset_encoder(1);
        clicks = (int)(1.5 * GRIPPER_LENGTH) - clicks;
        goStraight2(clicks, -20, 0, 0);
        updateRobotClicks(clicks);
        ao();
    }
    closeGripper();
    radarTo(0);
    sleep(0.5);
    if (trackOrange() > MIN_ORANGE_CONFIDENCE) {
        hasCube = 1;
    }
}

/*
 * Tracks orange things by turning and moving torwards them, and
 * then grabbing them.  Look Out....
 */
int turnAndGrab() {
    int clicks = 0;
    int targetAngle;
    reset_encoder(0);
    reset_encoder(1);

    if (scanForOrange(-80, 80)) {
        targetAngle = radarDegrees;
        turn(targetAngle, 1);
        grab();
        ao();
    } else {
        tone(200.0, 2.0);
        return 0;
    }
}

/*
 * TURNING FUNCTIONS
 ******************************************************************/
```

```
/*
 * Tries to turn using CMU Cam as anchor angle, returns
 * if successful.
 */
int turnWithCMUCam(int todegrees) {

    int degrees = -90;
    int max = 90;
    int go = 1;
    int target;

    int deathCount;

    todegrees = (int)((float)todegrees * 1.1);

    if (todegrees > 0) {
        degrees = 0;
    } else {
        max = 0;
    }

    radarTo(degrees);
    printf("Hello\n");

    go = !scanForOrange(degrees, max);
    degrees = radarDegrees;

    if (go) {
        tone(700.0, 0.1);
        tone(500.0, 0.2);
        tone(300.0, 0.3);
        tone(200.0, 0.5);
        return 0;
    }

    tone(300.0, 0.1);
    tone(500.0, 0.1);
    tone(400.0, 0.2);

    target = degrees - todegrees;
    radarTo(target);
    turnWithEncoders(todegrees);

    sleep(0.1);

    deathCount = 5;
    while (deathCount > 0) {
        int dif;
        float sleeptime;

        centerOnOrange();

        dif = diff(radarDegrees, target);
        sleeptime = ((float)dif * 0.15 / 90.0);
        if (dif < 2) {
            ao();
            radarTo(0);
            return 1;
        }
        if (dif < 4) {
            deathCount--;
        }
        if (sleeptime < 0.05) {
            sleeptime = 0.05;
        }

        while (radarDegrees < target) {
            if (centerOnOrange()) {
                motor(LEFT_MOTOR_PORT, -50);
                motor(RIGHT_MOTOR_PORT, 50);
                radarTo((int)((float)radarDegrees + sleeptime * 90.0));
                sleep(sleeptime);
                ao();
            } else {
                ao();
                tone(100.0, 2.0);
                return 0;
            }
        }
        while (radarDegrees > target) {
            if (centerOnOrange()) {
                motor(LEFT_MOTOR_PORT, 50);
                motor(RIGHT_MOTOR_PORT, -50);
                radarTo((int)((float)radarDegrees - sleeptime * 90.0));
                sleep(sleeptime);
                ao();
            } else {
                ao();
                tone(100.0, 2.0);
                return 0;
            }
        }
    }
    return 1;
}

/*
 * Turns using the encoders to count how far it thinks it has turned.
 */
void turnWithEncoders(int degrees) {

    if (degrees != 0) {

        int left;
        int right;
        int diff = 0;
        int clicks = (int)(CLICKS_PER_RIGHT_ANGLE * (float)degrees/90.0);

        if (degrees > 0) {
            left = 70;
            right = -70;
        } else {
            clicks = -clicks;
            left = -70;
            right = 70;
        }

        while (diff < clicks) {
            diff = read_encoder(0) + read_encoder(1);
            motor(LEFT_MOTOR_PORT, left);
            motor(RIGHT_MOTOR_PORT, right);
        }
        ao();
    }
}

/*
 * Makes sure that the robot doesn't turn more than 90
 * degrees at a time because the camera servo could not
 * move much farther than +/- 90 degrees.
 */
void turnLessThan90(int degrees, int usecmu) {
    reset_encoder(0);
    reset_encoder(1);
    if (usecmu) {
        if (!turnWithCMUCam(degrees)) {
            turnWithEncoders(degrees);
        }
    } else {
        turnWithEncoders(degrees);
    }
    robotDegrees += degrees;
    if (degrees > 180) {
        degrees = degrees - 360;
    }
    if (degrees < -180) {
        degrees = degrees + 360;
    }
}

/*
 * Easy function to call, just turn this many degrees...
 */
void turn(int degrees, int usecmu) {
    while (degrees > 90) {
        turnLessThan90(90, usecmu);
        degrees -= 90;
    }
    while (degrees < -90) {
        turnLessThan90(-90, usecmu);
        degrees += 90;
    }
    turnLessThan90(degrees, usecmu);
}

/*
 * Turns the robot to the specified toDegrees.  The angle of
 * the robot is measured as 0 degrees west, 90 degrees north,
 * -90 south, and +/- 180 as east, with all the other angles
 * in between also being used.
 */
void turnTo(int toDegrees, int usecmu) {
    //    printf("TURNING %d\n", toDegrees);
    //  sleep(3.0);
    if (toDegrees < 0) {
        if (robotDegrees > 0) {
            int td = 360 + toDegrees;
            if (td - robotDegrees > 180) {
                // shortest turn = -(360 - td + robotDegrees);
                turn(td - 360 - robotDegrees, usecmu);
            } else {
                // shortest turn = td - robotDegrees...
                turn(td - robotDegrees, usecmu);
            }
        } else {
            // both are negative...
            turn(toDegrees - robotDegrees, usecmu);
        }
    } else {
        // toDegrees > 0
        if (robotDegrees > 0) {
            // both are positive...
            turn(toDegrees - robotDegrees, usecmu);
        } else {
            int rd = 360 + robotDegrees;
            if (rd - toDegrees > 180) {
                // shortest = toDegrees + 360 - rd...
                turn(toDegrees + 360 - rd, usecmu);
            } else {
                // shortest turn = toDegrees - rd;
                turn(toDegrees - rd, usecmu);
            }
        }
    }
    robotDegrees = toDegrees;
}

/*
 * GO STRAIGHT
 *******************************************************************/

/*
 * Updates the robots location by the number of clicks given.
 */
void updateRobotClicks(int straightClicks) {
    float a = (float)robotDegrees * 3.14159 / 180.0;
    robotXClicks += sin(a) * (float)straightClicks;
    robotYClicks += cos(a) * (float)straightClicks;
}

/*
 * Tells the robot to go straight, but does not reset the
 * encoders.  This is a lower level function and is used
 * mainly to factor out code and stuff, not for easy readibility.
 */
void goStraight2(int clicks, int power, int checkForTape, int checkET) {
    motor(LEFT_MOTOR_PORT, power);
    motor(RIGHT_MOTOR_PORT, power);

    if (checkForTape) {
        if (checkET) {
            while (read_encoder(0) + read_encoder(1) < clicks) {
                leftIsBlack = analog(LEFT_REFLECTIVE_PORT) > BLACK_LEVEL;
                rightIsBlack = analog(RIGHT_REFLECTIVE_PORT) > BLACK_LEVEL;
                if (leftIsBlack || rightIsBlack) {
                    ao();
                    return;
                }
                if (analog(ET_SENSOR_PORT) > BLACK_LEVEL) {
                    et = 1;
                    ao();
                    return;
                } else {
                    et = 0;
                }
            }
        } else {
            while (read_encoder(0) + read_encoder(1) < clicks) {
                leftIsBlack = analog(LEFT_REFLECTIVE_PORT) > BLACK_LEVEL;
                rightIsBlack = analog(RIGHT_REFLECTIVE_PORT) > BLACK_LEVEL;
                if (leftIsBlack || rightIsBlack) {
                    ao();
                    return;
                }
            }
        }
    } else {
        if (checkET) {
            while (read_encoder(0) + read_encoder(1) < clicks) {
                if (analog(ET_SENSOR_PORT) > BLACK_LEVEL) {
                    ao();
                    return;
                    et = 1;
                } else {
                    et = 0;
                }
            }
        } else {
            while (read_encoder(0) + read_encoder(1) < clicks) {
                // foo....
            }
        }
    }
    ao();
    return;
}

/*
 * Makes the robot go straight as many clicks as you want.
 */
void goStraight(int clicks, int checkForTape, int checkET) {
    reset_encoder(0);
    reset_encoder(1);
    if (checkForTape) {
        goStraight2(clicks, 20, checkForTape, checkET);
    } else {
        goStraight2(clicks, 50, checkForTape, checkET);
    }
}

/*
 * ALIGN CODE
 *****************************************************************/

int leftAlignIters;
int rightAlignIters;

/*
 * Makes the robot align with black tape.
 */
void align2() {

    int goal;

    int forwardSpeed = 15;
    int reverseSpeed = -60;

    int maxForwardCount = 8;
    int maxBrakeCount = 2;
    int forwardCount = 0;
    int brakeCount = 0;

    int maxReverseCount = 5;
    int leftReverseCount = 0;
    int rightReverseCount = 0;

    int leftPower;
    int rightPower;

    leftAlignIters = 0;
    rightAlignIters = 0;

    while (1) {

        leftIsBlack = analog(LEFT_REFLECTIVE_PORT) > BLACK_LEVEL;
        rightIsBlack = analog(RIGHT_REFLECTIVE_PORT) > BLACK_LEVEL;

        if (leftReverseCount > 0) {
            if (leftIsBlack) {
                leftReverseCount = maxReverseCount;
            } else {
                leftReverseCount--;
            }
            leftPower = reverseSpeed;
        } else {
            if (leftIsBlack) {
                if (rightReverseCount == 0 && rightIsBlack) {
                    break;
                } else {
                    leftReverseCount = maxReverseCount;
                    leftPower = reverseSpeed;
                }
            } else {
                leftPower = forwardSpeed;
            }
        }

        if (rightReverseCount > 0) {
            if (rightIsBlack) {
                rightReverseCount = maxReverseCount;
            } else {
                rightReverseCount--;
            }
            rightPower = reverseSpeed;
        } else {
            if (rightIsBlack) {
                if (leftReverseCount == 0 && leftIsBlack) {
                    break;
                } else {
                    rightReverseCount = maxReverseCount;
                    rightPower = reverseSpeed;
                }
            } else {
                rightPower = forwardSpeed;
            }
        }
        if (leftPower < 0) {
            leftAlignIters++;
        }
        if (rightPower < 0) {
            rightAlignIters++;
        }
        if (leftPower > 0 && rightPower > 0) {
            if (forwardCount == 0) {
                if (brakeCount == 0) {
                    forwardCount = maxForwardCount;
                } else {
                    brakeCount--;
                }
                motor(LEFT_MOTOR_PORT, -leftPower);
                motor(RIGHT_MOTOR_PORT, -rightPower);
            } else {
                motor(LEFT_MOTOR_PORT, leftPower);
                motor(RIGHT_MOTOR_PORT, rightPower);
                forwardCount--;
                if (forwardCount == 0) {
                    brakeCount = maxBrakeCount;
                }
            }
        } else {
            motor(LEFT_MOTOR_PORT, leftPower);
            motor(RIGHT_MOTOR_PORT, rightPower);
        }
    }
    ao();

    goal = getClosestGoal(robotXClicks, robotYClicks);
    robotXClicks = (float)goalXs[goal];
    robotYClicks = (float)goalYs[goal];

    return;
}

#define ALIGN_ITER_RIGHT_ANGLE 7.0

/*
 * Makes the robot turn to the nearest right angle - north, south,
 * east, or west...
 */
void turnToNearestRightAngle() {
    if (robotDegrees >= -45 && robotDegrees <= 45) {
        turnTo(0, 0);
    } else if (robotDegrees >= -135 && robotDegrees <= -45) {
          turnTo(-90, 0);
      } else if (robotDegrees >= -180 && robotDegrees <= -135) {
            turnTo(-180, 0);
        } else if (robotDegrees <= 135 && robotDegrees >= 45) {
              turnTo(90, 0);
          } else if (robotDegrees <= 180 && robotDegrees >= 135) {
                turnTo(180, 0);
            } else {
                /// errorrrrr.....
            }
}

/*
 * Makes the robot center itself inside a goal by aligning itself
 * on three different sides of the goal.
 */
void situate() {

    int i = 0;
    int maxChecks = 3;
    while (1) {

        i++;

        turnToNearestRightAngle();

        reset_encoder(0);
        reset_encoder(1);
        align2();

        reset_encoder(0);
        reset_encoder(1);
        goStraight2((int)(CLICKS_PER_FOOT * 0.3), -30, 0, 0);

        if (i == maxChecks) {
            int g;
            motor(RIGHT_MOTOR_PORT, 30);
            motor(LEFT_MOTOR_PORT, 30);
            sleep(0.05);
            ao();
            if (robotDegrees >= -45 && robotDegrees <= 45) {
                robotDegrees = 0;
            } else if (robotDegrees >= -135 && robotDegrees <= -45) {
                  robotDegrees = -90;
              } else if (robotDegrees >= -180 && robotDegrees <= -135) {
                    robotDegrees = -180;
                } else if (robotDegrees <= 135 && robotDegrees >= 45) {
                      robotDegrees = 90;
                  } else if (robotDegrees <= 180 && robotDegrees >= 135) {
                        robotDegrees = 180;
                    } else {
                        /// errorrrrr.....
                    }
            g = getClosestGoal(robotXClicks, robotYClicks);
            if (g >= 0) {
                robotXClicks = (float)goalXs[g];
                robotYClicks = (float)goalYs[g];
            }
            return;
        } else {
            int add = (int)((float)(rightAlignIters - leftAlignIters) /
ALIGN_ITER_RIGHT_ANGLE);
            printf("Adding: %d\n", add);
            sleep(3.0);
            robotDegrees += add;
            while (robotDegrees > 180) {
                robotDegrees = robotDegrees - 360;
            }
            while (robotDegrees < -180) {
                robotDegrees = robotDegrees + 360;
            }
            turn(90, 0);
            reset_encoder(0);
            reset_encoder(1);
            goStraight2((int)(CLICKS_PER_FOOT * 0.07), -30, 0, 0);
        }
    }
}

#define ARENA_WIDTH 8.0 * CLICKS_PER_FOOT
#define ARENA_HEIGHT 12.0 * CLICKS_PER_FOOT

/*
 * Returns the distance from the closest wall, in encoder clicks.
 */
float clicksFromWall(float x, float y) {
    float xprox;
    float yprox;
    if (-x < (ARENA_WIDTH + x)) {
        // x is closer to north than south...
        xprox = -x;
    } else {
        // x is closer to south than north...
        xprox = ARENA_WIDTH + x;
    }

    if (y < ARENA_HEIGHT - y) {
        // y is closer to east...
        yprox = y;
    } else {
        // y is closer to west...
        yprox = ARENA_HEIGHT - y;
    }

    if (xprox < yprox) {
        return xprox;
    } else {
        return yprox;
    }
}

/*
 * Plays the music that is sounded off whenever it thinks a goal is made.
 */
void playGoalMusic() {
    tone(630.0, 1.0);
    tone(1000.0, 1.3);
    tone(630.0, 0.75);
}

/*
 * Called to release cube, if it has one.  This function
 * should only be called when,,, the robot is at the tape.
 */
void releaseCubeAtTape() {
    if (!gripperOpen) {
        openGripper();
        playGoalMusic();
        printf("GOAL: Press Start...\n");
        while (!start_button());
        // REMIND: check for cube...
        hasCube = 0;
    }
}

/*
 * Makes the robot move away, I mean into, the goal.
 */
void moveIntoGoal() {
    // go into black square and align position...
    if (diff(robotDegrees, 0) < 10) {
        align2();
        robotDegrees = 0;
    } else if (diff(robotDegrees, 90) < 10) {
          align2();
          robotDegrees = 90;
      } else if (diff(robotDegrees, 180) < 10) {
            align2();
            robotDegrees = 180;
        } else if (diff(robotDegrees, -90) < 10) {
              align2();
              robotDegrees = -90;
          } else if (diff(robotDegrees, -180) < 10) {
                align2();
                robotDegrees = -180;
            }

            goStraight((int)(0.25 * CLICKS_PER_FOOT), 0, 0);
    updateRobotClicks((int)(0.25 * CLICKS_PER_FOOT));
    radarTo(0);
    closeGripper();
    if (trackOrange() > MIN_ORANGE_CONFIDENCE) {
        releaseCubeAtTape();
    } else {
        openGripper();
    }
    situate();
}

/*
 * Causes the robot to move to the specified x, y coordinates.
 */
void moveTo(int x, int y, int usecmu, int checkForTape, int checkET, int isCube)
{

    float dx;
    float dy;
    float r;    // distance to go float
    float ato;  // angle to go float
    int togo;   // distance to go integer...
    int toga;   // angle to go integer

    if ((float)y == robotYClicks) {
        if ((float)x == robotXClicks) {
            return;
        }
        if ((float)x > robotXClicks) {
            toga = 90;
            togo = (int)((float)x - robotXClicks);
        } else {
            toga = -90;
            togo = (int)robotXClicks - x;
        }
    } else {

        dx = (float)x - robotXClicks;
        dy = (float)y - robotYClicks;
        r = sqrt(dx * dx + dy * dy);
        togo = (int)r;

        ato = atan(dx/dy);
        if (dy <= 0.0) {
            if (dx <= 0.0) {
                ato = -3.14159 + ato;
            }
            if (dx > 0.0) {
                ato = 3.14159 + ato;
            }
        }

        // convert to degrees...
        toga = (int)(180.0 * ato / 3.14159);
        // normalize...
        while (toga > 180) {
            toga = toga - 360;
        }
        while (toga < -180) {
            toga = toga + 360;
        }
    }

    if (insideGoal(robotXClicks, robotYClicks)) {
        int g = getClosestGoal(robotXClicks, robotYClicks);
        reset_encoder(0);
        reset_encoder(1);
        goStraight2((int)(CLICKS_PER_FOOT * 0.1), -30, 0, 0);

        if (toga >= -45 && toga <= 45) {
            turnTo(0, 0);
            robotXClicks = (float)goalXs[g];
            robotYClicks = (float)goalYs[g] + CLICKS_PER_FOOT * 0.4;
        } else if (toga >= -135 && toga <= -45) {
              turnTo(-90, 0);
              robotXClicks = (float)goalXs[g] - CLICKS_PER_FOOT * 0.4;
              robotYClicks = (float)goalYs[g];
          } else if (toga >= -180 && toga <= -135) {
                turnTo(-180, 0);
                robotXClicks = (float)goalXs[g];
                robotYClicks = (float)goalYs[g] - CLICKS_PER_FOOT * 0.4;
            } else if (toga <= 135 && toga >= 45) {
                  turnTo(90, 0);
                  robotXClicks = (float)goalXs[g] + CLICKS_PER_FOOT * 0.4;
                  robotYClicks = (float)goalYs[g];
              } else if (robotDegrees <= 180 && robotDegrees >= 135) {
                    turnTo(180, 0);
                    robotXClicks = (float)goalXs[g];
                    robotYClicks = (float)goalXs[g] - CLICKS_PER_FOOT * 0.4;
                } else {
                    /// errorrrrr.....
                }
        align2();
        turnTo(toga + 3, usecmu);
        goStraight((int)CLICKS_PER_FOOT, 0, 0);
        updateRobotClicks((int)CLICKS_PER_FOOT);
        togo -= (int)CLICKS_PER_FOOT;
    } else {
        turnTo(toga, usecmu);
    }

    if (isCube) {
        togo = togo - (int)(1.5 * GRIPPER_LENGTH);
    }

    // move it!
    goStraight(togo, checkForTape, checkET);
    updateRobotClicks(togo);
    if (checkForTape) {
        if (leftIsBlack || rightIsBlack) {
            moveIntoGoal();
        } else {
            ao();
            lost = 1;
        }
    }
}

#define MAX_CUBES 16

int cubeXs[MAX_CUBES];
int cubeYs[MAX_CUBES];
int numCubes = 0;

void addCubeClicks(int cx, int cy) {
    if (numCubes == MAX_CUBES) {
        return;
    }
    cubeXs[numCubes] = cx;
    cubeYs[numCubes] = cy;
    numCubes++;
}

void addCube(float xfeet, float yfeet) {
    addCubeClicks((int)(xfeet * CLICKS_PER_FOOT), (int)(yfeet *
CLICKS_PER_FOOT));
}

void removeCube(int index) {
    if (numCubes == 0) {
        return;
    }
    numCubes--;
    while (index < numCubes) {
        cubeXs[index] = cubeXs[index + 1];
        cubeYs[index] = cubeYs[index + 1];
        index++;
    }
}

int getClosestCube(float xclicks, float yclicks) {
    int i = 0;
    int minAt = -1;
    float min = 8.0 * 11.0 * CLICKS_PER_FOOT;
    while (i < numCubes) {
        float xd = xclicks - (float)cubeXs[i];
        float yd = yclicks - (float)cubeYs[i];
        float r = sqrt(xd * xd + yd * yd);
        if (r < min) {
            min = r;
            minAt = i;
        }
        i++;
    }
    return minAt;
}
```

```
int getClosestCubeAwayFromWall(float xclicks, float yclicks, float
minDistFromWall) {
    int i = 0;
    int minAt = -1;
    float min = 8.0 * 11.0 * CLICKS_PER_FOOT;
    while (i < numCubes) {
        float xd = xclicks - (float)cubeXs[i];
        float yd = yclicks - (float)cubeYs[i];
        float r = sqrt(xd * xd + yd * yd);
        if (clicksFromWall((float)cubeXs[i], (float)cubeYs[i]) >
minDistFromWall) {
            if (r < min) {
                min = r;
                minAt = i;
            }
        }
        i++;
    }
    return minAt;
}


#define MAX_GOALS 4

int goalXs[MAX_GOALS];
int goalYs[MAX_GOALS];
int numGoals = 0;

void addGoalClicks(int x, int y) {
    if (numGoals == MAX_GOALS) {
        return;
    }
    goalXs[numGoals] = x;
    goalYs[numGoals] = y;
    numGoals++;
}

void addGoal(float xfeet, float yfeet) {
    addGoalClicks((int)(xfeet * CLICKS_PER_FOOT), (int)(yfeet *
CLICKS_PER_FOOT));
}

int insideGoal(float xclicks, float yclicks) {
    int i = 0;
    while (i < numGoals) {
        if (xclicks >= (float)goalXs[i] - CLICKS_PER_FOOT/2.0 && xclicks <
(float)goalXs[i] + CLICKS_PER_FOOT/2.0
            && yclicks >= (float)goalYs[i] - CLICKS_PER_FOOT/2.0 && yclicks <
(float)goalYs[i] + CLICKS_PER_FOOT/2.0) {
            return 1;
        }
        i++;
    }
    return 0;
}

int getClosestGoal(float xclicks, float yclicks) {
```

```c
    int i = 0;
    int minAt = 0;
    float min = 8.0 * 11.0 * CLICKS_PER_FOOT;
    while (i < numGoals) {
        float xd = xclicks - (float)goalXs[i];
        float yd = yclicks - (float)goalYs[i];
        float r = sqrt(xd * xd + yd * yd);
        if (r < min) {
            min = r;
            minAt = i;
        }
        i++;
    }
    return minAt;
}

#define MAX_WAYPOINTS 10

#define TURN_WAYPOINT 0
#define GRAB_WAYPOINT 1
#define GOAL_WAYPOINT 2

int waypointXs[MAX_WAYPOINTS];  // waypoint x-coordinates...
int waypointYs[MAX_WAYPOINTS];  // waypoint y-coordinates...
int waypointTs[MAX_WAYPOINTS];  // waypoint types (i.e. goal or cube)...
int numWaypoints = 0;  // number of waypoints...


void addWaypointClicks(int x, int y, int waypointType) {
    if (numWaypoints >= 10) {
        return;
    }
    waypointXs[numWaypoints] = x;
    waypointYs[numWaypoints] = y;
    waypointTs[numWaypoints] = waypointType;
    numWaypoints++;
}

/*
 * Just a function to use for losers like me for testing...
 */
void addWaypoint(float xfeet, float yfeet) {
    addWaypointClicks((int)(CLICKS_PER_FOOT * xfeet), (int)(CLICKS_PER_FOOT *
yfeet), TURN_WAYPOINT);
}

/*
 * Strategy Foo M1-A1...
 * Returns if successfully made a plan or not...
 * Planning as follows...
 * 1) Find closest cube that is at either the x-coordinate or y-coordinate
 *    (+- whatever clicks) of the current location...  Go grab that cube
 *    and then go to closest goal to release cube...
 * 2) If there are no cubes that match current loc's x or y, then find
 *    closest cube and deliver to closest goal...
 * 3) Otherwise, self-destruct...
 */
```

```
int plan() {

    int i = 0;
    int g = getClosestGoal(robotXClicks, robotYClicks);
    float min = 8.0 * 12.0 * CLICKS_PER_FOOT;
    int minAt = -1;

    // forget about previous plans
    numWaypoints = 0;
    while (i < numCubes) {
        // check if same x- or y-coords of current loc...
        if (g == getClosestGoal((float)cubeXs[i], (float)cubeYs[i])
            && clicksFromWall((float)cubeXs[i], (float)cubeYs[i]) > 0.5 *
CLICKS_PER_FOOT
            && ((float)diff(cubeXs[i], (int)robotXClicks) < GRIPPER_LENGTH
                || (float)diff(cubeYs[i], (int)robotYClicks) < GRIPPER_LENGTH))
{
            // got a freakin' match or something...
            // check distance...
            float xd = robotXClicks - (float)cubeXs[i];
            float yd = robotYClicks - (float)cubeYs[i];
            float r = sqrt(xd * xd + yd * yd);
            if (r < min) {
                min = r;
                minAt = i;
            }
        }
        i++;
    }

    if (minAt >= 0) {
        addWaypointClicks(cubeXs[minAt], cubeYs[minAt], GRAB_WAYPOINT);
        addWaypointClicks(goalXs[g], goalYs[g], GOAL_WAYPOINT);
        return 1;
    }

    i = 0;
    while (i < numCubes) {
        if (getClosestGoal((float)cubeXs[i], (float)cubeYs[i]) == g) {
            i = getClosestCubeAwayFromWall(robotXClicks, robotYClicks, 0.5 *
CLICKS_PER_FOOT);
            if (i >= 0) {
                addWaypointClicks(cubeXs[i], cubeYs[i], GRAB_WAYPOINT);
                addWaypointClicks(goalXs[g], goalYs[g], GOAL_WAYPOINT);
                return 1;
            }
        }
        i++;
    }

    // if here, then could not find any good cube to grab...
    // find the next closest goal...

    beep();
    beep();
    tone(600.0, 0.5);
    tone(400.0, 0.5);
    tone(150.0, 0.7);

    i = 0;
    min = 8.0 * 12.0 * CLICKS_PER_FOOT;
    minAt = -1;

    while (i < numGoals) {
        if (i != g) {
            // make sure there is a cube whose closest goal is at 'i'...

            int c = 0;
            int hasClosestCube = 0;

            while (c < numCubes) {
                if (getClosestGoal((float)cubeXs[c], (float)cubeYs[c]) == i) {
                    hasClosestCube = 1;
                    break;
                }
                c++;
            }
            if (hasClosestCube) {
                float dx = (float)goalXs[i] - robotXClicks;
                float dy = (float)goalYs[i] - robotYClicks;
                float r = sqrt(dx * dx + dy * dy);
                if (r < min) {
                    min = r;
                    minAt = i;
                }
            }
        }
        i++;
    }

    if (minAt >= 0) {
        addWaypointClicks(goalXs[minAt], goalYs[minAt], GOAL_WAYPOINT);
        return 1;
    }

    return 0;
}

float dist(float x1, float y1, float x2, float y2) {
    float dx = x2 - x1;
    float dy = y2 - y1;
    return sqrt(dx * dx + dy * dy);
}

void go() {
    int i = 0;

    while (i < numWaypoints) {
        int closestCube = getClosestCube(robotXClicks, robotYClicks);
        int tryCMU = 0;
        float dist = dist((float)cubeXs[closestCube],
(float)cubeYs[closestCube], robotXClicks, robotYClicks);
        if (dist < CLICKS_PER_FOOT * 2.0) {
            tryCMU = !hasCube;
        }
        if (waypointTs[i] == GRAB_WAYPOINT) {
            // then found a cube on same x- or y-axis as current loc...

            if ((float)diff(waypointXs[i], (int)robotXClicks) < GRIPPER_LENGTH)
{

                // then on same x-coord...
                if ((float)waypointYs[i] < robotYClicks) {
                    turnTo(180, tryCMU);
                } else {
                    turnTo(0, tryCMU);
                }

                if (insideGoal(robotXClicks, robotYClicks)) {
                    // align with black tape...
                    align2();
                }
                goStraight(diff(waypointYs[i], (int)robotYClicks) -
(int)(GRIPPER_LENGTH), 0, 0);
                updateRobotClicks(diff(waypointYs[i], (int)robotYClicks) -
(int)(GRIPPER_LENGTH));
            } else if ((float)diff(cubeYs[i], (int)robotYClicks) <
GRIPPER_LENGTH) {
                  // then on same y-coord...
                  if ((float)waypointXs[i] < robotXClicks) {
                      turnTo(-90, tryCMU);
                  } else {
                      turnTo(90, tryCMU);
                  }

                  if (insideGoal(robotXClicks, robotYClicks)) {
                      // align with black tape...
                      align2();
                  }
                  goStraight(diff(waypointXs[i], (int)robotXClicks) -
(int)(GRIPPER_LENGTH), 0, 0);
                  updateRobotClicks(diff(waypointXs[i], (int)robotXClicks) -
(int)(GRIPPER_LENGTH));
              } else {
                  moveTo(waypointXs[i], waypointYs[i], 0, 0, 0, 1);
              }
            motor(LEFT_MOTOR_PORT, -30);
            motor(RIGHT_MOTOR_PORT, -30);
            sleep(0.1);
            ao();
            grab();
            removeCube(getClosestCube(robotXClicks, robotYClicks));
        } else {
            if (waypointTs[i] == GOAL_WAYPOINT) {
                moveTo(waypointXs[i], waypointYs[i], tryCMU, 1, 0, 0);
            } else if (waypointTs[i] == TURN_WAYPOINT) {
                  moveTo(waypointXs[i], waypointYs[i], tryCMU, 0, 0, 0);
              }
        }
        sleep(1.0);
        i++;
```

```
    }
}

void goToClosestGoal() {
    int g = getClosestGoal(robotXClicks, robotYClicks);
    closeGripper();
    moveTo(goalXs[g], goalYs[g] - (int)(0.25 * CLICKS_PER_FOOT), 0, 1, 0, 0);
    openGripper();
}

void randomMotion() {
    while (1) {
        int tox = random((int)(CLICKS_PER_FOOT * 5.0));
        int toy = random((int)(CLICKS_PER_FOOT * 5.0));
        radarTo(0);
        closeGripper();
        if (trackOrange() > MIN_ORANGE_CONFIDENCE) {
            hasCube = 1;
        } else {
            hasCube = 0;
        }
        if (hasCube) {
            // then has a cube, so seek black tape...
            moveTo(tox, toy, 0, 1, 0, 0);
        } else {
            openGripper();
            // no cube yet... just continue...
            moveTo(tox, toy, 0, 0, 0, 0);
        }
    }
}

void main() {

    ao();
    numWaypoints = 0;
    RADAR_SERVO = RADAR_CENTER;
    init_expbd_servos(1);
    enable_encoder(0);
    enable_encoder(1);
    init_camera();

    openGripper();

    addCube(-7.0, 3.0);
    addCube(-2.0, 3.5);
    addCube(-6.0, 4.5);
    addCube(-6.0, 6.0);
    addCube(-1.0, 7.0);
    addCube(-1.0, 9.5);
    addCube(-6.5, 10.0);
    addCube(-7.5, 10.0);
    /**/
    addGoal(-1.0, 1.0);
    addGoal(-6.0, 3.0);
    addGoal(-2.0, 11.0);
    /**/
    robotXClicks = CLICKS_PER_FOOT * (-4.0);
    robotYClicks = CLICKS_PER_FOOT * 5.5;
    /**/
    while (1) {
        radarTo(0);
        printf("Press Start\n");
        while (!start_button());
        /**/
        //while (1) align2();
        /**/

        /*
        while (1) {
            if (centerOnOrange()) {
                tone(500.0, 0.05);
                tone(750.0, 0.1);
            } else {
                tone(100.0, 0.1);
            }
        }
        /**/

        goToClosestGoal();

        while (1) {
            //openGripper();
            //tone(700.0, 3.0);
            if (lost) {
                printf("Lost...\n");
                // randomly go...
                randomMotion();
            } else {
                printf("Planning...\n");
                if (plan()) {
                    printf("Going...\n");
                    go();
                } else {
                    printf("Done...\n");
                    break;
                }
            }
        }
    }

    ao();
    disable_encoder(0);
    disable_encoder(1);
    init_expbd_servos(0);
}
```