Maze Solving Quad-Rotor Aircraft

Temple University
Timothy Boger, Matthew Bosack, Michael Korostelev, John Ruddy

Abstract

Quad Rotor aircraft have seen a massive increase in development in recent years, but have lacked worthwhile applications. A proposed disaster scenario was the focus of a Drexel University sponsored Indoor Aerial Robotics Competition. Given a map of a predefined area, competitors must autonomously navigate the maze, which is representative of a buildings architecture. The maze will be populated with unknown obstacles and shortcuts, simulating a potential post-earthquake environment. Using designated begin and end points, the quad rotor completing the maze in the shortest time would be victorious. The Temple University team sought to complete these objectives through a novel design using readily available and advanced technologies. The quad rotor construction is based off a fiber glass frame driven by brushless DC motors powered by lithium polymer batteries. Control is handled by an Arduino processing board, namely the Android Development Kit, which allows it to interface with an Android phone via USB. The open structure and hardware package in the Android phone, a Samsung Nexus S, provides sufficient processing power, computer vision, and communication utilities, creates an ideal embedded platform for the task. In conjunction with the Arduino board, a fully autonomous, maze solving quad rotor aircraft is developed.

Introduction

Design of Unmanned Aerial Vehicles (UAV) is not a task to be taken lightly. A great deal of planning is required before even starting to lay ground work, and needs a multi-disciplinary and dedicated team to succeed. From design, construction, modeling, testing, and implementing, no project phase passes without challenges. However it is these same challenges that motivate progress. The following report closely documents the decisions and implementations for design and construction of an autonomous maze solving aircraft. Nearly entirely built and programmed from scratch, the project is closing on its goal. Beginning with planning, the report progresses through the assembly, sensor usage, control design, and maze solving techniques used in order to accomplish exploration of a real world disaster scenario. Work is not yet completed, but steadily advancing, with major improvements and redesign phases facilitating the development. Full documentation will be disclosed upon completion.

Design

The design methodology for the quad-rotor aircraft began with addressing a few essential needs:

1) Input and solve a maze
2) Avoid unknown obstacles
3) Start/Stop/Override and Monitor with base station
4) Detect and land in designated zone

The solutions for these were determined by

1) A Java GUI is developed on a base station PC that allows the maze to be input. The maze is transmitted to an Android phone for solving, then issuing commands for movement.
2) Obstacles can be detected and verified in two methods: ultrasonic range finding sensors, and computer vision. The sensors can be mounted on the quad arms, and the computer vision is implemented on the Android Phone.
3) While the majority of the work is to be autonomous, Start and Stop commands must be issued, as well as a safety over ride and progress monitoring. This is all handled between a PC base station using a wireless router to communicate with the Android phones wireless capability.
4) In the generated scenario, a landing zone is designated as a red circle, which can be detected through the computer vision process on the Android Phone.

These distinctions allowed us determine we would need to interface the ultrasonic sensors and Android Phone. The Arduino microcontroller boards, a popular and easy to program solution, was selected based on its ability to establish USB connectivity to the Android Phone, allow analog input from the sensors, as well as generate Pulse Width Modulation (PWM) signals to drive brushless DC motors, with additional RS-232 interfaces for communication with an Inertial Measurement Unit (IMU) already available from the lab. With the majority of the hardware selected, a frame and motors could be decided on that could support these elements and generate enough lift to efficiently navigate enclosed areas. The final selection was made to power the system from rechargeable lithium polymer batteries. Table 1 outlines the equipment obtained with general pricing information.

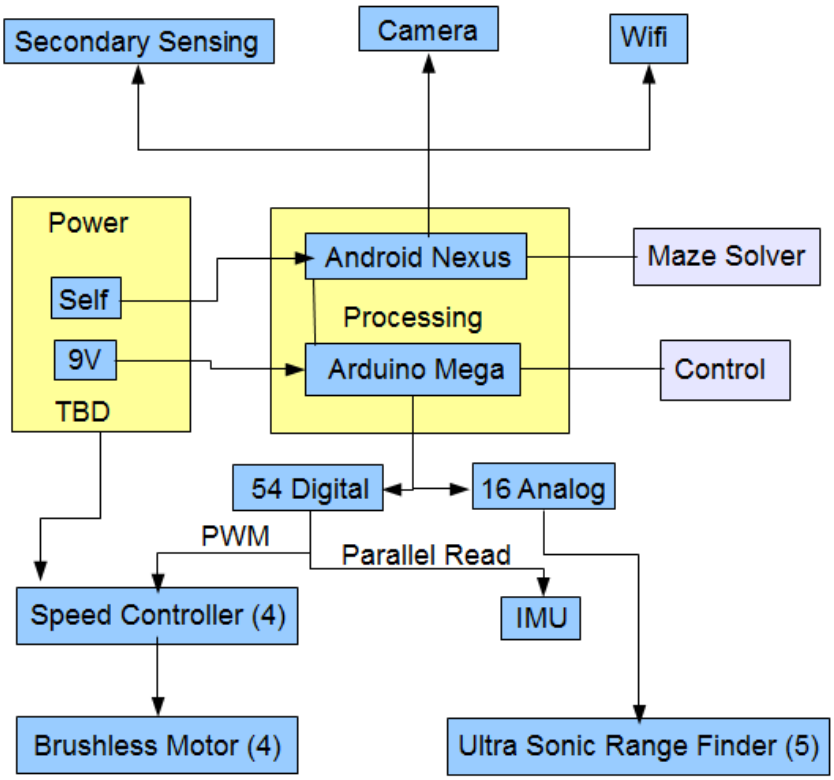

Figure 1: Underlying block diagram of system planning.

Figure 2: Fully constructed frame, calling out location for phone and controller housing, with ultrasonic sensor positions.

Table 1: IARC Proposed Hardware and Budget

Item	Description	Unit Cost	Quantity	Total
Nexus S	Image Pro, Maze Solver	$300	1	$300
Arduino ADK	Ctrl Prcsing	$80	1	$80
Ultrasonic	Ranging	$30	5	$150
Base Kit	Frame, Motors, ESC, Pwr Dist.	$250	1	$250
IMU	Ctrl Sensors			
Lipo w/ charger	PWR	$100	1	$100
Misc.	-	$100		$100
				$1100

Construction

The frame was constructed from a quad copter fiberglass kit. Each piece was sanded, joint together and glued. This allowed for easy modification and mounting of additional hardware.

Figure 3: Center housing, legs, and camera mount.

Once the frame was constructed, mounts were designed and made for mounting the ultrasonic sensors, blade guards, camera, microcontroller and battery. All brackets were made from Plexiglas that was heated and bent to the specific design needs.

Figure 4: Completed design, with rotor guards.

The brackets for each hardware component were custom made. There were four ultrasonics mounted, one on each arms of the quad copter.

Figure 5: Custom mounts for ultrasonic sensors.

Sensors

Arduino Prototyping

A shield was soldered together to allow for easy connection/disconnection of all other hardware and sensors. Five sets of 3 pin header inputs were used to connect the five ultrasonics located around the craft; four sets were used to connect the four speed controllers to run the four dc motors, and a final 4 pin header was used to interface with the IMU. We also created a small battery monitor circuit that was used to monitor battery power to ensure the batteries were not over depleted and damaged. The three LEDs light up in sequence as the battery drained. The green light was lit when the battery was at full power, the yellow for when the battery was getting low and red when the battery needed to be unplugged before it was damaged.

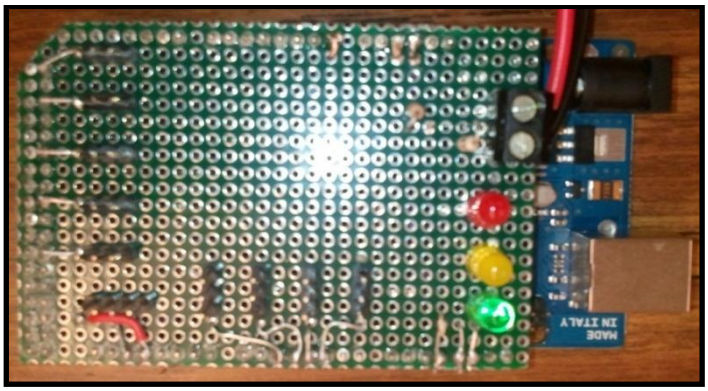

Figure 6: Custom proto-board, showing power monitoring LED's.

Ultrasonic

The ultrasonicsensors were used to monitor the distance of objects from each of the four arms as well as the quad copter's distance from the ground. They work by sending a ping and measuring the time it takes for the ping to bounce off a surface and return to the sensor. Each sensor is pinged in a round-robin fashion. If a sensor does not receive a ping back in a given time period, it times-out to ensure it does not keep counting indefinitely. The five sensors along with the IMU and phones measurement unit were used for autonomous control of the maze environment.

Inertial Measurement Unit

The inertial measurement unit (IMU) device allows for a constant reading of orientation of the aircraft. The IMU used for this purpose is the 3DM-GX1 by Microstrain, which contains gyros, accelerometers and magnetometers providing orientation outputs in matrix, quaternion and Euler formats. Onboard filtering ensures accurate output results as well as gyro drift

compensation. RS-232 is used as the communication format output, which, for the purposes of this project, is level shifted to ±3v, allowing for communication with the Arduino.

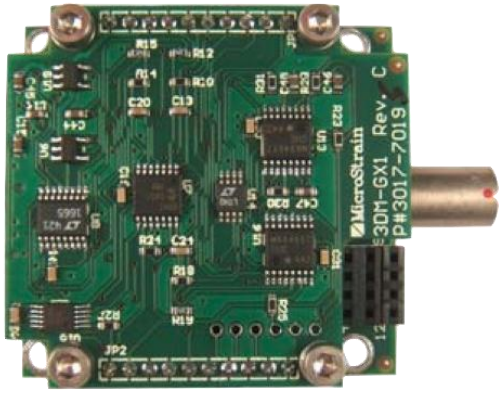

Figure 7: 3DM-GX1 Inertial Measurement Unit.

The 3DM-GX1 has several commands for requesting data in the various formats, as well as for reading and writing to the internal EEPROM, which stores data such as sensor calibration coefficients and digital filter parameters. Each command is one byte in length, and the IMU responds with a fixed number of bytes constituting a data packet. All data packets start with a header byte, equal to the command byte issued, and end with a 16-bit checksum. The intermediate bytes consist of the requested data. A single command is used for the purposes of this project. This command returns an acceleration vector, a drift-compensated angular rate vector, and gyro-stabilized Euler angles. Sending a command byte equal to 0x31 from the Arduino to the IMU triggers this response. The data returned consists of a packet of 23 bytes. The first byte is a mirrored response of the command byte. The remaining bytes are grouped into pairs, with each pair representing the MSB and LSB of the 16-bit requested data. There are six bytes corresponding to each of the acceleration, angular rate, and Euler angle data sets -- two for each axis. The remaining four bytes are the MSBs and LSBs of the timestamp and the checksum. In order to process this data using the Arduino, each byte is read in at a time using the serial library. After verifying that the first byte matches the command byte, the remaining bytes are grouped into pairs. The MSB and LSB bytes of each pair are concatenated into a 16-bit word by using the Arduino *word* command. The checksum is also accumulated at the same time, since it is computed by adding the command byte and all of the 16-bit words. A sample of this procedure is shown below.

```
    //pitch
    MSB = Serial1.read();
    LSB = Serial1.read();
    MSBLSB = word(MSB,LSB);
checks = checks + MSBLSB;
r->pitch = ((float)MSBLSB)*EulerGainScale;
```

Due to the high resolution of the 3DM-GX1, the data elements need to be scaled down to real world values. This is the meaning of the last line of code above. The values for the scaling of each data set is obtained from reading the EEPROM. Since the values in the EEPROM do not change unless the device is recalibrated, these values only need to be determined once, and are then hard coded, as shown below.

```
floatAccelGainScale = 3855.059;
floatGyroGainScale = 3276.8;
floatEulerGainScale = (float)360/(float)65536;
```

All of the data is read in, concatenated, and scaled as shown above, and then stored into a global structure. This allows for the Arduino to request IMU data and then store it for use during the current control loop iteration. This structure is shown below.

```
struct IMU {
    float roll;          /* 16 bit signed Euler roll angle */
    float pitch;         /* 16 bit signed Euler pitch angle */
    float yaw;           /* 16 bit unsigned Euler yaw angle */
    floataccel_x;        /* 16 bit signed acceleration - X axis */
    floataccel_y;        /* 16 bit signed acceleration - Y axis */
    floataccel_z;        /* 16 bit signed acceleration - Z axis */
    floatrate_x;         /* 16 bit signed rotation rate - X axis */
    floatrate_y;         /* 16 bit signed rotation rate - Y axis */
    floatrate_z;         /* 16 bit signed rotation rate - Z axis */
    float ticks;         /* 16 bit 3DM-GX1 internal time stamp */
    intchksum;           /* checksum */
    };
```

The information presented in this section is used to create a function that can be called to read the IMU as needed. This only parameter for this function is the address of the IMU structure. The function performs the sending of the command byte and the receiving of the data packet. The checksum is calculated and compared with the checksum transmitted by the IMU. If the checksum matches, the function returns a 1. If the checksum fails, a -2 is return to signify the error. The error code -1 is reserved for when the first byte in a response from the IMU does not equal the command byte.

Control

The ability for safe and stable control is paramount to any autonomous aircraft system. The control design for the quad rotor aircraft is based on six IMU measurements controlled by four inputs to each individual motor. The concept is that a desired movement can be created by a difference in forces between the motors. Each motor and propeller is defined as capable of producing a static thrust, which is a vector, and can be broken into x- and y-components. Driving the motors is accomplished using a digital output on the Arduino processor board, via Pulse Width Modulation. This signal is fed to the Electronic Speed Controllers (ESC), that are connected to the main battery power, and amplify the signal for the motor. The motors are brushless, and driven by a 3-phase signal output from the ESC. Thus a feedback control loop is not necessary – any desired speed will be maintained automatically by fluctuations in current rather than the PWM. The motors were characterized using a digital tachometer which attached to the tip of the motor. Input angles to the motors were sent, and readings were taken in RPMs. This data was used in MATLABs Curve Fitting Toolbox to determine a stable, third order equation that relates input angle measurements to blade speed.

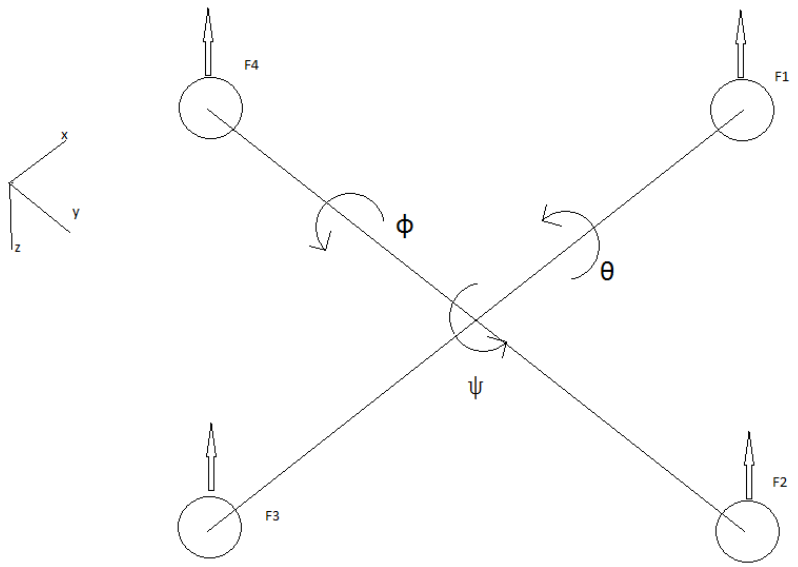

Figure 8: Orientation of the aircraft showing control surfaces.

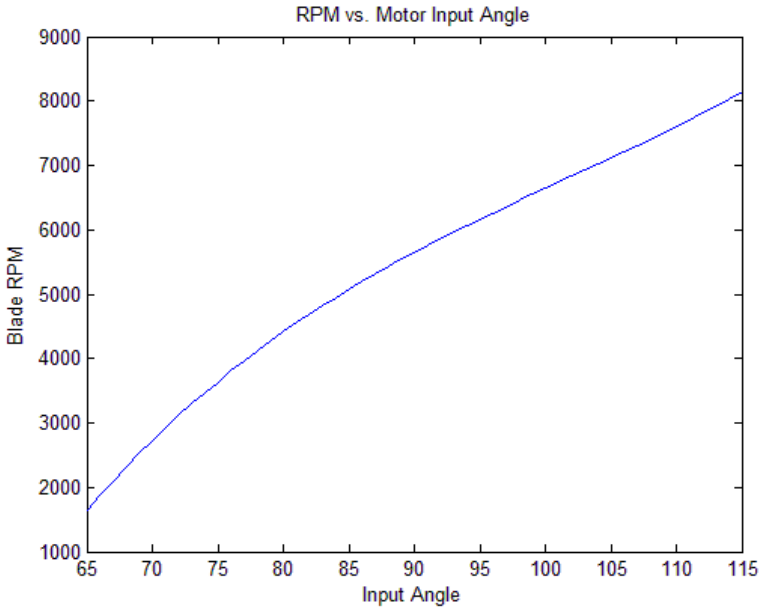

Figure 9: PID control signals to corresponding rotor RPM.

$$\omega = 0.03577*\zeta^3 - 10.87*\zeta^2 + 1195*\zeta - 3.993e4$$

Using the blade speed, knowing the prop is 8”x4.5”, we can use the equation to compute static thrust.

$$F_x = k_1 * \omega_x^2 * d^4 * (\rho/k_2) * C$$

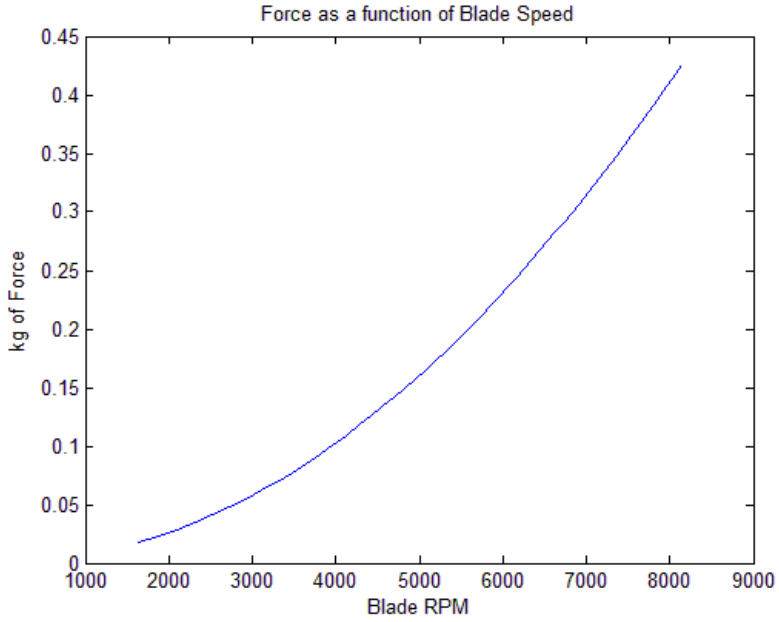

Figure 10: Resulting static thrust from rotors at speed.

The force for each motor is determined as a constant times the blade speed squared, times the diameter d to the fourth power, times the air density divided by a constant k_2 times the correction factor C. The correction factor was determined using known values of the motor from data in a different experiment to adjust for our prop size.

Hovering, increasing, and decreasing altitude are all functions of the downward force of gravity opposed by the z-component of the prop force vectors. With the total weight of the system equaling 1.189 kg, this is the minimum z-vector force required at all times to maintain altitude. IMU data is used to monitor current aircraft orientation, and control signals adjusting blade speeds, and thus force, are used to generate movement.

The control surfaces are as follows:

Roll	θ		
Pitch	ϕ		
Yaw	ψ		
x-Force	x	=	$F_x\cos(90 - \phi)$
y-Force	y	=	$F_x\cos(90 - \theta)$
z-Force	z	=	$F_x\cos(90 - \theta - \phi)$
Torque, θ	τ_θ	=	F_1-F_3
Torque, ϕ	τ_ϕ	=	F_2-F_4
Torque, ψ	τ_ψ	=	$(\omega_1^2+\omega_3^2)-(\omega_2^2+\omega_4^2)$

Because each blade receives a control signal, four separate PID parameters are used for error correction. The desired orientations and movements are input as setpoints, where the corresponding force of each motor can be calculated. This force is used in the static thrust equation to solve for omega, which is then solved for a corresponding output angle. The usages of these angles are needed to work within the existing PID software library for Arduino microcontrollers. The current control design is in this phase. As the model progresses, it can be empirically tested on the aircraft. Future work will look to implement inertial moments in the computations to give better control over the craft, and predictive models will be used to coordinate with the Android maze solver and movement generator to give locational awareness to the UAV.

Maze Solving

In the context of autonomous aerial and land vehicles, path planning is a challenging problem in unknown and semi-known environments. Emulating a possible scenario from the Drexel challenge rules, the problem we will consider is a 5x5 maze with a single origin and a single goal. Multiple paths can be taken from the origin to the goal. The agent traversing the maze has previous knowledge of the maze's configuration; however after some shock to the environment, some of the maze spaces will present a hardship for a robot to traverse.Even with general knowledge of the shock event location, the agent can only discover these spaces by exploration. From the challengedue to non-constant environments, an agent may have to reconsider its path to the goal. However, as the agent is traveling through the maze the first shortcut may not necessarily be the best one, and it may be more beneficial to not take this path and keep on course until a better one shows up. To quantify the quality of the paths, a reward system is considered, and since this problem involves a maze, we plan to use the maze traversal algorithm A* (A-star) with a Manhattan heuristic to plan the paths. In our preliminary research, we came across papers that deal with similar exploration problems.

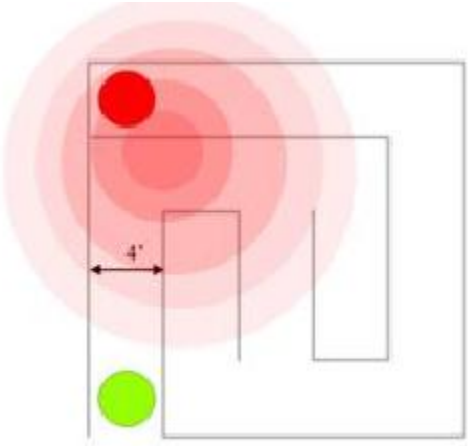

Figure 11: A floor plan of a maze that has been affected by a damaging event that may change our movement patterns in the affected area.

A* Search Algorithm and Manhattan Heuristic

Optimal path planning begins by acknowledging the starting location, the green square, and adds it to the "open list." It then looks for the lowest cost square described by the equation

$$F = G + H$$

Where G is the movement cost to move from the starting point A to a given square on the grid, following the path generated to get there.H is the estimated movement cost to move from that given square on the grid to the final destination, point B.This lowest costing square is referred to as the "current square" and is switched to the "closed list". Each of the adjacent 8 squares from the current square are analyzed by determining which path to each of the squares is better. The current square becomes the parent and the path each square is checked. If the square is not walkable or if a location is already on the "closed list", it is ignored. If a square is walkable, and not on the "open list", it is added. The F, G, and H costs of the square are determined, and if already on the open list, it is checked using the G cost, to see if the path to that square is better. A better path is determined with a lower G cost. If it is deemed better, the parent of the square is changed to the current square, and the G and F scores of the square are recalculated. The process stops when the target square is added to the closed list. Now to find the optimal path, the method works backwards from the target square going from each square to its parent square, shown as red dots, until the starting square is reached. Though A* Search does find the optimal path, it does not account for environmental statistics including potential obstacles, shortcuts, or errors that the hardware solving the maze may experience in certain environments. Sensor systems are implemented on aircraft to detect these, and Markov Decision Process takes into account some of these aspects. This is a prioritized task for future project work.

Figure 12: With the Manhattan Heuristic, the agent knows the destination coordinates, but not obstacles in its way and determines its distance in horizontal and vertical spaces like city blocks.

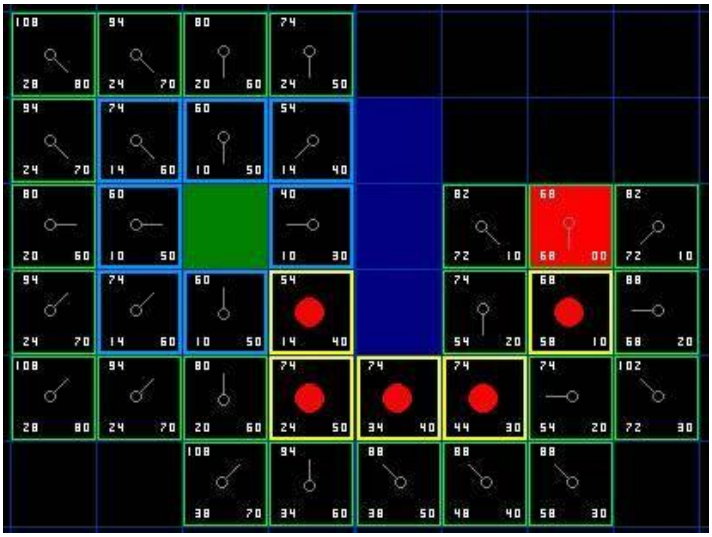

Figure 13: The progress of an agent through a Manhattan grid space using the A* algorithm with the Manhattan heuristic

Android Implementation

The Android platform is at the center of our portable maze solver and system navigator. In order to develop software applications for the mobile operating system, we need to understand how applications must be structured. With an Android system, applications are composed of *activities*. They are components of the complete application, and provide processes for user interaction. Most applications can consist of a number of activities. These activities are in turn composed of smaller building blocks called views. Views host components for user interfaces such as buttons, text fields, and graphics. The application starts with a main activity and activities can call other activities at which point it is important to consider the activity life cycle. Activities have states that include *start, resume, pause, stop,* and *destroy*. The start state refers to when an activity is about to become visible. The resume state is called when the activity has become visible. Thepause state is called when another activity is about to take focus from the activity. The stop state refers to when an activity is no longer visible, and the destroy state refers to when an activity is about to be removed from memory. Each activity can have each of these states implemented. When an activity calls another activity, it executes the pause state which saves the current state of the activity.

New activities often need data from other activities that are becoming inactive. This information must be passed to the new activity as soon as it is activated or the data become inaccessible. This method is sometimes inappropriate for applications that require continuous access to data. A static service can provide a way to pass data between activities. When one is created, all activities are able to access information it provides during the lifetime of the application. These activities however must inherit the needed functions to have control over the service.In order to initialize a maze map in the Android application, we must come up with an efficient way to either create an intuitive user interface or an off board application that can then dispatch instructions and transfer the maze map. To make the UAV more self-sufficient, we chose the Java application method hosted on a laptop pc with local network communication to the Android device on the aircraft.

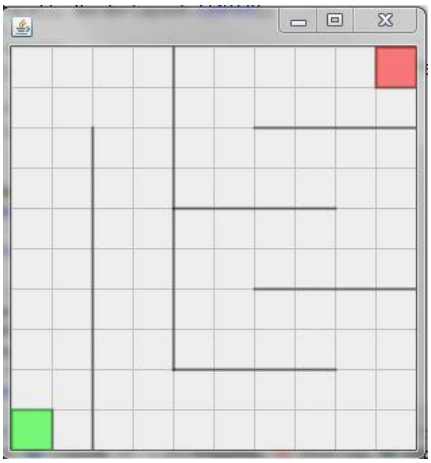

Figure 14: The PC Java application shown became the user interface for the aircraft.

There an operator was able to design the maze map using simple click and drag methods to draw a maze on a grid. Using a “Send” button, a *serializable* class containing an NxN maze map, start and stop positions and message was passed to the Android device over a network socket object stream. On the phone, an *Async* task hosted a socket listener that *deserialized* the received object. When *deserialized*, the map was parsed and sent to a maze solver service in the phone application. When solved, the android device was to communicate with the aircrafts’ IMU to determine its position within the maze map. With the position, paths could be planned to get the UAV to its destination.

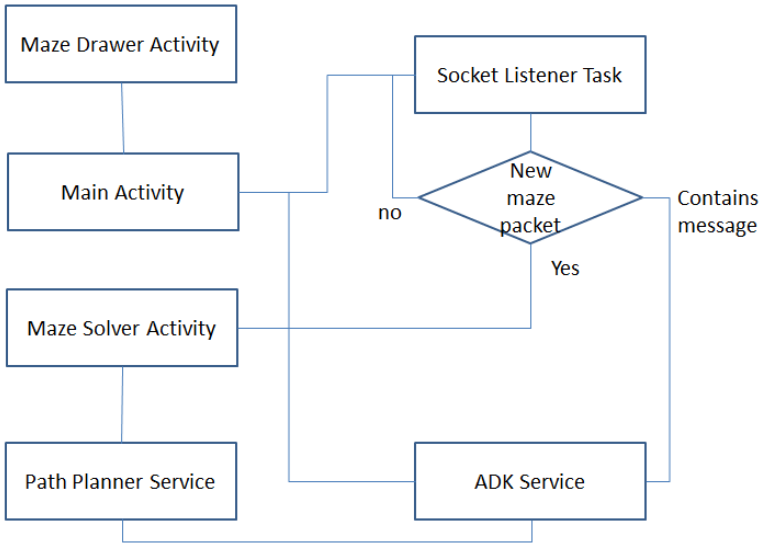

Figure 15: Application structure flowchart.

In the main activity, a socket listener task and a service to send instructions to the Android Development Kit were created. The main activity also hosted the maze drawer view which displayed the maze along with a planned path. This is mostly used for debugging purposes since the device is on board the UAV. When a new maze packet object was received in the object stream of the socket, the maze map was created and the maze solver activity was started. The maze solver activity creates the path planner service which directly interacts with the ADK service to communicate movement instructions and handle localization. When the case is such that the packet received contains a message, this message is interpreted as a movement command which is directly handled by the ADK service in the application.

Conclusion

The autonomous maze solving aircraft is still in development. A large amount of work remains for the team, which is outlined in many of the sections above. However, the work completed thus far yields an excellent platform for this to take place. Important tasks remain specifically in areas of motor control and trajectory, locational awareness and hazard detection, and movement planning. Working towards a fully function UAV remains the goal, with practical experience and a more expansive robotics laboratory at Temple University as subsidiaries. The team would like to graciously thank Dr. John Helferty, Temple University, for his support, assistance, and belief in the project and the team. Acknowledgement is owed as well to NASA for funding student research projects such as this.

References

[1] http://publications.asl.ethz.ch/files/bouabdallah07design.pdf

[2]http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2008/rapporter08/sikiric_vedran_08027.pdf

[3] Hongshe Dang; Jinguo Song; Qin Guo; , "An Efficient Algorithm for Robot Maze-Solving," *Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2010 2nd International Conference on* , vol.2, no., pp.79-82, 26-28 Aug. 2010

[4] Sutherland, I.E.; , "A Method for Solving Arbitrary-Wall Mazes by Computer," *Computers, IEEE Transactions on* , vol.C-18, no.12, pp. 1092- 1097, Dec. 1969

[5] http://robotics.ece.drexel.edu/events/iarc/wp-content/uploads/2011/09/IARC-2012-v2.pdf