The Maneuvering Board, MoBoard for short, is a tool used by sailors to track the motion of vessels in the water. That information can be used in a slew of different ways, from determining if risk of collision exists to helping two vessels station next to each other while in motion. As such, every U.S. Naval officer in training must be proficient in the many MoBoard techniques used daily in naval operations.

The most basic MoBoard problem takes the following form:

1. You know your own speed and direction in the water relative to the earth, which we call absolute motion.
2. By taking contacts over time, you can determine the speed and direction of other vessels underway, relative to your movement.
3. Adding the relative motion of a target vessel to your absolute motion yields the absolute speed and direction of the target vessel.
4. Of the many pieces of information that can now be solved for, the most important is the Closest Point of Approach (CPA). The CPA bearing, range, and time describe the time and place that your two ships will be closest in the water.

While many concepts in naval training can be taught in lecture or seminar type learning environments, the only way to learn maneuvering board techniques is through repetition. The goal of my project is to create a learning aid that walks students through the solution to MoBoard problems without the individual attention of an instructor. With a tool that outlines each step of the process, students can get the individual attention and repetition they need to become proficient in MoBoard techniques.

This learning aid has massive user potential, mainly within naval training units. Before development started, I polled the market for demand. Everyone I talked to, from midshipmen at the Naval Academy and my ROTC unit, to Naval Officers with more than 20 years of experience all thought very highly of this project. With successful implementation of a solution, I believe this app can be a required resource at every U.S. naval training station.

Part 2: Motivation and Related Work

Currently, maneuvering boards are taught first-hand by teachers, or must be learned from instruction manuals and static PowerPoints and YouTube videos. The different methods can be viewed at the following links:
Sans teacher, there is no dynamic method of teaching this material. Every aid demonstrates a specific example, but cannot adapt to do other problems. With so many unique situations that could arise in the solutions to these problems, having a static learning aid in not sufficient. By creating an aid that takes student supplied problems and then walks them through the solution, students could pick up this material much faster and more reliably than in the past.

Additionally, my solution to this problem is appealing to teachers. Instead of having students crowd around a single MoBoard, teachers could project this app and add their own commentary as the program walks them through the solution.

**Part 3: Design of Solution**

I wrote the app in Python, using the wxPython extension to develop the graphical interface. I chose the Python / wxPython combination because I was already familiar with Python and the wxPython GUI package was the simplest one that allowed me to develop a proof of concept model. Learning wxPython was heavily dependent on the previous knowledge of classes and hierarchal design.

The app runs in two modes, answer and tutorial. Answer mode is straightforward: it takes user input and shoots out answers. This mode is intended for students to use to check their work. Tutorial mode is where a bulk of the programming and ingenuity of this project occurred. When the user enters data into tutorial mode, the solution is broken into the following nine steps:

1. Determine Range and Speed Scales
2. Plot Own Ship At Center of MoBoard
3. Plot M1 and M2
4. Extend Target's Course of Relative Motion and Determine DRM
5. Calculate Target's SRM
6. Plot Target's Relative Motion
7. Determine Target's Actual Motion
8. Determine CPA Bearing and Range
9. Determine CPA Time

Each step has an explanation on the left side of the app and an accompanying graphic on the right side. The explanation is meant to help the student understand what to do, but also to
help them understand what is happening in the physical space and what each mark on the plot means.

The actual coding of the app was broken into three parts. The first was setting up the graphical interface. This included making sure the different panels worked in unison, linking up buttons to action items, and making sure the graphic was formatted properly. The second was figuring out the math. MoBoards by design are meant to be extremely graphical and limit the need for any calculations. However, this isn’t all that helpful when trying to solve them digitally. Each of the nine steps listed above needed to be taken from the graphical space to the digital space, and stored as a system of equations. Once the equations were checked and GUI was working, the final step was to plot all the points on the MoBoard and to write up the dynamic explanations that change with each different submitted problem.

**Part 4: Technical Problems to Overcome**

I overcame three major technical problems while developing this app. The first was correctly representing the MoBoard in the digital space. This had a number of sub-problems, including but not limited to:

1. Rotating bearing input 90° before storing into variables to account for 0° pointing due north on MoBoards, but due east in Cartesian coordinates. All bearing answers then had to be rotated back 90° before displaying answer.
2. Multiple parts of the MoBoard solution require the user to “add or subtract 90°” or “choose the correct answer.” While these processes are easy for humans looking at the physical board, coming up with algorithms to consistently discern the correct answer was challenging.
3. Solving general forms of certain equations in order to reuse them in multiple parts of the program. Solving the general form for the intersection between a line and a circle proved to be exceedingly challenging, especially since the program had to correctly decide which of the two solutions was correct.

The second was figuring out how to correctly draw on the MoBoard graphic. I stored a lot of coordinates internally in the program, like the center of the board, top and bottom of each scale, and the nomogram. I also sampled and stored a lot of ratios, like the certain number of pixels per yard or knot for the different scales, and created log functions to help determine where points were in the nomogram. Every point plotted needed to be done so relative to these coordinates so that the graphic would look clear to the user.

The third was trivial, but also the hardest to get working properly. As the graphics were the main learning objective for me during this project, getting the layout and functionality that I wanted took a lot of trial and error testing. Linking up the panels so that certain graphics appear on the correct button clicks was very challenging. Once I had the GUI working the way I wanted it to, laying everything else into it happened quickly.
Part 5: Description of Underlying Mathematics

The math involved in finalizing this project was extensive. If you parse the code, most of the math is correction factors that allow the program to function in all four quadrants and solve for vessels moving in any direction, including math that accounts for edge cases such as divisions by 0. Including that math in this section would be unnecessary, as the core equations are meant to describe the main functionality, not account for all edge cases.

We require 8 pieces of input from the user, defined below:

1. Course: true course of your vessel
2. Speed: speed of your vessel
3. T1: time at which the first reading is taken
4. B1: true bearing of the contact at T1
5. R1: range of the contact at T1
6. T2: time at which the second reading is taken
7. B2: true bearing of the contact at T2
8. R2: range of the contact at T2

Step 1: Plot ER vector onto MoBoard
ER vector is a line from the origin to the R coordinates, which can be solved by:

\[ RX = \text{Speed} \times \cos(\text{Course}) \]
\[ RY = \text{Speed} \times \sin(\text{Course}) \]

Step 2: Plot M1 and M2
M1 and M2 are the two readings acquired off the target ship. Solve for the location of each point to plot using:

\[ M1X = R1 \times \cos(B1) \]
\[ M1Y = R1 \times \sin(B1) \]
\[ M2X = R2 \times \cos(B2) \]
\[ M2Y = R2 \times \sin(B2) \]

Step 3: Solve for Direction of Relative Motion (DRM)
The line through M1 and M2 is the line of relative motion. Solve for the DRM by first getting the slope of the line through M1 and M2, then taking the tangent to get degrees.

\[ DRM_{\text{slope}} = \frac{(M2Y - M1Y)}{(M2X - M1X)} \]
\[ DRM_{\text{degrees}} = \arctan(DRM_{\text{slope}}) \]

Step 4: Solve for Speed of Relative Motion (SRM)
While the SRM is solved graphically using the nomogram, the app must solve for it internally by dividing the distance between M1 and M2 by the time it took to cover that distance. That number must then be converted from yards/minute to knots.
SRM = \frac{\sqrt{(M2Y - M1Y)^2 + (M2X - M1X)^2}}{T2 - T1 \text{ minutes}} \text{ yards} \times \frac{1 \text{ knot}}{33.756 \text{ yards/minute}}

Step 5: Plot Target’s Relative Motion
With the speed and direction of relative motion known, the app plots the second leg of the speed triangle, the RM vector. Starting at the previously determined R point, the app plots the M point based on rise and run of the DRM and SRM, then connects the points.

\[ MX = RX + (SRM \times \cos(DRM)) \quad MY = RY + (SRM \times \sin(DRM)) \]

Step 6: Plot and Solve for Target’s Actual Speed and Bearing
Graphically, the point between E and M represents the target’s actual motion through the water, relative to the earth. Internally, the app solves for these two data points using the following equations:

\[ \text{Contact Course} = \arctan \frac{MY}{MX} \]

\[ \text{Contact Speed} = \sqrt{(MX)^2 + (MY)^2} \]

Step 7: Plot Closest Point of Approach (CPA)
The CPA is the intersection between the line of relative motion and a perpendicular line that extends from the origin. Solve for the coordinates of the CPA by solving for the intersection.

\[ \text{CPA slope} = \frac{-1}{\text{DRM slope}} \]

\[ CPA_X = \frac{M2Y - \text{DRM slope} \times M2X}{\text{CPA slope} - \text{DRM slope}} \]

\[ CPA_Y = \text{CPA slope} \times CPA_X \]

Step 8: Solve for Bearing, Range, and Time of CPA
Once the coordinates of the CPA are known, the app solves for the bearing, range, and time of the CPA using the following equations:

\[ \text{CPA bearing} = \arctan(\text{CPA slope}) \]

\[ \text{CPA range} = \sqrt{(CPA_X)^2 + (CPA_Y)^2} \]

\[ \text{CPA time} = T2 + \frac{\sqrt{(CPA_X - M2X)^2 + (CPA_Y - M2Y)^2}}{SRM} \]
Additional Notable Equations
In addition to the equations used above, a couple other equations were used on the graphics side to keep the plot looking clean. The most important solved the general form of the intersection between any line, defined by a point X1, Y1 and slope M, and the edge of the circular plot. With a fixed center at pixel (299, 269) and radius of 231 pixels, the general form allows the app to extend lines from the origin to the edge of the plot at point (X, Y) neatly without going over or coming up too short.

\[ Z = (-M * X1) + Y1 \]
\[ A = 1 + M^2 \]
\[ B = (2 * Z * M) - 598 + (538 * M) \]
\[ C = Z^2 + (538 * Z) + 108401 \]
\[ X = \frac{-B + \sqrt{B^2 - (4 * A * C)}}{2 * A} \]
\[ Y = M * X + Z \]

Part 6: Screenshots of the System

Answer Mode
Answer mode is the simpler of the two user interfaces. It takes the 8 inputs, stores them as variables, the solves for the 7 different pieces of information when the user presses “Enter.” Answer mode is meant for students to check their work after they have already solved the problem.

Tutorial Mode
Tutorial mode is the primary feature of the app. It takes the 8 user inputs, then walks the student through the solution in 9 different steps. As it solves the same 7 pieces of information from answer mode, it states them explicitly at the bottom of the app.
Through talking with Navy personnel and showing my prototype to students and officers, I strongly believe that this app has potential for major development and proceeding distribution within the Navy. Of immediate improvements to the app, I think that the three most important are:

1. Making the MoBoard graphic scalable: Right now, the graphic is fixed at 600x619 pixels, which made it realistic to get the drawings on the board in the time that I had. People I have talked to want to get this up on a projector, but the resolution won’t improve until an algorithm to scale the graphic correctly is added.
2. Dramatically increase error handling: I built almost no error handling into this system. If anyone tries to break or stall the app, they will succeed.

3. Add more functionality: So many different problems can be solved using a MoBoard, but my program only addresses a couple of them. The system is in place, so adding more functions would be not be terribly difficult.

I plan to pursue a copyright on the intellectual property, then allow this product to go out to the Navy in a rather infant stage. There isn’t much money to be made from this platform, but it’s a tool that all training stations need, and one that could be dramatically improved if programming was crowd-sourced out to anyone that was interested.

Part 8: Glossary of Navy Technical Terms

- **Contact:** Any ship that has been detected, either by line of sight or electronically
- **CPA:** Closest Point of Approach, the closest a contact will ever be to your ship
- **DRM:** Direction of Relative Motion, the motion of a contact relative to your ship
- **EM Vector:** Graphical representation of a contact’s absolute motion, relative to earth
- **ER Vector:** Graphical representation of your absolute motion, relative to earth
- **GUI:** Graphical User Interface, any interface that a user interacts with
- **M1 and M2:** The location of a ship at two different points in time, relative to your ship
- **MoBoard:** Maneuvering Board, a polar plot used to graph the motion of ships
- **RM Vector:** Graphical representation of a contact’s relative motion, relative to you
- **ROTC:** Reserve Officers Training Core, a four year commissioning program
- **SRM:** Speed of Relative Motion, the speed of a contact relative to your ship

Part 9: References

All the references for this project are links that I used to learn how to code wxPython. I solved the method on my own, and already knew how to program in Python. The links I used to learn wxPython can be broken into two groups, tutorials and forums that discuss the solutions to specific coding related problems such as Stack Overflow.

**Tutorials**
wxPyWiki: [http://wiki.wxpython.org/AnotherTutorial](http://wiki.wxpython.org/AnotherTutorial)

**Stack overflow peer help links:**
Drawing onto Bitmaps: [http://stackoverflow.com/questions/20287152/wxpython-draw-text-onto-existing-bitmap-or-image](http://stackoverflow.com/questions/20287152/wxpython-draw-text-onto-existing-bitmap-or-image)
Mouse vs Python
Event Handlers: http://www.blog.pythonlibrary.org/2012/07/24/wxpython-how-to-fire-multiple-event-handlers/