Using a Virtual Robot Model – Testing and Debugging Autonomous Code Through Simulation

By Chris Hibner
Team 51 – GM Powertrain and Pontiac High School: Wings of Fire
Rev: 2/10/2010

Background

One of the most difficult challenges in the FIRST Robotics Competition is having a highly capable and properly functioning autonomous routine prior to shipping the robot. There are many things that make this a difficult challenge, such as:

- The robot is not finished until late in the build season, leaving inadequate time for testing and debugging.
- Conflicts within team subgroups for time with the robot (e.g. mechanical tweaking, driver practice, etc.)
- Bugs in the autonomous code may cause the robot to crash and cause damage.
- Bugs in the autonomous code can be difficult to track down, especially with complex autonomous code using feedback.

The difficulties listed above aren't confined to FIRST robotics. Similar difficulties exist throughout industry, especially when schedules are tight and hardware is expensive or hard to come by. Because of these difficulties, engineers often rely on simulations.

Summary

In order to fully exercise and debug autonomous code, a simple yet effective robot model was created. It was decided that the model should be capable of the following:

- Inputs to the model should include the software outputs, such as drive motor PWM levels.
- The model should simulate the dynamics of the robot reasonably well (although perfection is not necessary).

- Outputs of the model should be what are used as feedback by the autonomous code. Examples include: robot heading, distance traveled, encoder outputs, etc.
- The model should also record and output the path driven by the robot, so that expected behavior can be confirmed in a simple, visual manner.

What is the model good for?

This model is excellent for performing the following testing:

- Software logic. The model is an excellent way of testing many paths and scenarios that your autonomous code may encounter. You can test all paths quickly and easily, and without fear of crashing the robot.
- Calculations. When autonomous code becomes complex, it is common to have errors in calculations. Common errors include using the wrong sign, using the wrong signal, using an incorrect constant, etc. The simulation environment should allow you to discover these errors.
- Determining unforeseen problems in your code. Whether you call them bugs or "undocumented features", it's nice to find and remove them before getting 130 lb of robot travelling at 10 ft/s.
- Ballpark estimates of feedback gains. The model has a basic representation of the dynamics of the robot. Because of this you can get a ballpark estimate of gains. However, the dynamic model isn't really that great, so final tuning must be done on the robot.
- Generating new autonomous paths. Since the model draws the path that the robot follows, obvious errors in your path can be determined before testing on the robot. I've seen many times in the past that the robot turned left when you wanted it to go right, all due to human error in entering the path. By using the model and plotting the path, you can eliminate these errors without ruining a match.

What are the limitations of the model?

- This model works for tank-style steering only. Any other steering systems would require major changes. That doesn't mean you should stop reading. You can still use the framework and tweak the model.
- This model doesn't account for sensor errors. The model assumes the sensors are perfect, which is great for testing logic in the code – but it doesn't show you what happens if a sensor starts to drift.
- The model cannot be used for fine tuning of control gains. However, in order to get a reasonable simulation, I would still suggest tuning your gains for use with the model. Otherwise you may not exercise your code the way

you think you should. However, don't get too hung up in making your feedback behave perfectly.

- The model doesn't account for robot imperfections. If you apply full PWM to both sides, the model will show your robot going perfectly straight (which we all know rarely happens). This can easily be changed by scaling one of the PWMs by some constant, such as 0.95 to simulate extra drag on that side of the drive train. I'll leave that for you to add to the simulation, if you want to.
- The physics aren't very well represented. The dynamic character of the robot is modeled well enough to exercise the autonomous code. Don't expect perfect correlation to your robot – just expect to exercise your code.

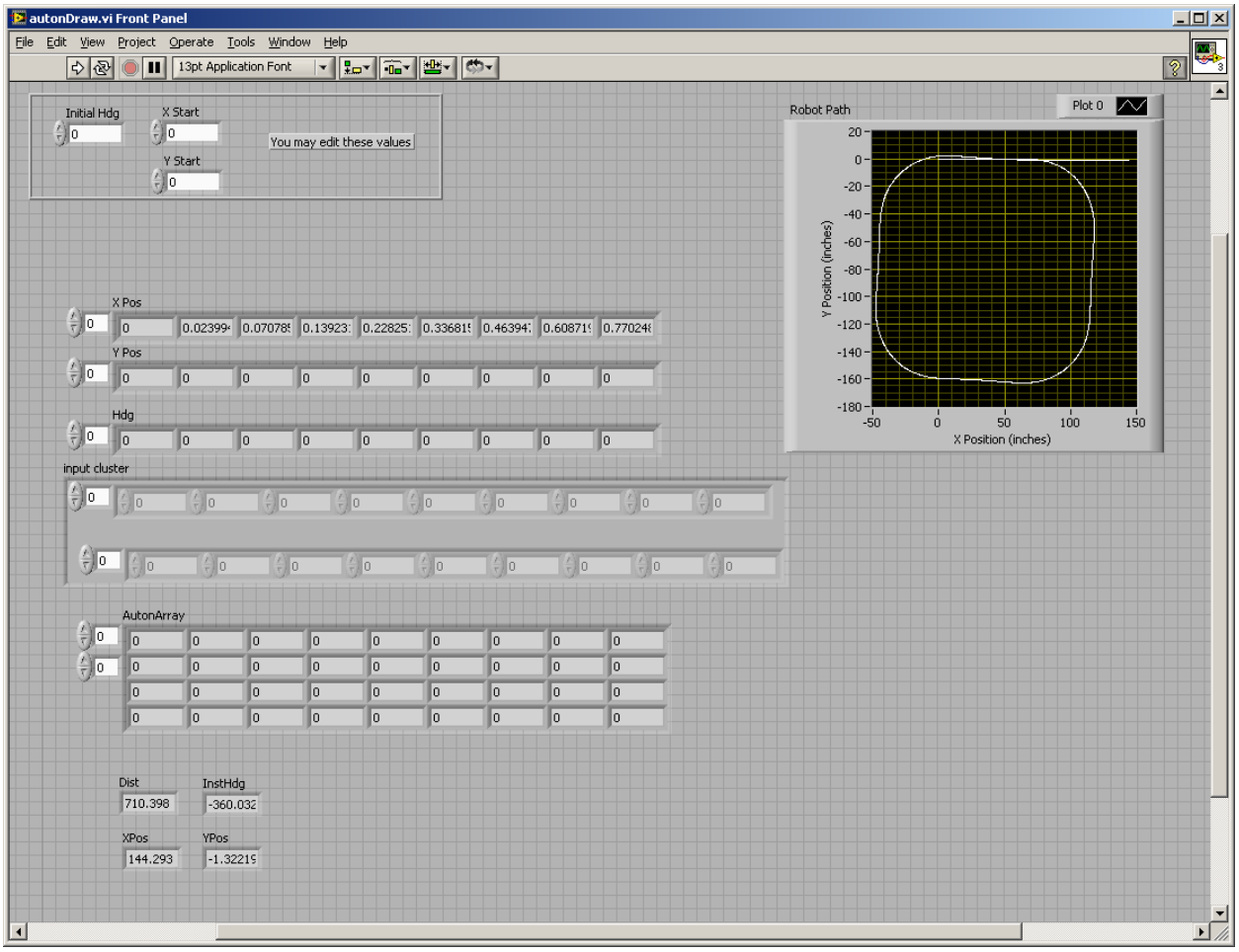
Details

Modeling Method

I don't want to go into great depth discussing how the dynamics of a robot were modeled. It's not that important for testing your code. If people show a lot of interest, I'll revise this section in the future. In one simple statement, the robot dynamics are modeled as two interacting first order systems. If you know what that means, then that's great. If not, don't worry about it.

Simulation Interface

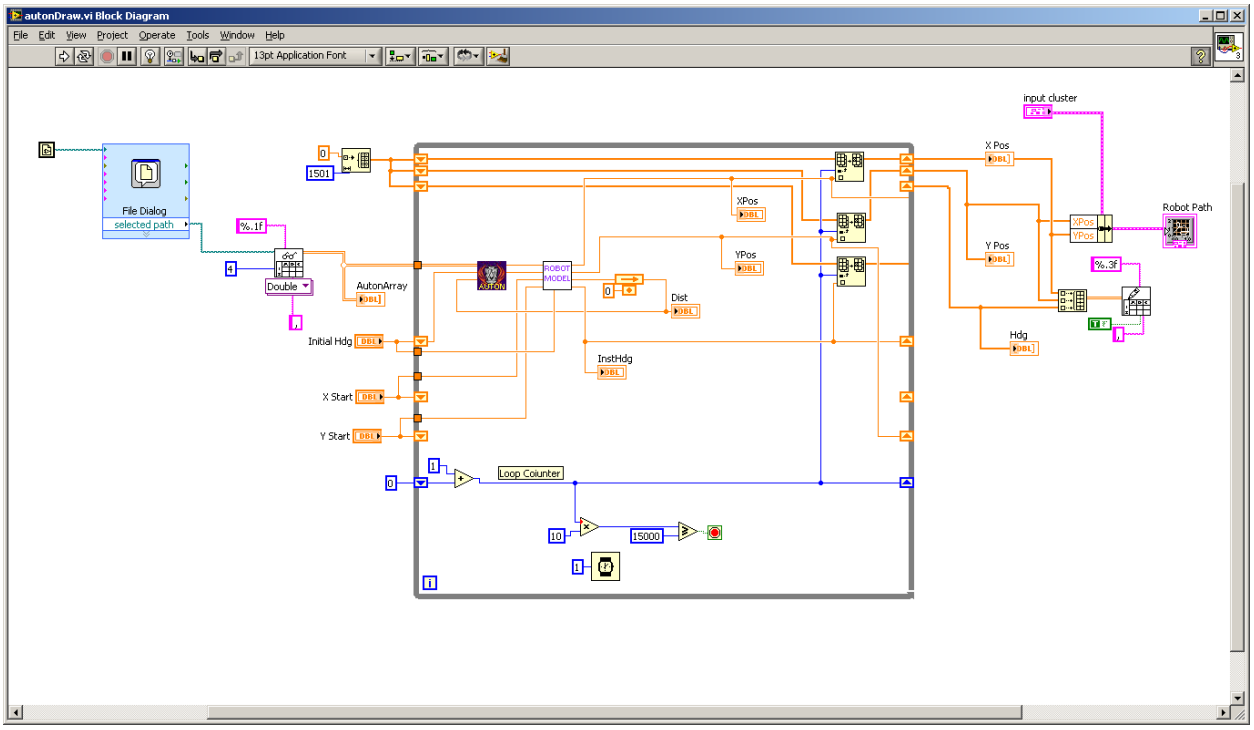
The robot model is embedded in a simulation interface. The simulation interface is a While Loop that simulates calculating the robot position every 10 ms, and ends after 15 simulated seconds. Once the simulation ends, data (X position, Y position, and Heading) is stored to a .csv file which can be opened and analyzed in Excel or any other data analysis package. The X,Y position is also plotted on the front panel. The front panel of the simulation interface is shown below.



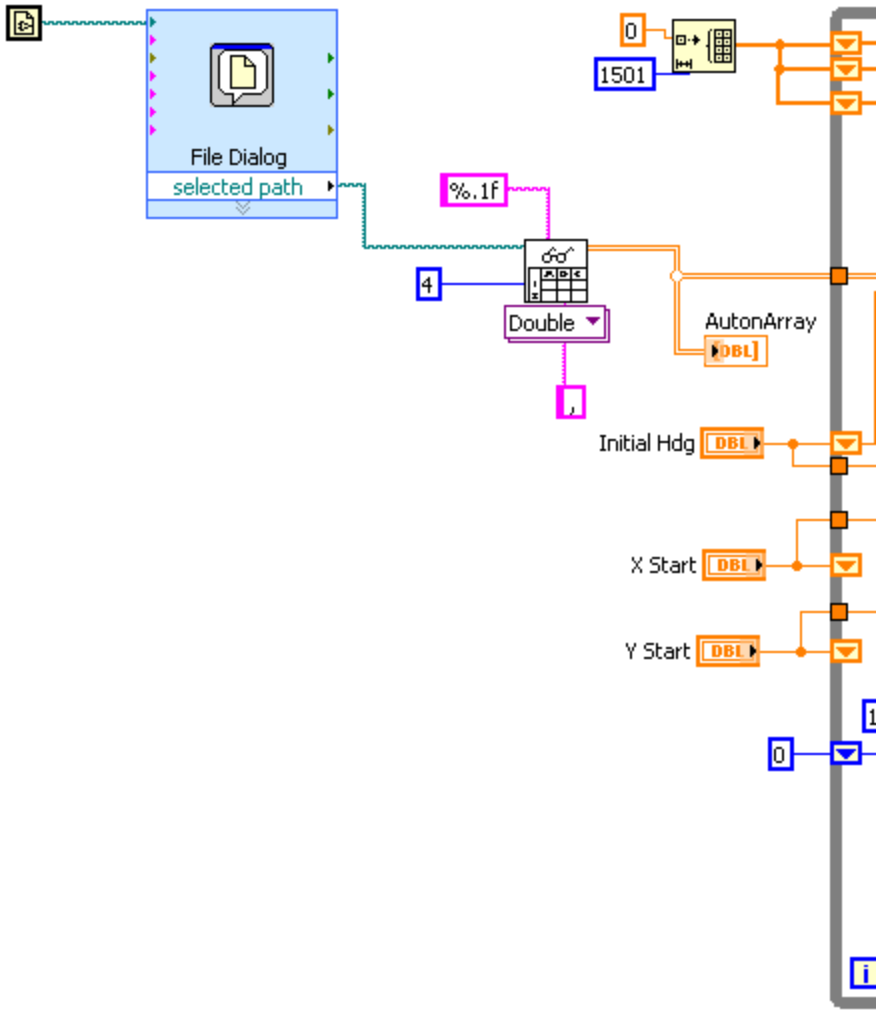
In the simulation interface, you can edit the starting X and Y position of the robot, as well as the starting heading of the robot. I usually leave these at 0, but feel free to change them if you want to. At the end of the simulation, the path of the robot is plotted in the XY graph, and position and heading data are shown in the array indicators. You can feel free to change the front panel to suit the needs of your robot. The "AutonArray" is something specific for Team 51's robot, so don't worry about that for now – I'll explain what it is in the next section.

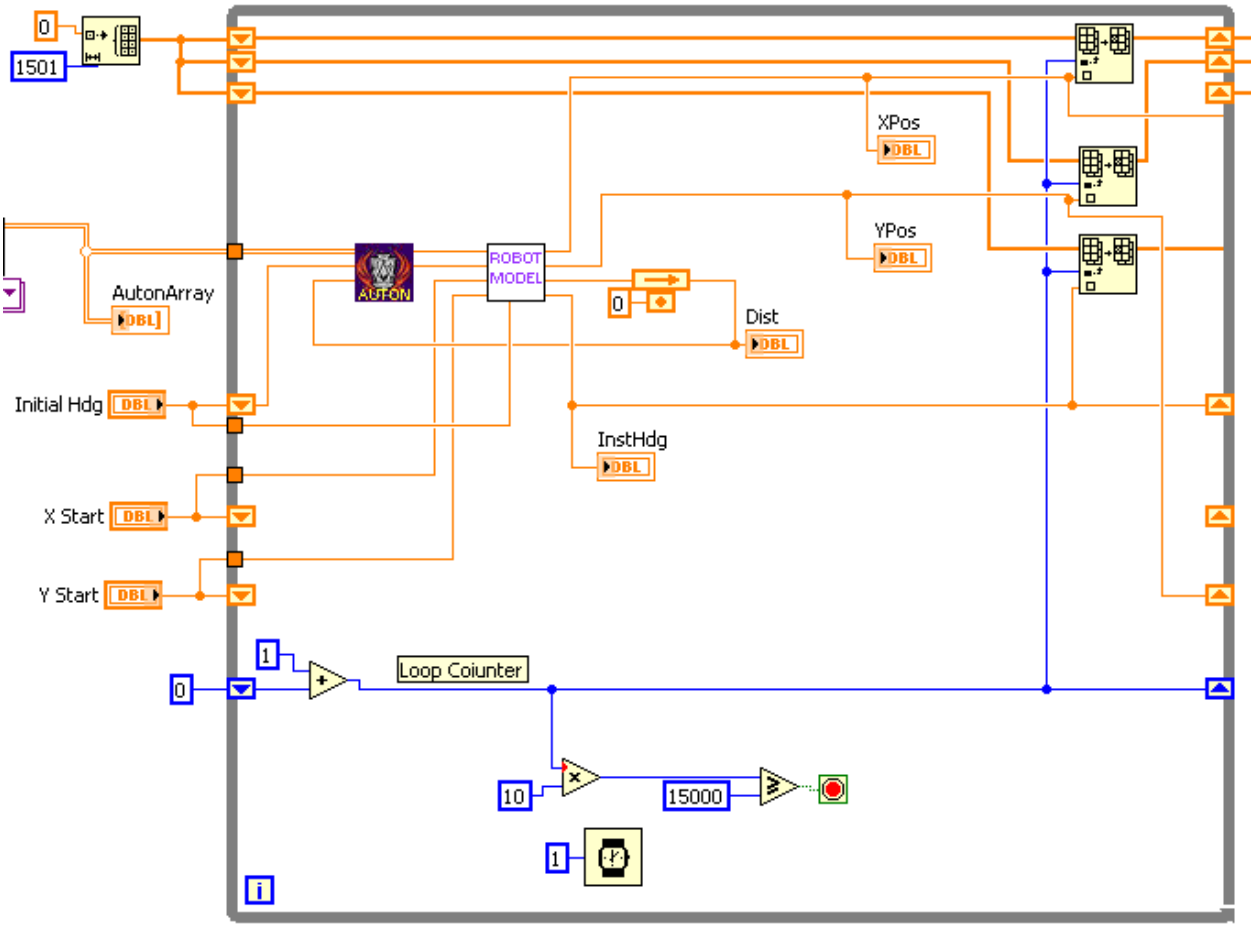
Simulation Block Diagram

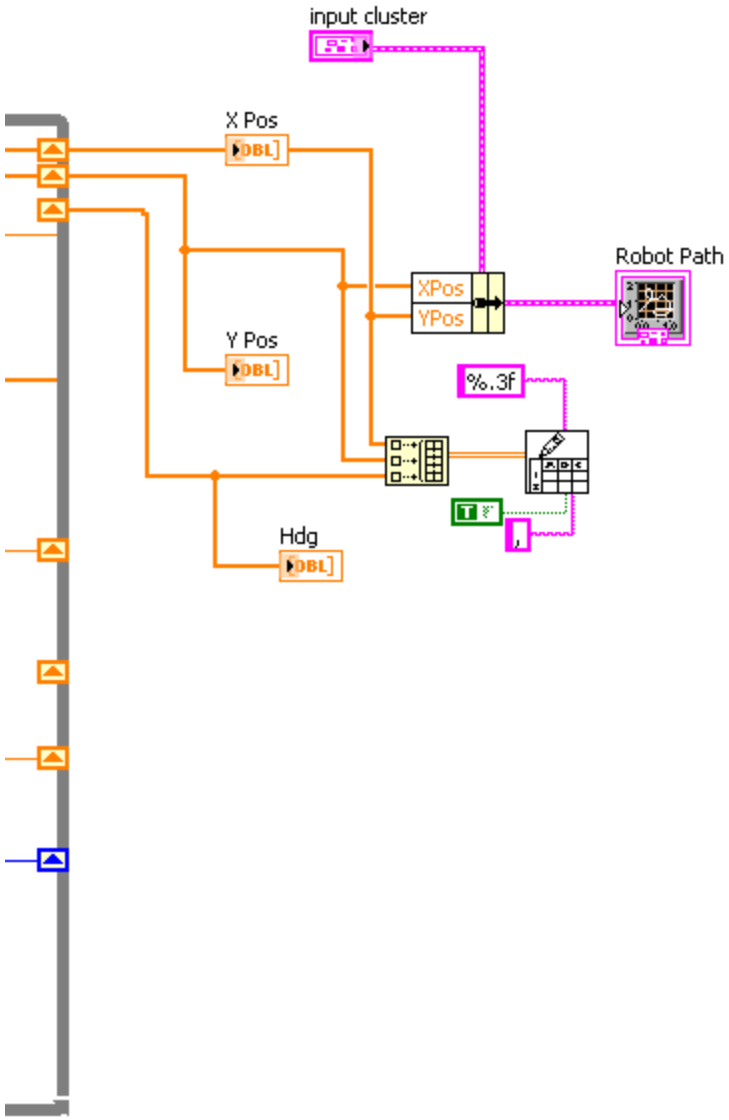
The simulation block diagram is shown below.



Since that is a bit difficult to see, I cut it up into three sections which are shown below.


File Dialog
selected path
%.1f
4
Double
,
0
1501
AutonArray
DBL]
Initial Hdg
DBL
X Start
DBL
Y Start
DBL
0
i


0
1501
XPos
DBL
YPos
DBL
AutonArray
DBL]
AUTON
ROBOT
MODEL
0
Dist
DBL
Initial Hdg
DBL
InstHdg
DBL
X Start
DBL
Y Start
DBL
1
0
Loop Coiunter
10
15000
1
i



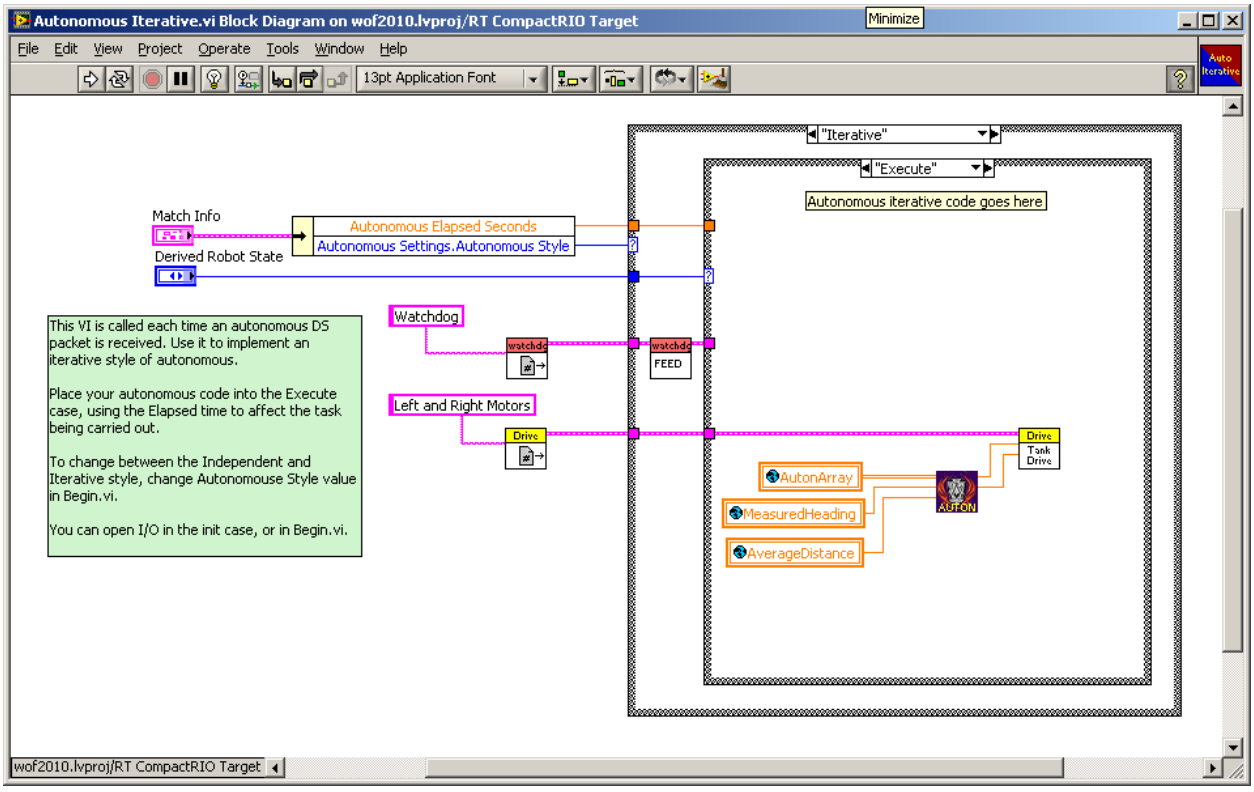
The autonomous code for our team is shown here in a sub VI called AUTON, which is not included in the supplied code (sorry). The supplied simulation code includes a simple autonomous routine that drives in a circle. You can run the simulation with this simple autonomous routine just to get a feel for how the simulation works.

The "AutonArray" that you see is how we tell our autonomous code what we want to do for that match. Note that you also see "AutonArray" on the front panel. You will need to modify this block diagram and front panel to work with your autonomous mode and how your team selects autonomous modes (like switch inputs or something like that).

Note: the While Loop is shown with a Wait of 1 ms. You may set this Wait to whatever you want and it will not change the results of the simulation. The simulation uses 10 ms as the simulated time step for integral solving, so if you want to watch the model run in real-time, wire a 10 into the Wait and the simulation will then take 15 seconds to run (like I said – real time). If you want to watch things in slow motion, wire in a larger number – just be aware that the simulation will take longer to complete. If you want to do some fast number crunching and debugging, leave the Wait at 1 (or remove the Wait altogether).

Recommendations for Using the Simulation

I highly recommend that you create a SubVI out of your autonomous code and then use this SubVI in the simulation. This does NOT mean you should copy the "AutonomousEnabled" VI from the robot framework. What I mean is that you should create a SubVI *inside* the AutonomousEnabled VI. See the figure below to see how we did it.

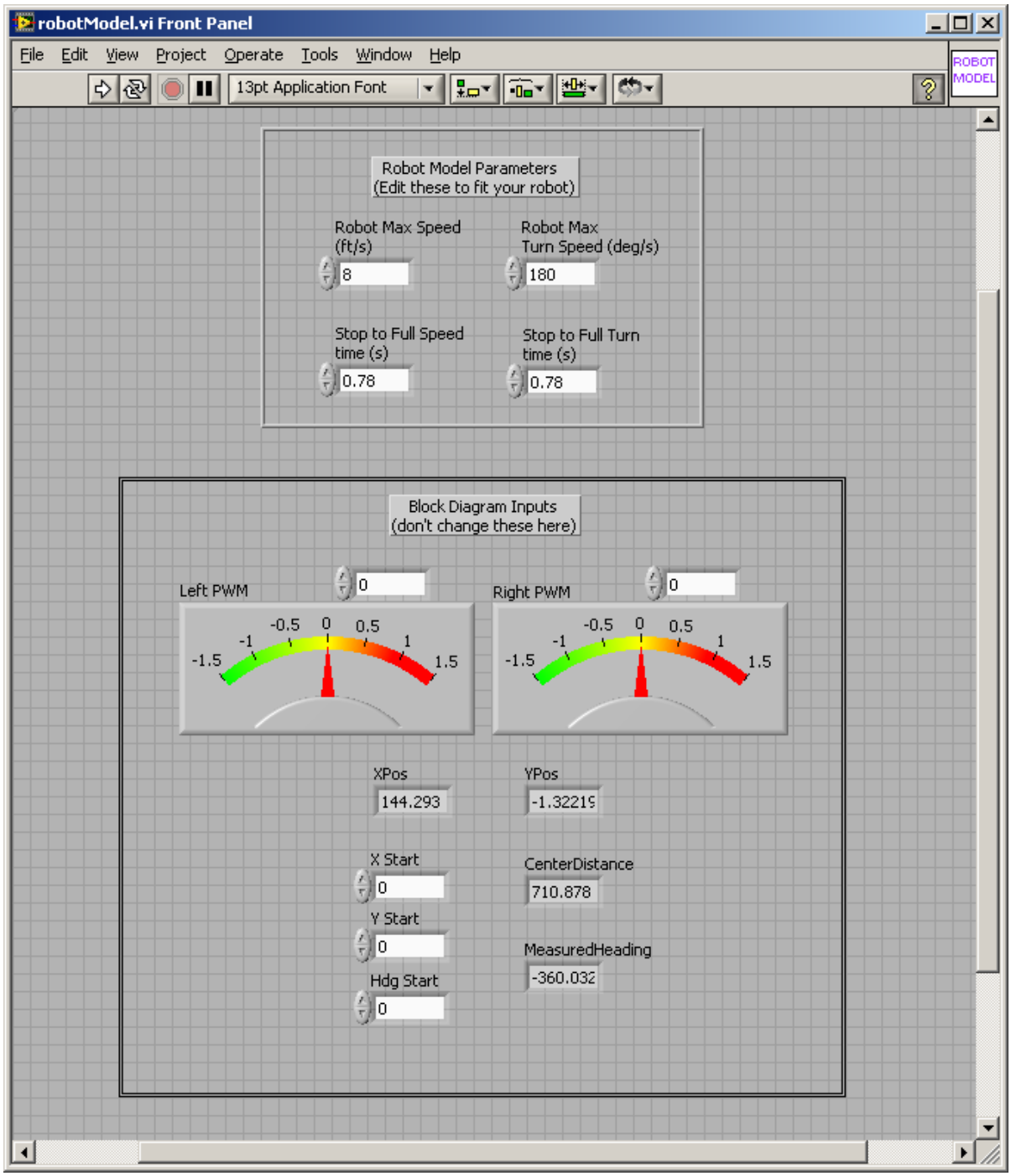


For best results, use the SubVI for your autonomous code in the simulation, NOT a copy. To do this, use "Select a VI" from the function panel, and then select your autonomous SubVI from the folder where your robot code resides. By doing this, when you debug your autonomous code using the simulation, you are making

changes to your actual robot code – there will be no need to copy the changes to your robot project after you're finished debugging! That's pretty cool.

Robot Model Interface

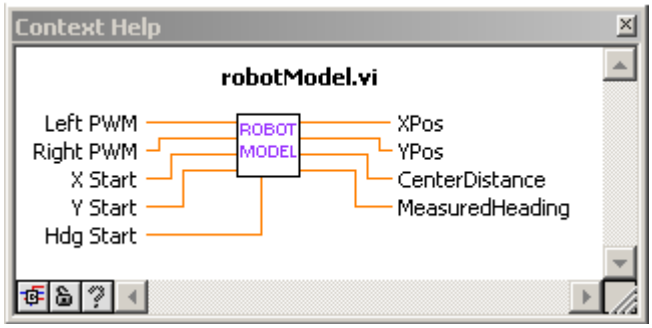
The front panel for the robot model in shown below.



You may edit the parameters in the top box. Set them based on your robot. You shouldn't need exact values here – just use your team's best judgment based on your robot design. For the "Stop to Full Speed Time", you should use the time that it takes your robot to accelerate from stopped to full speed straight ahead. For "Stop to Full Turn Time", you should use the time that your robot takes to go from stopped to a full speed turn-in-place. For both of these times, typical values should be between 0.5 to 1.0 seconds. If you have no idea what these numbers will be for your robot, the default numbers work pretty well.

The values in the bottom box are inputs in the block diagram – you cannot change them, but you can watch them while the simulation runs to see what is happening.

The context help for the Robot Model VI is shown below.



The context help shows that you are able to input the PWM for the left and right side of the drive train, as well as the starting X,Y position and the starting heading of the robot. The outputs of the model are the X,Y position of the robot , as well as Distance Traveled and Heading data that can be used as sensor values for feedback control. If you need specific left and right encoder values, you will need to change the model. I might do this in the future if enough people want it.

Important Note: The robot model uses "vehicle coordinates". That means when the robot drives forward, it is driving in the positive X direction, and the positive Y axis is to the left of the robot. Also, the robot model uses a right-hand coordinate system. That means heading would increase when turning to the left. Note that this is opposite of compass heading. If your autonomous mode uses compass headings, simply negate the MeasuredHeading output. Lastly, the robot model does NOT wrap the heading. In other words, if you keep turning left, your heading will continue to count up past 359 degrees (e.g. 357, 358, 359, 360, 361, 362, etc.) This is done to facilitate autonomous software testing.

Conclusions

As autonomous code become increasingly complex, debugging becomes increasingly difficult – especially when feedback control is used. A representative model and simulation environment not only makes testing and debugging easier and faster, it also allows testing and debugging of the code earlier in the development cycle, and allows the other team members more valuable time with the robot. It also helps eliminate "crazy runaway robot syndrome" that I'm sure every team has experienced.

The model provided here is not suitable for all robots. If this model doesn't work for your robot, the simulation framework and robot model should provide you with a great starting point for you to create a simulation that works for your team.

We have used simulation to test and debug autonomous code for many years, and it has proven to be an invaluable tool. I hope it works for you as well.