Autonomous Robotic Systems

EEL 5669

*Adaptive 2.5D Visual Servoing of Kinematically Redundant Robot Manipulators*

The following is a brief description of the results for the autonomously controlled camera simulation, completed in MATLAB via Simulink.

**Design Objective:**

The goal for this project was to re-orient the end-effector of a 6 DOF robot manipulator to a desired position and orientation, without human intervention. The desired position and orientation was to be calculated from visual input provided by a camera attached to the end-effector. Four reference points on a plane were needed to make a complete model that could be solved autonomously. The points started with initial coordinates projected onto the image obtained by the camera, and each respective point had a desired location in the image space. The points would approach their desired locations in the image space by re-orienting the robot manipulator, and hence, the camera.

**Procedure:**

1. Use static model to calculate initial conditions
   1. Calculate target point locations using camera characteristic matrix (eq 9)
   2. Calculate initial translational velocity error from extended coordinates (eq 16)
   3. Calculate homography (system of 8 equations, with 8 unknowns) (eq 10)
   4. Decompose homography (Peter Corke’s code used from online) (eq 14)
   5. Calculate initial rotational velocity error from extended coordinates (eq 19)
2. Create a dynamic camera model
   1. Calculate homography (system of 8 equations, with 8 unknowns) (eq 10)
   2. Decompose homography (Peter Corke’s code used from online) (eq 14)
   3. Solve differential equation to find angular velocity error (eq 38)
   4. Calculate each point’s translational velocity error by solving a set of differential equations (eq’s 34, 32, 27, 16)
   5. Feed error signals back into the top of the system, and allow the new values to propagate through.

**System Design:**

The system was designed largely in Simulink, with the help of specially designed MATLAB functions intended to divide and conquer the various key tasks which needed to be performed. Rotational parameters only needed to be calculated once for a given time iteration in the simulation, and translational parameters needed to be calculated for each of the four points for every time iteration in the simulation. Hence, the translational blocks have been put into 4 separate subsystems; one for each point, while the rotational blocks are more global. This can be seen in Figure 1 on the next page.

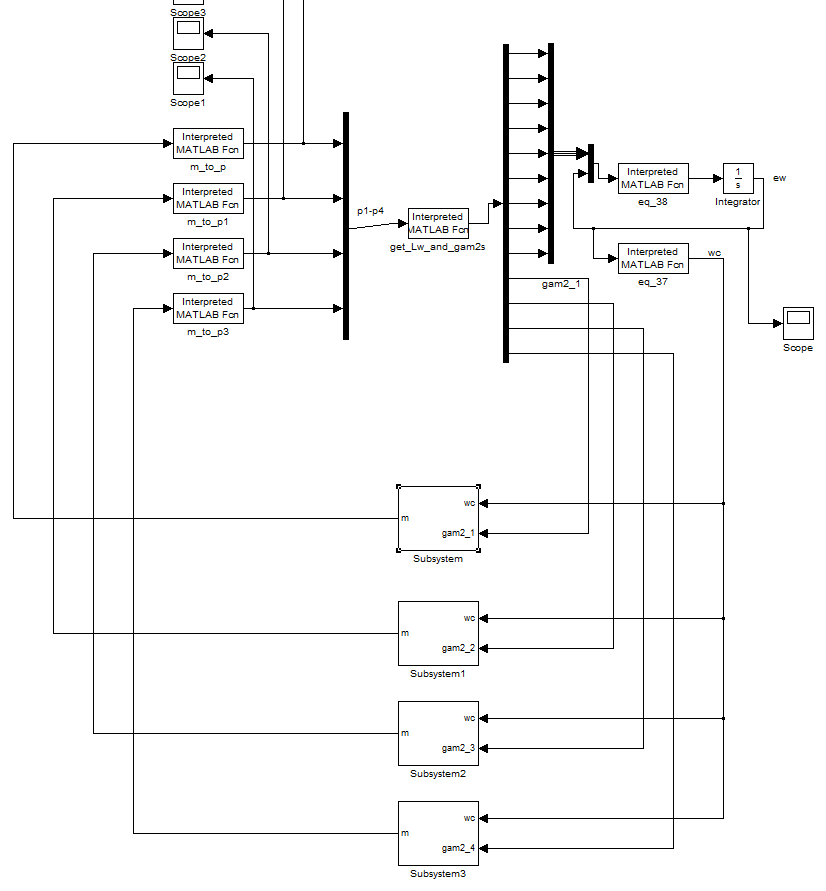


Figure 1: System Overview

For each of the four points, a new gamma 2 needed to be calculated, and thus a new subsystem needed to be created. A generic example of this subsystem can be seen in Figure 2 on the next page.

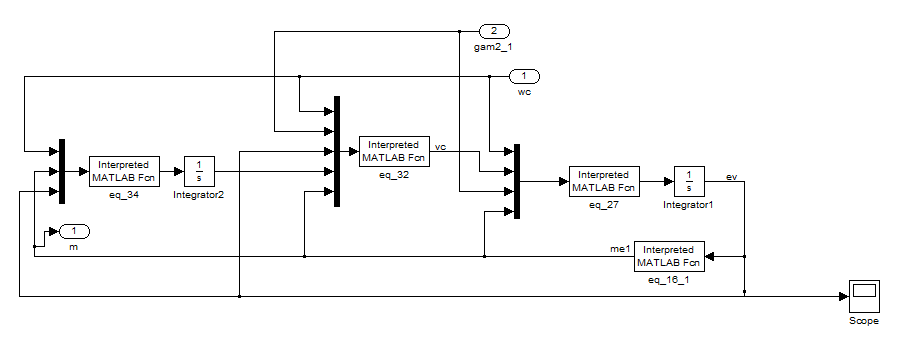


Figure 2: Generic Subsystem

**Results:**

Two things were important to see in the results of the simulation. The first was to see that all of the values converge to a finite value as time increases. This indicates system stability. The next important thing to see in the simulation results was that the values each parameter converged to was the actual desired value for that respective parameter. Fortunately, the equations and gain parameters which were implemented put the poles in the left-hand plane, so to speak, and allowed the outputs to converge into a stable solution.

Figure 3 on the next page shows the error in angular velocity, for each coordinate axis component, converging to zero as hoped for. This means that as time goes on, there will be less and less angular velocity error. After 10 seconds, it can be seen that the angular rotation error is negligible.

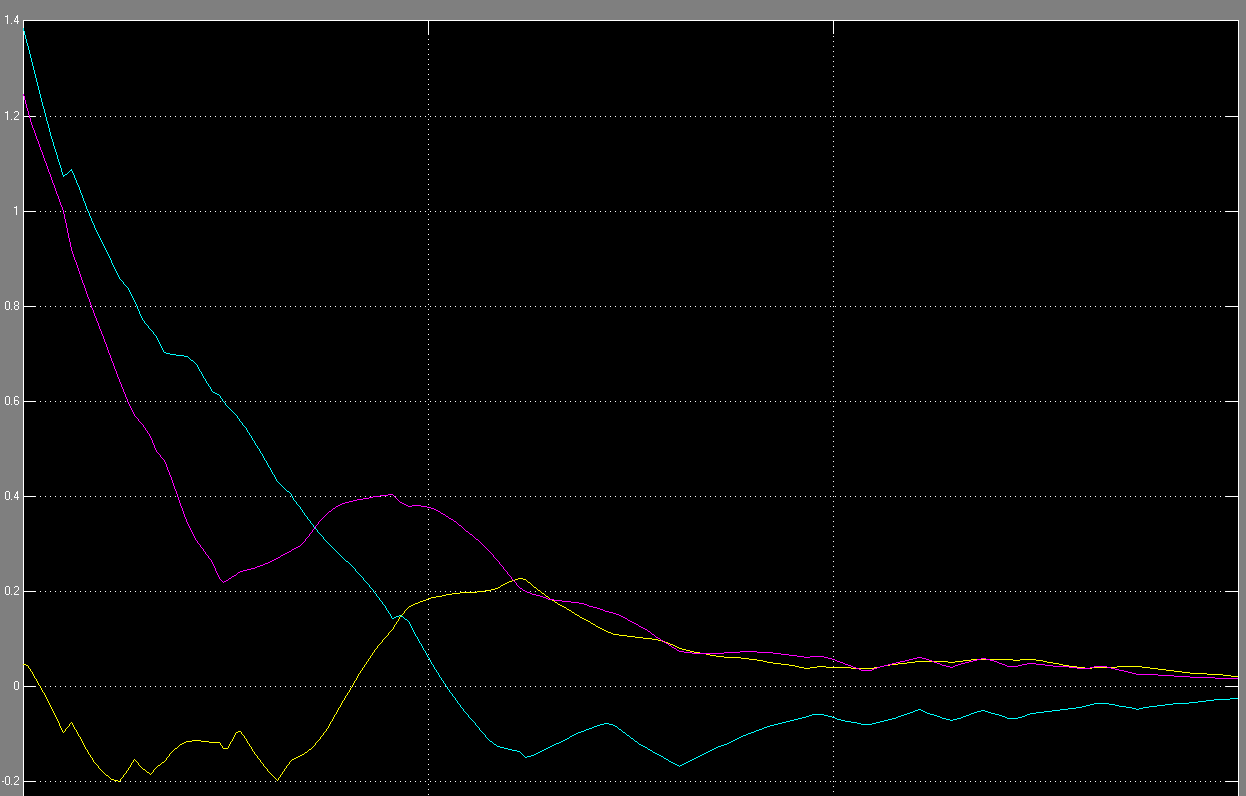


Figure 3: Angular Velocity Error (ew) vs. Time

Figure 4 shows the translational velocity error for the point 1. As time goes on, the points get closer and closer to the displacement they need to be from one another, so each point will approach the same translational error. Therefore, only one picture is necessary to demonstrate the overall translational error for the collection of four reference points.

As it can be seen in the figure, the errors in translational velocity for each coordinate axes converge to zero as time increases. This is evidence of a successful simulation.

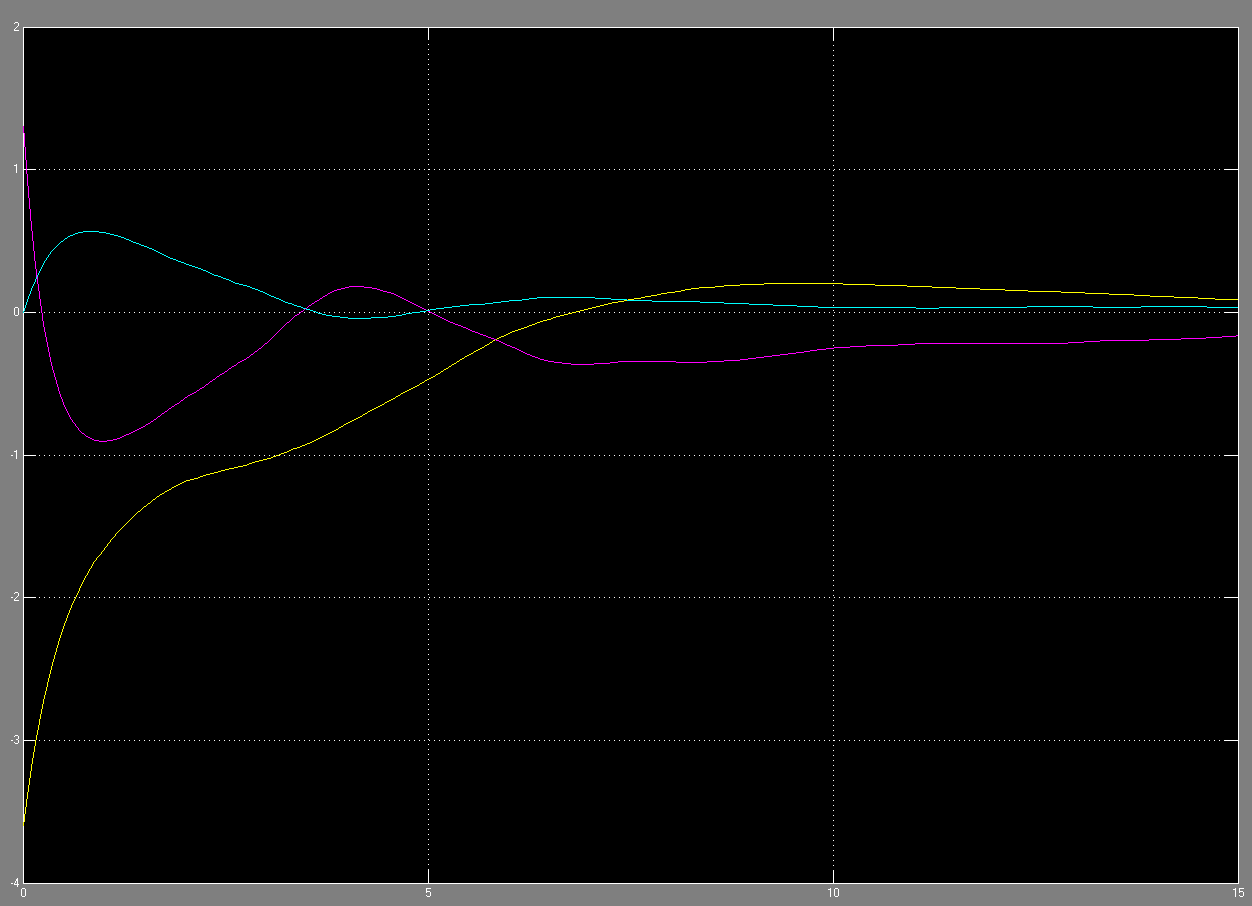
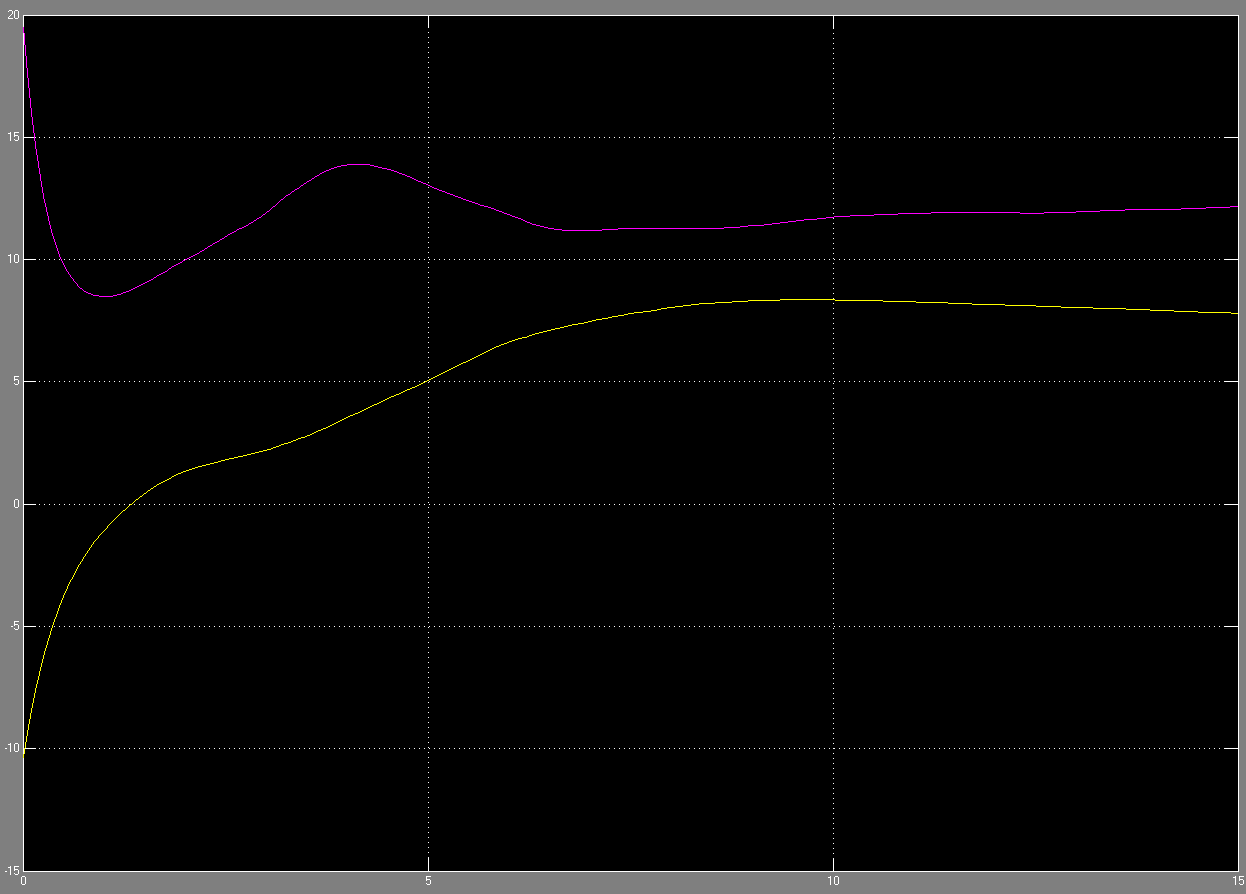


Figure 4: Translational Velocity Error (ev) vs. Time

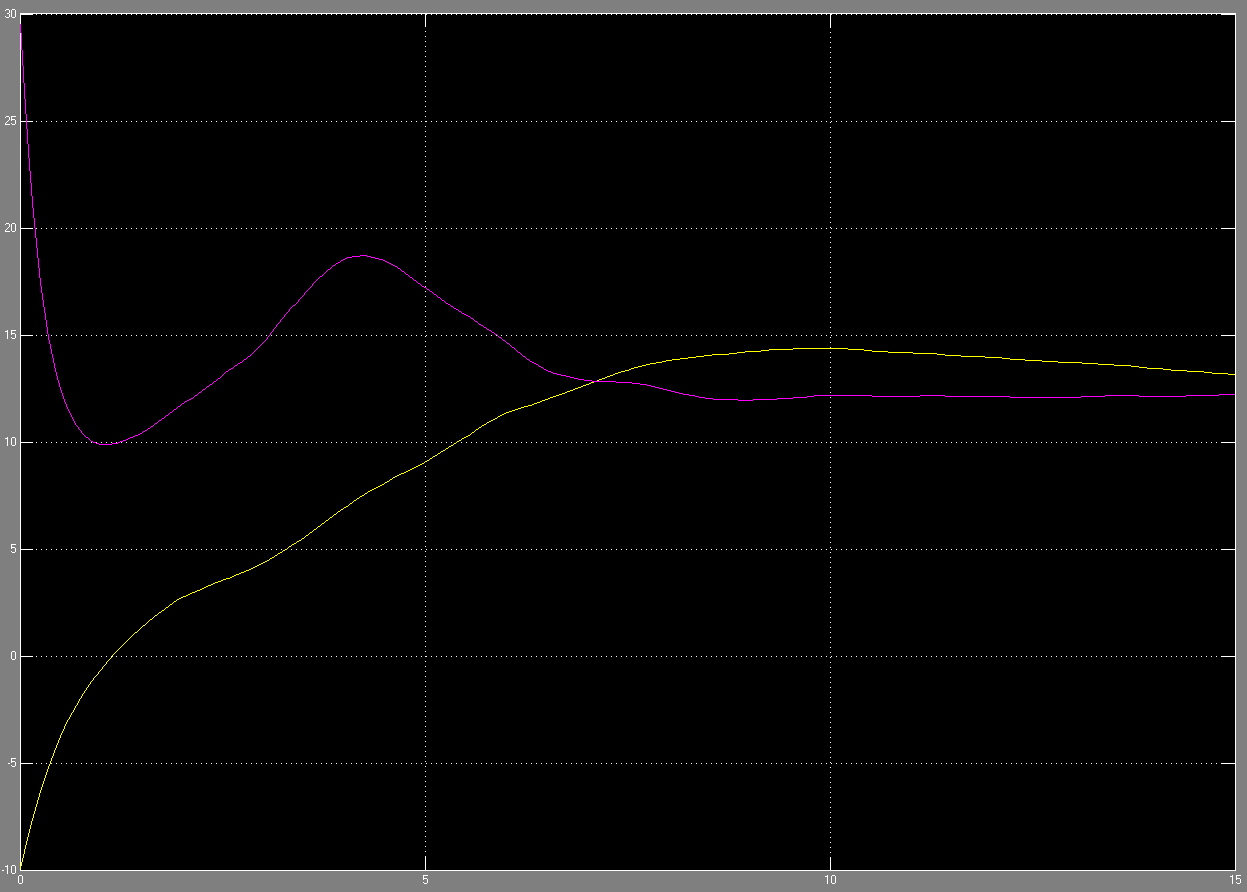
*Reference Points:*

The following four graphs map out the paths taken by each of the reference points in the image space.



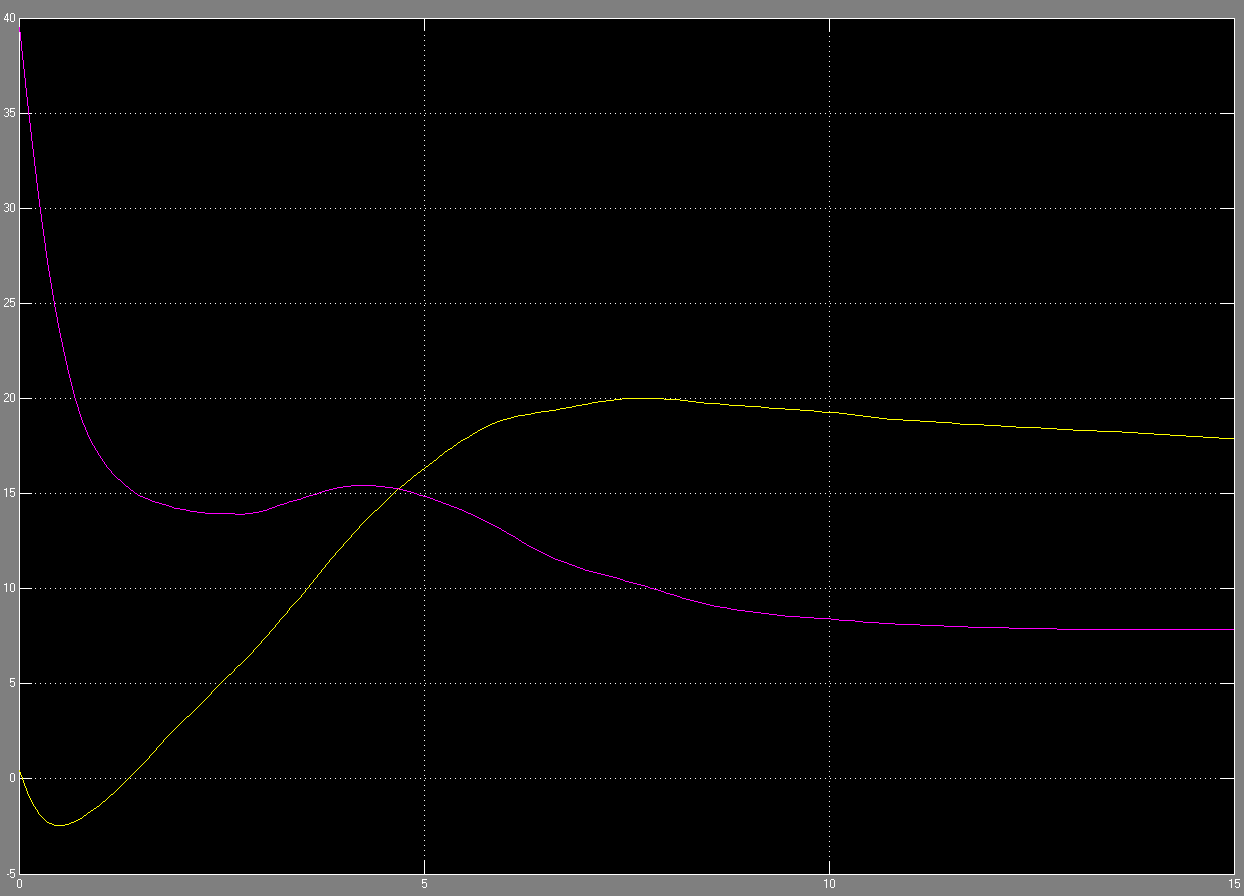
**Initial = [-10.34 19.5] 🡺 Final = [7.4 13 ]**

Figure 5: Reference Point 1



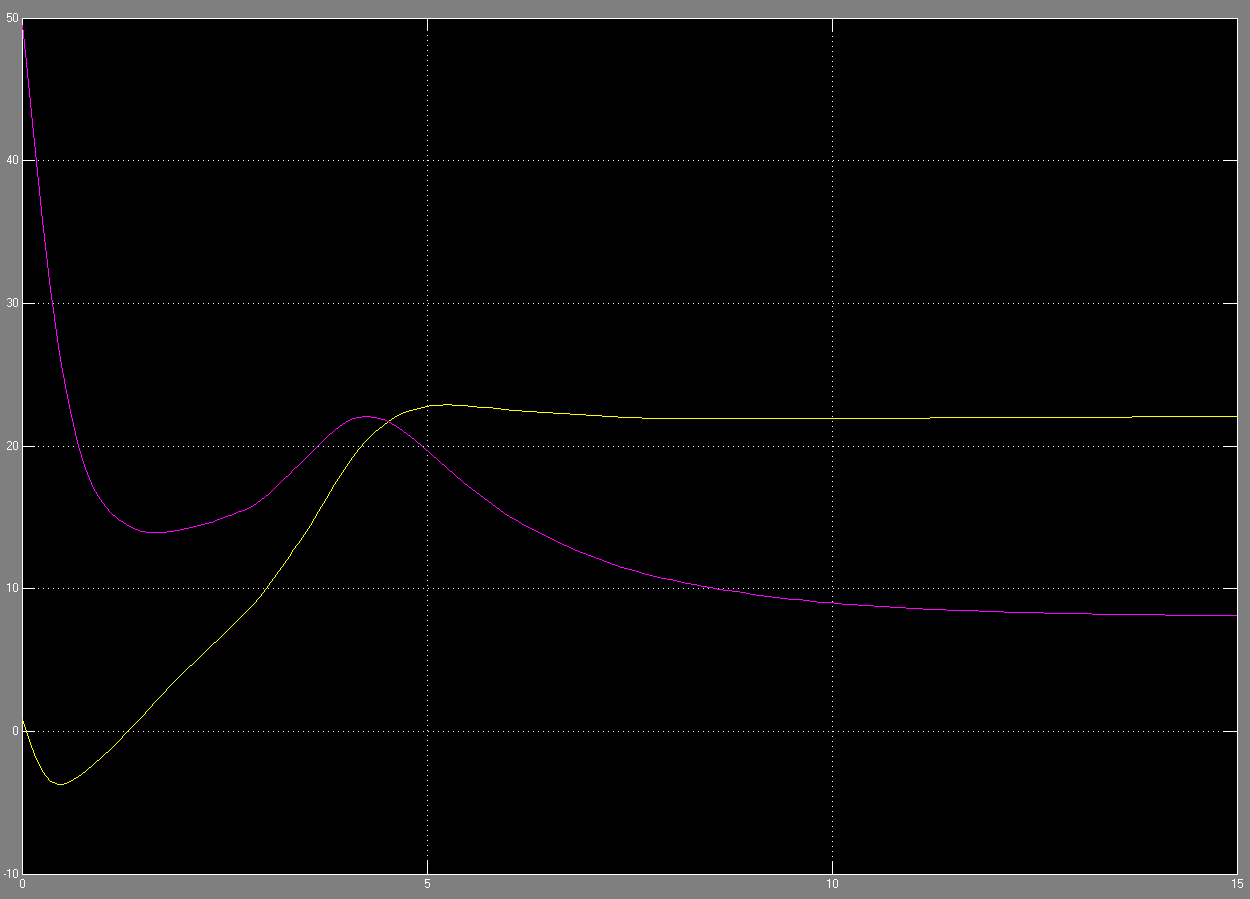
**Initial = [-9.94 29.5] 🡺 Final = [12.4 13 ]**

Figure 6: Reference Point 2



**Initial = [0.46 39.5] 🡺 Final = [17.2 8 ]**

Figure 7: Reference Point 3



**Initial = [0.86 49.5] 🡺 Final = [22.2 8 ]**

Figure 8: Reference Point 4

**Conclusions:**

While the complete robot dynamics were not implemented due to lack of experience and time constraints, the results for the camera model were very pleasing to see. All of the measured physical parameters converged to their expected values. This could not have been possible without being based on solid mathematical theory, and giving meticulous attention to detail. The differential equations and gain factors were chosen and set up in such a way that the system turned out to be very stable.

Throughout the design process, much was learned about MATLAB and Simulink. Converting arrays, matrices, and other parameters into a suitable format for passing between blocks was somewhat unintuitive, yet easy to debug, thanks to MATLAB’s robustness. Very few semi-colons were used, so that if and when a run-time error did occur, the source of the problem could be easily spotted. Some of the mechanics were a bit unexpected, but once the syntax and procedures had been learned, the process became very consistent and straightforward.

The fact that the robot took about ten seconds to converge to the desired points seems to have much room for improvement; especially in today’s world, where speed is a very attractive quality to have in the marketplace.