Robonaut Solution Writeup

Dennis Huo, 2013 May 12th

Overview

My solution consists of all self-written code (no open source libraries), and only performs 2-dimensional image processing; it does not attempt to extract any 3D model, and dual-image inputs are only used to reconcile the location of solution elements if the normal algorithm fails to do so on one of the images.

All code was first implemented in Java before porting to C++ for easier visualization using Java graphics libraries.

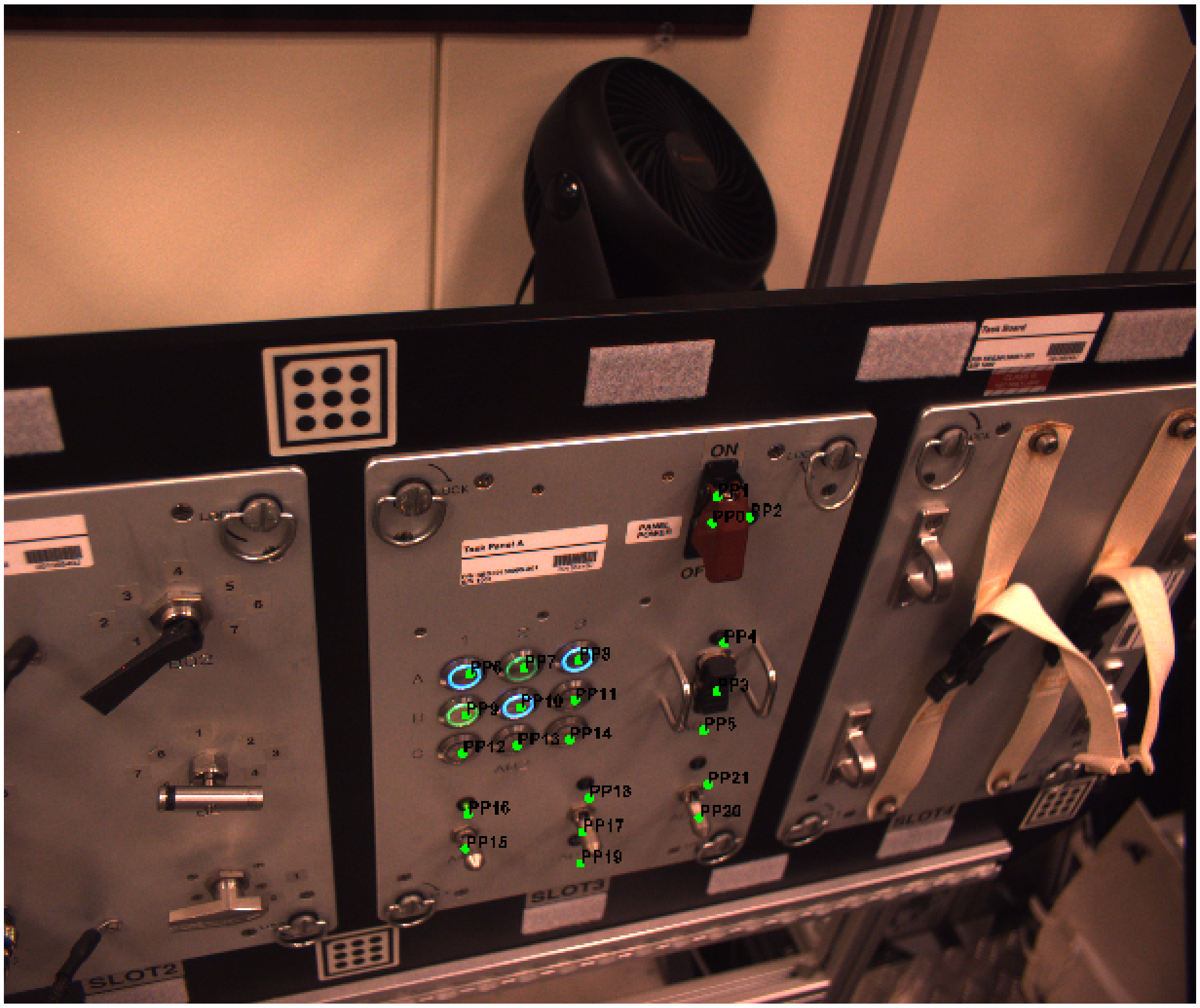
In total, there are roughly a handful of algorithms used in my solution, these will be described in more depth below.:

- Niblack for defining each pixel as “foreground” or “background”
- Flood fill to find all connected components as defined by their Niblack category being unchanged in the 8-pixel neighborhood; ignore pixels which are “middle” niblack states.
- Hand-tuned heuristics based on size/shape/color to find which components are the Red Cover, the Rocker Switch, and Green/Blue LEDs (same color thresholds for donuts and dots).
- Define “expected” positions using extracted basis-vector coordinates from training data after making best effort to find the components which define the basis vectors (e.g. Panel LED, Rocker switch).
- Brute-force DFS to find optimal matchup of candidate connected-components to taskboard components; optimality is defined by a function of RMS distance from “expected” position times a function of the consistency in angle deviation from the expected position. First performed for lit green LEDs, then lit blue LEDs, then unlit grid buttons, then everything remaining.

Tools and helpers

The following Java main-classes are present:

BatchProcessor.java - This tool runs through all input files as listed inside Trainer.java, and loads model.csv to test extracted basis-vector coordinates by drawing associated dots for all the taskboard components on a new copy of each input image before saving the marked images to a new directory.
Example output image:



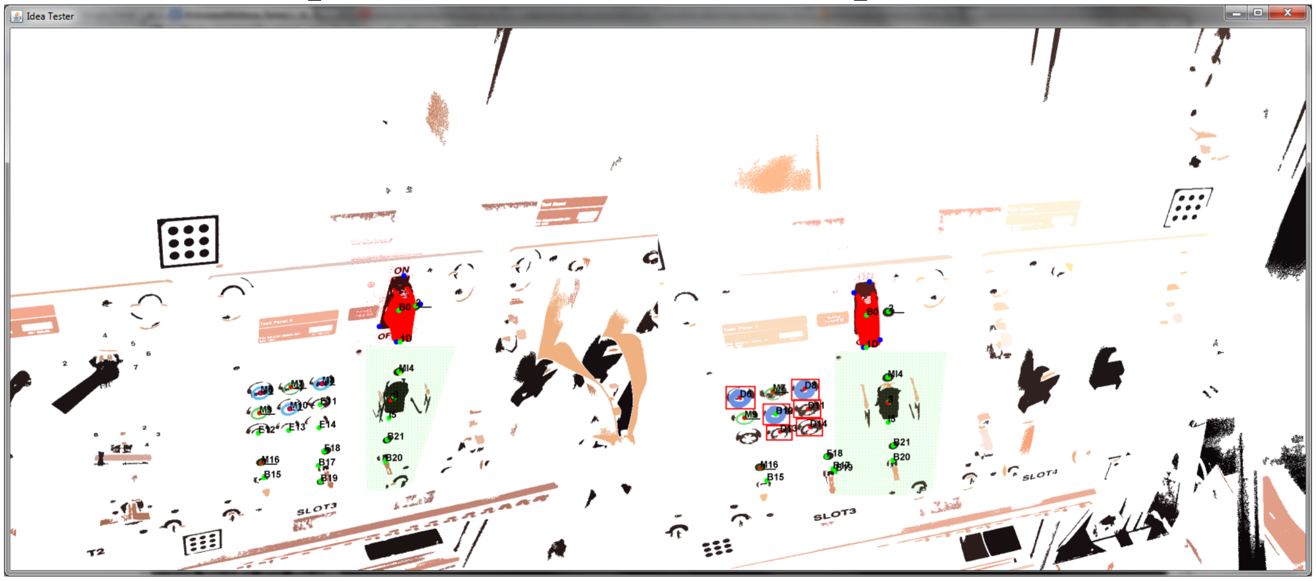
Trainer.java - This tool loads model.csv and runs through every row of the training data, treating some manually selected triplets of points (e.g. Panel LED, Rocker Switch, and middle of button Grid) as basis vectors for all other components on the board. It then prints the relative coordinates of every component as defined by the basis vectors. It averages all such values across all images and prints a handy double[22][2] array.

```
C:\projex\topcoder\ved_robonaut>java Trainer
ISS/BHXG.tif: N/A 0.80,1.03 0.78,0.82 0.76,0.61 1.00,1.00 0.99,0.80 0.97,0.57 1.20,0.97
1.19,0.76 1.18,0.56 1.50,0.87 1.47,0.40 1.79,0.37 1.42,-0.11
ISS/BHXG.tif: N/A 0.80,1.04 0.78,0.83 0.76,0.60 1.00,1.00 0.98,0.79 0.96,0.57 1.19,0.95
1.18,0.75 1.16,0.53 1.48,0.84 1.46,0.35 1.79,0.29 1.42,-0.19
<...>
Sim/YXQV.jpg: 1.19,0.02 0.77,1.03 0.77,0.82 0.75,0.60 1.00,1.00 0.99,0.79 0.99,0.58
1.21,0.98 1.21,0.78 1.20,0.57 1.56,0.90 1.55,0.44 N/A 1.56,-0.03
Sim/YXQV.jpg: 1.18,-0.01 0.77,1.04 0.76,0.82 0.76,0.59 1.00,1.00 0.99,0.79 0.98,0.57
1.22,0.97 1.21,0.76 1.20,0.55 1.57,0.89 1.56,0.42 1.91,0.39 1.55,-0.06
double compRatios[22][2] = {
    {0.02,0.15},{0.07,0.19},{0.00,0.00},{1.00,0.00},
    {0.71,-0.01},{1.22,0.02},{0.78,1.02},{0.77,0.81},
    {0.75,0.59},{1.00,1.00},{0.99,0.78},{0.98,0.57},
    {1.22,0.98},{1.21,0.77},{1.20,0.56},{1.76,0.89},
    {1.57,0.90},{1.73,0.42},{1.54,0.42},{1.91,0.41},
    {1.70,-0.05},{1.51,-0.06}};
Everything took 21346 ms
```

RobonautEyeTester.java - This is the visualizer provided by TopCoder to all competitors. I modified it to output raw bytes to the C++ solution instead of using a PrintWriter, and this made testing a single large (ISS, Lab) image on the naive (hard-coded) solution go from ~60s to ~500ms.

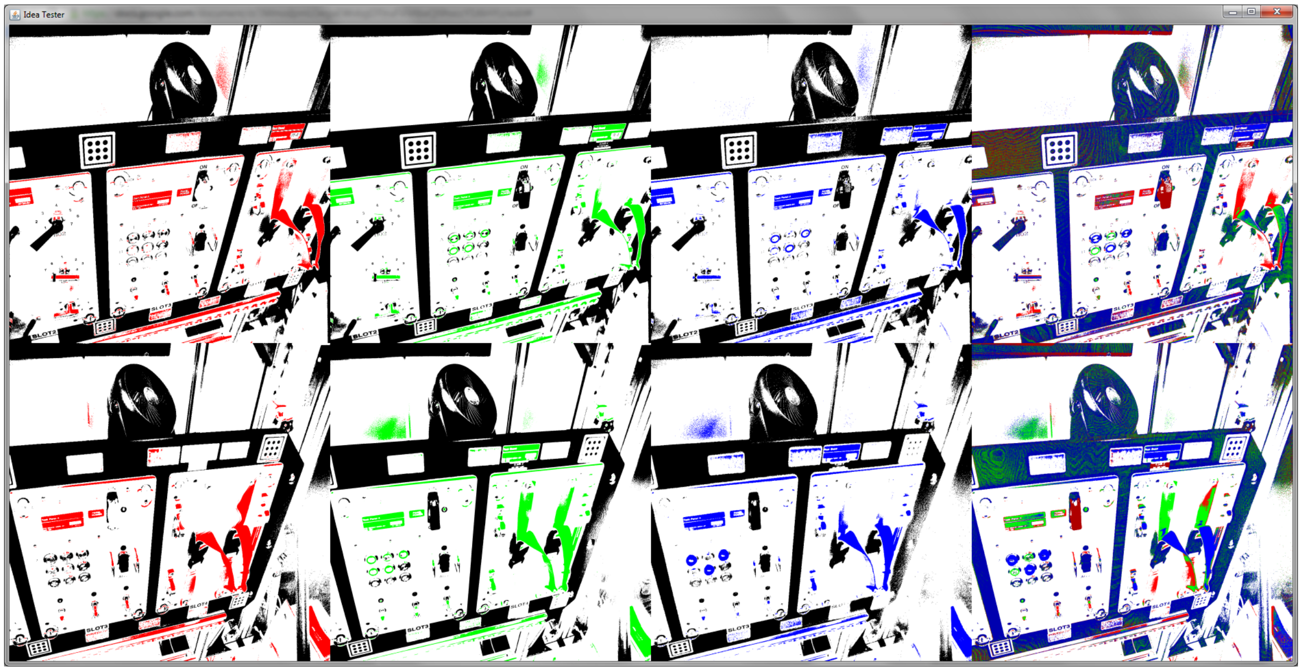
IdeaTester.java - This is the main "solution" program which closely parallels the C++ solution. Instead of formatting the overall model state for output, however, it draws dots on the image and displays it in a popup JFrame/JPanel. It draws the connected-components instead of the raw image as the baseline, using the "average" color of each component instead of the real color of each pixel in the component.
Screenshot:

```
C:\projex\topcoder\ved_robonaut>java IdeaTester Lab\LeftImage_AHJH.tif
```



ImageProcessor.java - This file contains the logic for performing niblack categorization of pixels, including the initial creation of the per-channel "Sum Images" to efficiently compute average and standard deviation of each pixel. It displays the per-channel niblack results, as well as the combined version ultimately used in the solution.
Screenshot:

```
C:\projex\topcoder\ved_robonaut>java ImageProcessor Lab\LeftImage_AHJH.tif
```



TestAll.java - Simple multithreaded test harness which automatically lists all input images and runs “a.exe” with RobonautEyeTester, recording the final score given by RobonautEyeTester. Allows running all 128 images on the naive solution in about 20 seconds; the final implementation takes around a minute to test all images.

Solution Details

Every solution begins with the following two preprocessing algorithms: Niblack Binary Classification and Flood-Fill Connected Components.

Niblack Binary classification

The Niblack algorithm simply compares each pixel with independent RGB channels to the average and standard deviation in that pixel’s vicinity. If the pixel is less than “average - A * stddev” then it is considered “background”, and if larger than “average + B * stddev”, it is considered foreground. “A” and “B” in this case were manually configered through trial and error, using the ImageProcessor.java as seen in the ImageProcessor screenshot.

The kernel over which the averages and stddev were taken were simply a hard-coded “width / 5”. This large kernel still allowed efficient computation of average/stddev by using a “Sum Image” and “Sum Squared Image” which is just the 2-D grid of integrals over the RGB value of each preceding pixel; this way the time complexity is not a function of the kernel size. The following code snippt from sol.cc inside “doNiblackMulti” illustrates the core of the algorithm:

```
static double UPPER_NIBLACK = 1.5;

  double UPPER_LIM = UPPER_NIBLACK * UPPER_NIBLACK;
  double LOWER_LIM = -.75 * .75;


      if (abs(maxDev) > abs(minDev) && maxDev > UPPER_LIM) {
        sumImage.isForeground[cur] = true;
        if (maxChan == 2) {
          sumImage.isRed[cur] = true;
        } else if (maxChan == 1) {
          sumImage.isGreen[cur] = true;
        } else {
          sumImage.isBlue[cur] = true;
        }

      } else if (abs(maxDev) < abs(minDev) && minDev < LOWER_LIM) {
        sumImage.isBackground[cur] = true;
        if (maxChan == 2) {
          sumImage.isRed[cur] = true;
        } else if (maxChan == 1) {
          sumImage.isGreen[cur] = true;
        } else {
          sumImage.isBlue[cur] = true;
        }
      }
```

Flood-Fill Connected Components

The screenshot from IdeaTester.java shows the results of componentization; it simply flood-fills from each pixel as long as the pixel has a neighbor in the 8-neighborhood of the same niblack classification. The niblack pixels with value between LOWER_LIM and UPPER_LIM are ignored, and considered to be "noise"; in practice, these pixels occur at the boundaries between background/foreground components. This is the method "findScc" in sol.cc and in Scc.java.

An earlier solution used the actual color values instead of niblack cateogry; this tended to work better for Lab3 images but worse for ISS/Lab/Lab2/Sim.

Overall Solution Flow

The following steps are performed in order in the final solution:

1. Compute niblack classification.
2. Compute connected components. While doing so, recognize very bright green

components as “lit green LEDs” and very bright blue ones as “lit blue LEDs”. Throw away very small and very large conncected components.

3. Tentatively guess the topmost lit green LED is the Panel LED.
4. Use the Panel LED guess to look for the Red Cover, as defined by the function “**findRed()**” in sol.cc and IdeaTester.java; look directly to the left of the estimated Panel LED.
5. If no Red Cover found based on Panel LED guess, search entire image for highest-density red pixels; use this new position as the revised Red Cover, undo the previous guess for Panel LED.
6. If we had to search whole image for Red Cover, look to its right for a new position of Panel LED (“**findPanelLed()**”), with more relaxed definition of “lit green LED”. Otherwise, commit position of both Red Cover and the Panel LED.
7. Use the better estimate of Red Cover and Panel LED to search for Rocker switch immediately downwards; its shape should be close to rectangular, its color Max(r, g, b) < 100, and we must not have to cross a “very large connected component” along the path from the Rocker Switch to any lit LED (**findRockerSwitch(), mightBeRockerSwitch()**)..
8. Use the vector (rockerPos - panelLedPos) and its own orthogonal vector as a set of basis vectors for expressing the coordinates of all other components using the rewritten coordinates generated by Trainer.java.
9. Take the set of lit green LEDs found in the SCC pass; brute-force all permutations of their matchup against the set of possible taskboard green LEDs ({ 4, 7, 9, 11, 13, 16, 18, 21 }) (**matchSccToGreenLedsDfsRect()**). The matchup with lowest “error” as a function of cartesian distance from expected positions in (8) and the maximum deviation of their angle differences. The angular deviation means we expect the estimates to be systematically wrong because of unaccounted 3D effects; the matchup is best if all angular corrections agree to within a minimal bound. Successful matchups also affect the component “state” between “ON” and “OFF”. 16, 18, 19, and 21 also change the state of the bottom toggle switches between “UP”, “DOWN” and “CENTER”. Additionally, the dfs matchup is subject to the constraint of following the x-axis and y-axis topological ordering (e.g. Panel LED is smaller y-coordinate than A2 LED, etc).
10. Do the same as (9) for lit blue LEDs.
11. Within the donut-button grid, we can accurately extrapolate positions of unlit buttons in any row or column which has 2 lit buttons with simple linear interpolation.This is done interatively as long as we found an interpolation in a previous pass inside the function “**fillGrid()**”.
12. Basis-interpolation is used for “rocker top LED” and “rocker bottom LED” regardless of whether a connected component was found there. (**interpolateRockerLeds()**).
13. Use the raw basis-vector estimated position for component 21, since it is close to being aligned with the line between panelLed and rockerSwitch so the 3D correction isn’t very severe.
14. Now use the same DFS algorithm for the unlit buttons, either using (rocker(3) - panelLed(2)) plus (gridCenter(10) - rocker(3)) if gridCenter was found from previous

work, otherwise just use rocker(3) - panelLed(2) and its orthogonal vector just like before. This is **basisDfsMatch()** and **basisDfsMatchWeak()**, respectively.

15. Re-run raw basis-vector estimation in **basisExtrapolate()** and **basisExtrapolateWeak()** for any remaining unlit buttons, looking for the nearest connected-component from predicted positions up to a maximum distance, otherwise just take the raw predicted position.