Computer Vision

In FRC Robot Designs

Table of Contents

Specifications

Goals

Develop a software/hardware sub-system that can be used for computer vision based auto-targeting and FRC field element recognition

Requirements

1. Solution needs to operate in real time on real hardware
2. Solution needs to be supported in simulation as well as real hardware
3. Solution must at least offer basic image processing libraries
4. Solution needs to be low cost

Desirable Features

1. Can segment objects (“particles”,”ROIs”) from images (using filters, moment analysis etc.)
2. Can identify specific objects in images (e.g. target tape pattern, totes etc.)
3. Object identification accuracy can be improved by training
4. Offers Multi-Processor Support (multi-core cpus, GPUs)

Available Software packages and APIs

1. CUDA (Compute Unified Device Architecture**)**
    - C++ API that includes compiler directives that support multi-processing
    - Only supported on NVIDIA hardware (graphics boards)
2. cuDNN (NVIDIA Deep Neural Network Library)
    - Provides high level API for “deep learning” programming
    - Uses CUDA compute engine as back end
    - Only supported on NVIDIA hardware
3. Caffe (Deep learning Framework)
    - Developed by the Berkley Learning and Vision Center
    - Provides Python or Matlab API shells to CUDA(GPU) or OpenCL(CPU only) compute engines
    - Caffe-cuDNN requires top-end NVIDIA hardware for training
    - Trained networks can be deployed on lower-end NVIDIA hardware (e.g. Jetson)
4. Digits (graphical interface for deep learning)
    - Requires top-end NVIDIA hardware and software
    - Trained networks can be deployed on lower level hardware (e.g. Jetson)
    - Supports “deep learning” algorithms (Neural networks)
5. OpenCL (Open Computing Language)
    - Open source alternative to CUDA
    - Runs on most hardware platforms (AMD,NVIDIA,INTEL)
6. OpenCV (Open Computer Vision)
    - Runs on most os platforms (Linux, OSX, Windows, Arm)
    - Supports image filtering
    - May also support object identification and training (need to investigate)
7. IMAQ (ni-vision)
    - National Instruments proprietary software (commercial)

- Only supported on RoboRio hardware
- No possibility of simulation support
- Supports object segmentation only (no object classification or training)

8. GRIP
    - Recently adopted by First as an alternate image processing package (over ni-vision)
    - Uses OpenCV image processing background engine
    - Uses JavaFX for front end UI
    - Can be built from source on Windows and Linux (Mac?)
    - Has GUI tuning application and headless implementation for deployment

Target Hardware platforms

1. Multicore x86_64 CPU
2. Raspberry Pi
3. Jetson
4. RoboRio

Strategies

Image processing Strategies

Use a relatively simple image processing API (e.g. OpenCV, IMAQ)

- The image processing system uses filters, color match algorithms etc. to isolate portions of a camera image that meet certain criteria for special target objects (reflective tape patterns etc.)
- The image processing system performs an analysis on these regions of interest (ROIs) and extracts basic information such as length,width (in pixels) position of object center on the screen etc.
- The reduced ROI data is sent from the image processing system to the Robot application
- The Robot application uses the ROIs center point width, height etc. along with prior knowledge of the objects size and shape to determine both the distance of the object from the camera and it's horizontal and vertical offset angles
- The Robot application uses the geometric data obtained for the target to adjust it's heading,

shooter angle etc. as appropriate

Use an advanced object recognition API such as cuDNN

- Similar to that described above except that a more advanced technique (e.g. cuDNN) is used to identify objects based on general training data rather than relying on a set of unique target properties that may not be available in all situations
- Using such advanced techniques might require a more capable compute resource such as a NVIDIA Jetson or graphics card

Deployment Strategies

Use a single processor for image processing and Robot application

1. Scenario
    - Capture images on RoboRio and run separate processing application (or integrated library) to extract features
        - e.g. GRIP(application) or IMAQ(library)
        - Can use inter-process or network communication protocols if image processing is carried out in a separate application
    - Use extracted image features to direct control
        - eg. Change angle for shooter target
    - Send Images up to driver station for user feedback
        - target areas in images could be annotated (surrounded by boxes etc.)
2. Pros
    - Simplest and cheapest to implement (since only one processor board is required)
    - Can use similar a strategy for simulation (since everything must run on the Linux host)
3. Cons
    - May run into processing bottleneck problems

Use a separate computer for image processing on deployed hardware

1. Scenario
    - Use RoboRio for Robot application
    - Use a second computer (e.g. Raspberry Pi or Jetson) for Image processing

    - Camera data only captured on processing board
- Image processor extracts feature information and sends it to the RoboRio (e.g. via local ethernet)
- Image processor sends raw or annotated images up to driver station
- Roborio uses feature information to direct control

2. Pros
    - Uses an external processor to reduce computation overhead on RoboRio CPU
    - Has the potential to employ advanced object recognition algorithms such as NVIDIA cuDNN or openCV
3. Cons
    - Adds expense and complexity

Process raw images on host platform (driver station)

1. Scenario
    - Use RoboRio for Robot application
    - Send raw images from camera to driver station
    - Process images on driver station and extract features
    - Send feature information back to RoboRio to direct control
2. Pros
    - Can use high performance laptop to do the image processing
    - In teleop mode the driver probably wants to see the camera images anyway
    - No extra hardware to buy
3. Cons
    - May suffer from network IO bottlenecks because of the need to transfer images from the RoboRio to the Driver Station laptop
        - In competition, radio bandwidth is limited (e.g. 7MB/s) which will reduce image frame rate for both teleop and autonomous modes
    - Requires a higher performance driver station computer
    - Since the host data station does not run a RTOS (it runs Windows 10) image processing

latencies will be indeterminate

Simulation Strategies

At the present time all simulation processing needs to run on the same (multi-core) CPU in a Ubuntu 14.04 operating system environment. In addition, because of a bug in Gazebo that causes a loss of depth information in the (simulated) camera images, the computer needs to run Linux natively (e.g. in a separate boot partition) and be equipped with a middle to high end graphics card (e.g. an integrated Intel GPU running Linux also exhibits the problem). The depth ordering bug is also evident when running Gazebo in a Virtual machine (under Windows) even on a system with a high end NVIDIA graphics card

Requirements

From a software point of view the vision interface to the Robot application should not require any (or much) special coding in order to run in the simulator or on real hardware

Strategy

- Capture a (simulated) camera publication from Gazebo and convert the image data to a form that can be input to an application like GRIP, raw openCV or an advanced API such as cuDNN
- Process the image using one of the tools mentioned above and extract relevant object position information such as center of mass, height, width etc.
- Pass the position information for the identified object (or objects) in a array of structures using a protocol that is also available on real hardware (e.g. the FRC NetworkTables protocol)
- Let the Robot application decide what to do with the object position data

Software SubSystems

GRIP

Notes for building and testing on Ubuntu 14.04

- note: GRIP can also be built on Windows (but this was not tested)

References

1. https://wpilib.screenstepslive.com/s/4485/m/50711/c/150895

   GRIP “Alpha”

2. https://usfirst.collab.net/sf/projects/grip_computer_vision_engine/ (?)

   note: this doesn't appear to be the same source tree that is described in the screenstepslive web page above (orphaned project ?)

3. https://github.com/WPIRoboticsProjects/GRIP/wiki/Setting-up-build-tools

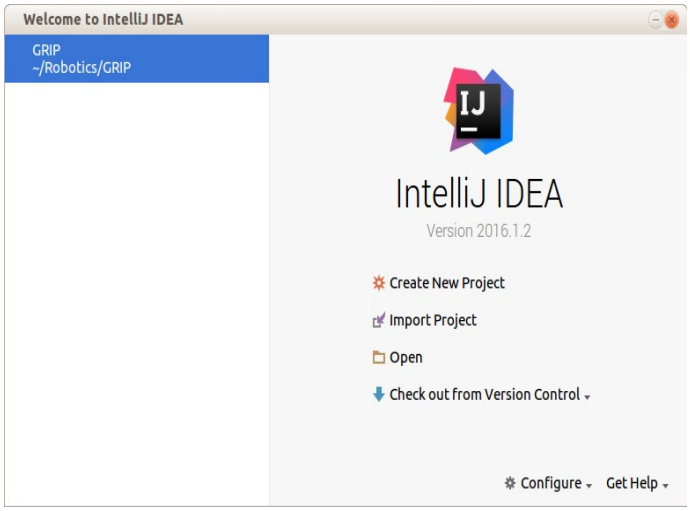
Installation

1. Clone the git repository

   git clone https://github.com/WPIRoboticsProjects/GRIP

   - creates a subdirectory GRIP in parent directory (i.e. where git clone was issued from)

2. Install IntelliJ (recommended Java IDE from ref. 3 above)
   - download from: https://www.jetbrains.com/idea/#chooseYourEdition (Community version)
     - ideaIC-2016.1.2b.tar.gz
     - installation instructions: https://www.jetbrains.com/help/idea/2016.1/installing-and-launching.html?origin=old_help#d1799863e158
   - Untar .gz file into public directory (e.g. /opt)
     - created directory: /opt/idea-IC-145.972.3
   - run installation script
     - cd /opt/idea-IC-145.972.3/bin
     - ./idea.sh
       - prompts for root password, then installs launcher script: /usr/local/bin/idea
   - Open GRIP project from IntelliJ
     - $ idea
     - Choose Import project



- Browse to GRIP main directory and select build.gradle as project file

Building and Launching

1. Build UI and Run from command line (instructions from README.md)

   $ ./gradlew :ui:run

2. Build and run UI from IntelliJ
   - (menu)Run->Edit Configurations
   - press “+” and then select new Gradle configuration
   - Enter a configuration name (e.g. “Run UI”)
   - press “box” icon to right of “gradle project” entry line
   - Select GRIP:ui from popup menu
   - enter “:ui:run” on “tasks” line
   - press “OK”
   - (menu)Run ..
   - enter “Run UI” from popup menu

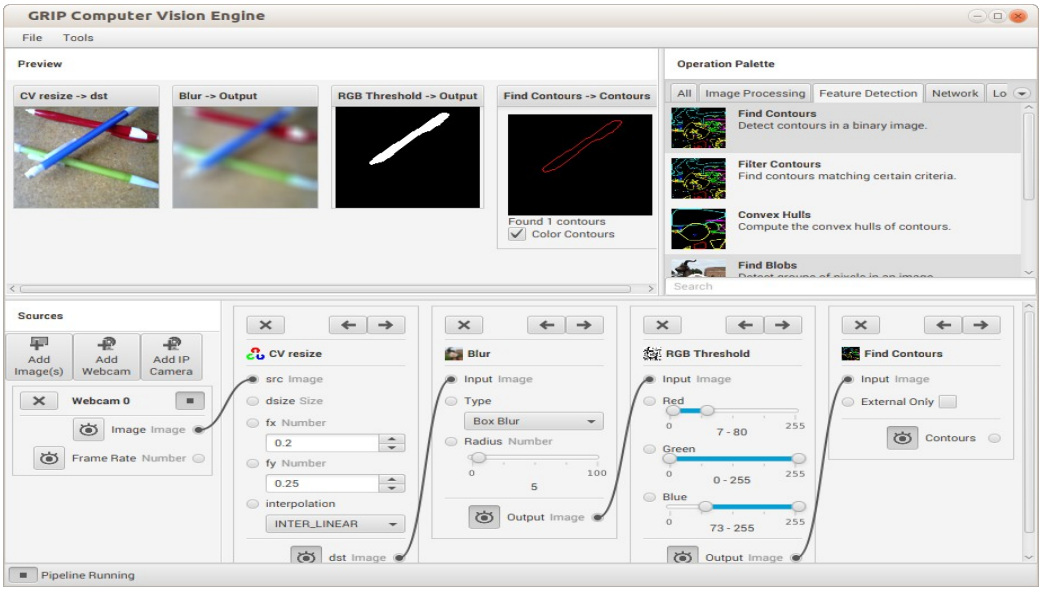
Investigation Plan

1. Build and Run JavaFX GUI on Ubuntu
2. Test Headless GRIP interface
   - Use print statements, write to log file or look at network tables GUI to validate
3. Develop interface between Gazebo output (Camera publication) and GRIP input (UI and headless)
   - Emulate webcam or mpeg stream input modes ?
4. Test or develop interface between GRIP output and Eclipse Robot application (FRCUserProgram)
   - In simulation mode everything runs on the CPU
   - In deploy mode FRCUserProgram runs on the RoboRIO (but will not test this)
   - Q: Can FRCUserProgram intercept networkTables publications ?
5. Test or develop interface between GRIP output and Smart Dashboard

- Smart Dashboard normally intercepts networkTables publications from FRCUserProgram
  - Q: Can it also capture networkTables publications from GRIP ?

Tests

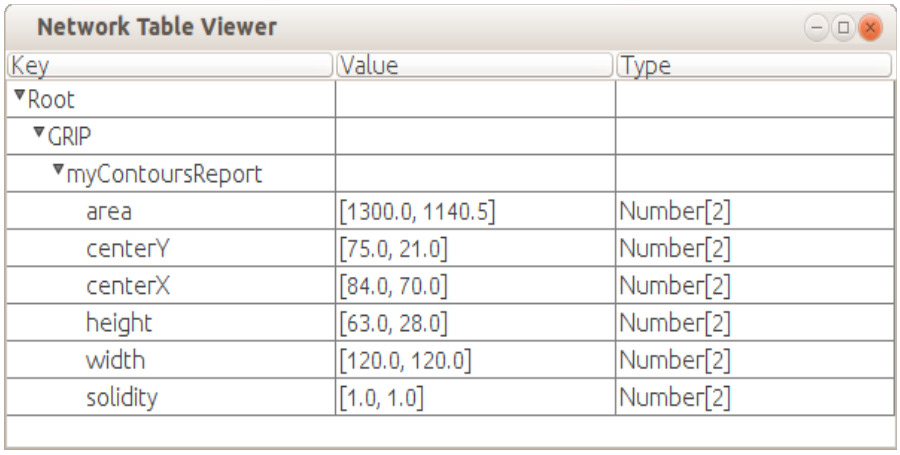
1. Find the Blue pencil using GRIP UI panel



2. Test Network tables publication
   - Add NTPublish CountoursReport panel and connect to “FindContours” panel

   - Run “OutlineViewer”
     - $ java -jar ~/wpilib/tools/OutlineViewer.jar
     - Start “Server” (leave entry fields blank)

       note: If Server not started get “Connection refused error” in GRIP.log



3. Capture GRIP NetworkTables publications in a Robot Program (linux_simulate build)
   - Run the GRIP “Find blue pencil” project with the NTPublishContoursReport panel connected to the “FindContours” panel as described above
     - make sure “Publish area” is checked in the NTPublishContoursReport panel
   - In Eclipse, write the following small “SampleRobot” program (e.g. in a FRC c++ project called GripTest):

```
class Robot: public SampleRobot {

public:
    std::shared_ptr<NetworkTable> table;
    Robot() {
        table = NetworkTable::GetTable("GRIP/myContoursReport");
    }
    void RobotInit() {
        static unsigned int old_area = 0;
        while (true) {
            std::vector<double> arr = table->GetNumberArray("area",
                    llvm::ArrayRef<double>());
            if (arr.size() > 0) {
                unsigned int new_area = (unsigned int) arr[0];
                if (new_area != old_area) {
                    std::cout << "area=" << new_area << std::endl;
                    old_area = new_area;
                }
            }
        }
    }
    void Autonomous() {
    }
    void OperatorControl() {
    }
    void Test() {
    }
};
START_ROBOT_CLASS(Robot)
```

     - Compile “GripTest” using linux_simulate build
       - Generates FRCUserProgram in project's linux_simulate sub-directory

- In a command shell, start up gazebo
  - $ gazebo
  - note: not actually using gazebo in this case but FRCUserprogram wont start unless Gazebo is running
- In a command shell start up “FRCUserProgram”
  - cd GripTest/linux_simulate
  - $ ./FRCUserProgram
  - Example output in the command shell:

….

area=1000

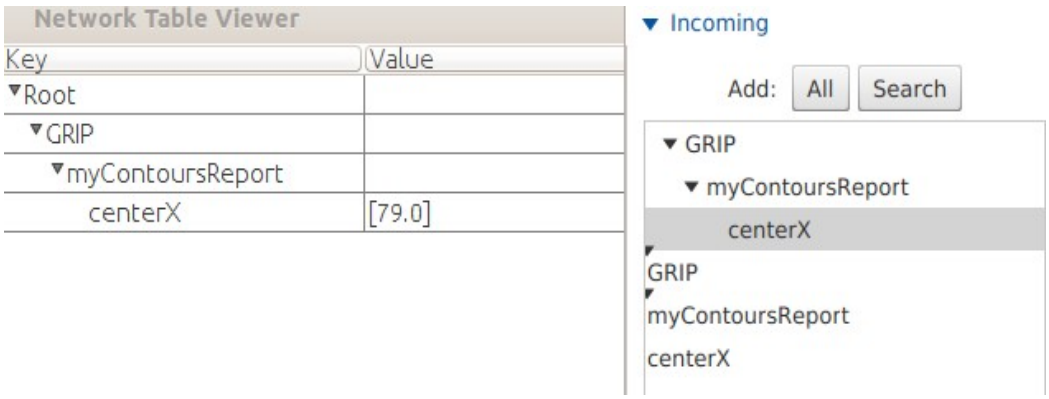
area=998

area=778

area=967

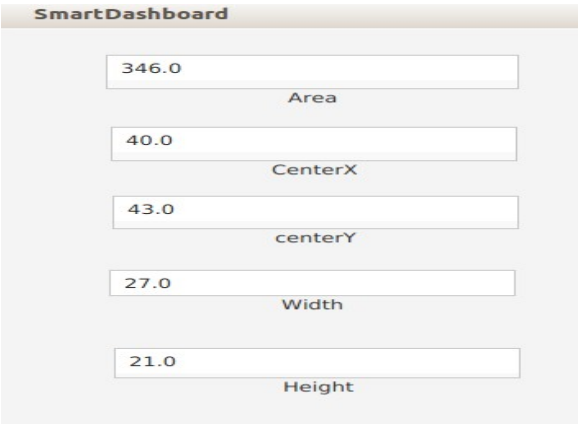
….

4. Show data from GRIP ContoursReport in SmartDashboard
   - Start SmartDashboard
   - Open Incoming list



- Drag an item (e.g. centerX) from “incoming” tree into panel (fails)
  - A Popup appears that offers a list of widget choices (e.g. number label,value meter etc.) but nothing gets generated in the panel

- Add an item manually (works)
  - open “+” ->Toolbox-general list and select “Array View” widget and drag to panel
  - double click array widget in panel and set Path to /GRIP/myContoursReport/centerX (etc.)
  - press labeler button and set label
    - “left” orientation doesn't resize column enough to show text but “bottom” seems to work (see below)



5. Build Smart Dashboard from source code
   - Was able to checkout the SFX project(s) from wpi git repository:

     https://usfirst.collab.net/sf/scm/do/listRepositories/projects.smartdashboard2/scm

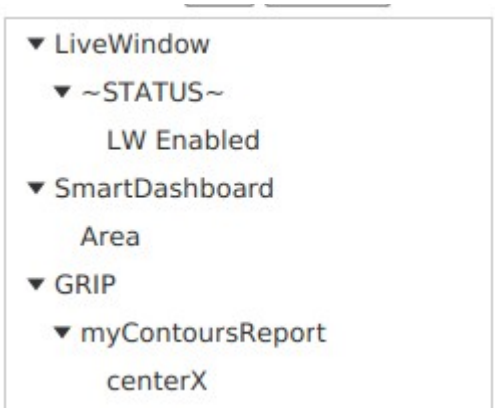
     but haven't yet been able to build it (note: needed to also get Netbeans 8.1 etc.)

     There seems to be some missing or changed URLs used in the build scripts (e.g. build couldn't find netbeans plugins at ..usfirst.collab.net:29418/netbeans_plugins -> repository doesn't exist)
6. Show GRIP data routed through FRCUserProgram in SmartDashboard
   - Added the following line to the sample robot program shown above that captures GRIP data:

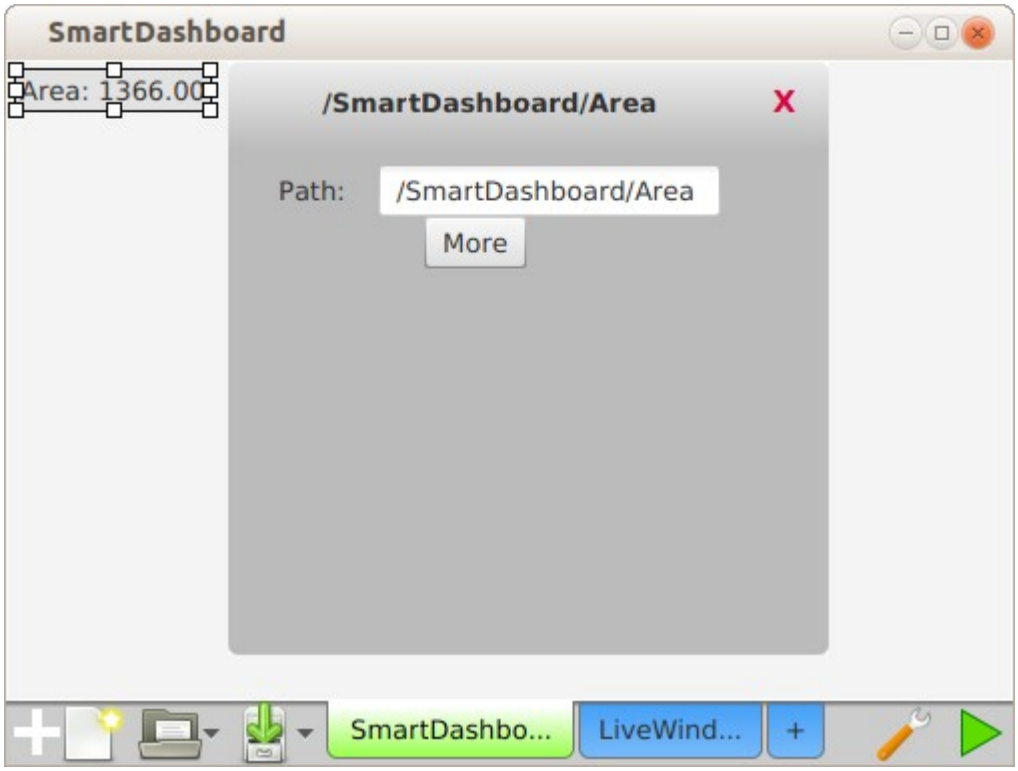
```
if (new_area != old_area) {
     std::cout << "area=" << new_area << std::endl;
  →  SmartDashboard::PutNumber("Area",new_area);
     old_area = new_area;
}
```

   - built and started FRCUserProgram (and gazebo)

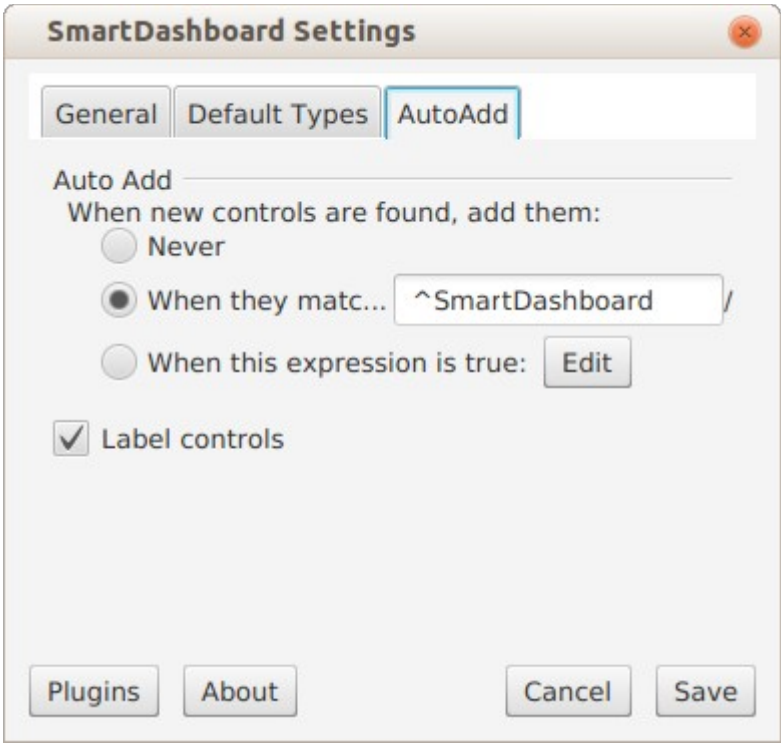
- Started SFX and opened “incoming” tab (now see SmartDashboard node in tree)



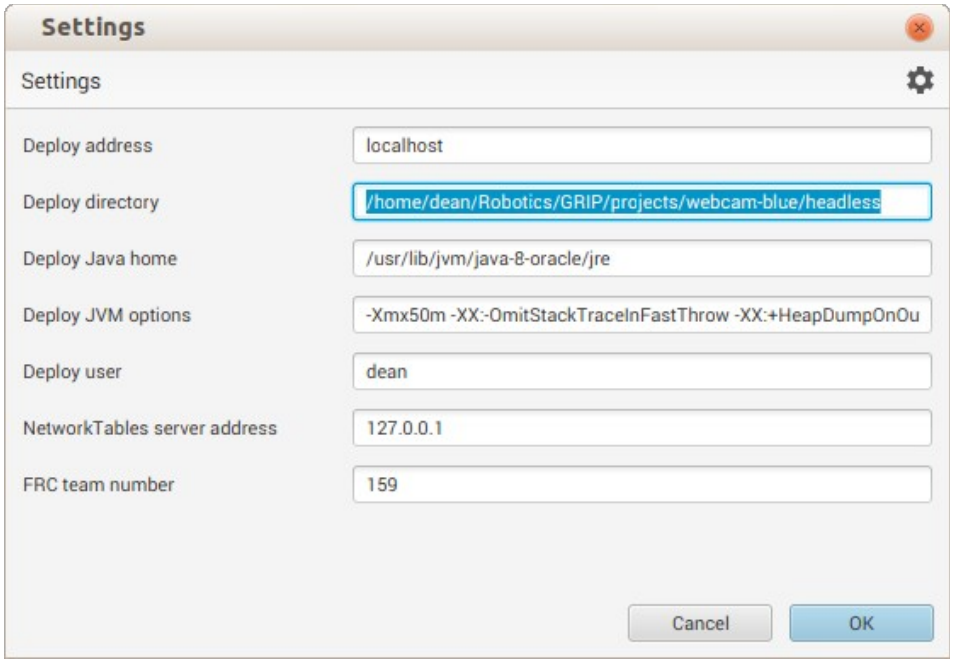
- Added a “number” label widget to canvas and set Path to /SmartDashboard/Area
    - GRIP “Area” numbers now get updated in data field



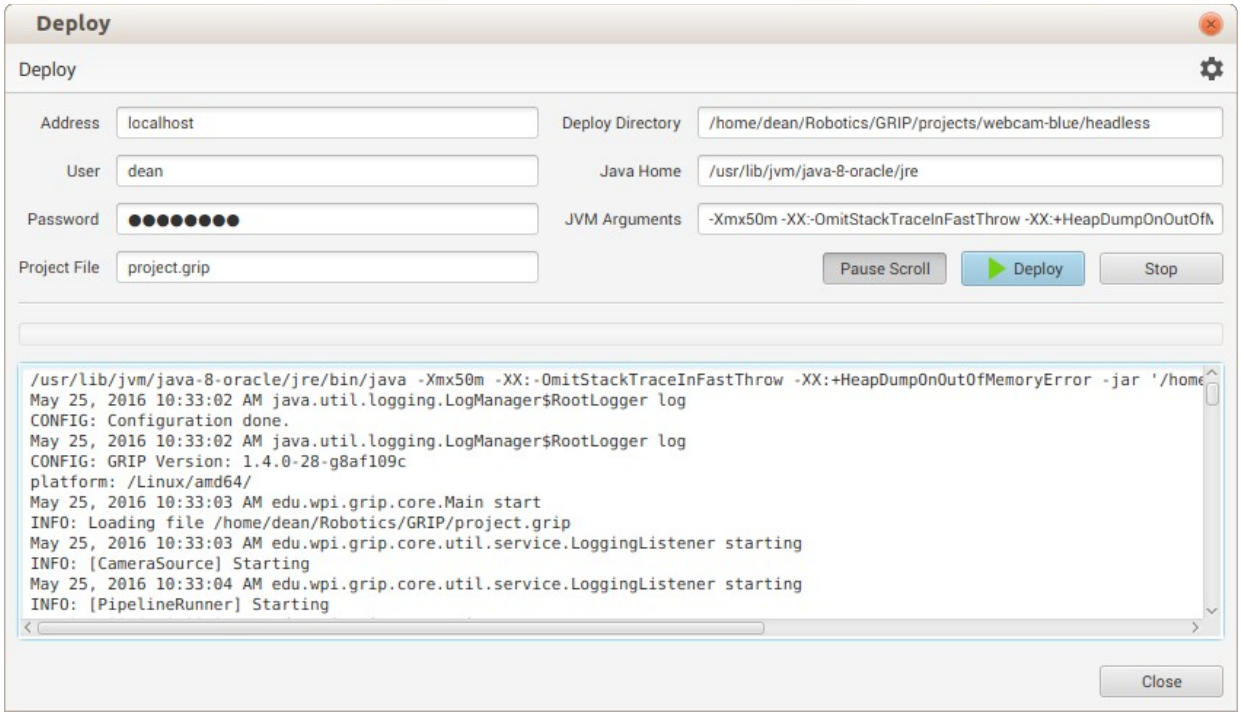
- Can also cause “Area” widget to be automatically created on startup by changing AutoAdd settings as shown:



7. Run GRIP ui from command line (i.e. not using gradlew)
    - $ gradlew :ui:installDist
        - creates startup shell script “ui” in GRIP/ui/build/install/ui/bin
    - $ ui/build/install/ui/bin/ui
        - starts up GRIP UI
    - $ ui/build/install/ui/bin/ui <path-to-project-file>
        - starts up GRIP UI and opens project file (e.g. project.grip)
8. Setup and run “Headless” GRIP project from command line (on localhost)
    - Start up GRIP UI and load project (using any method described above)
    - Open <menu>Tools->“Settings” dialog



- Press <menu>Tools->Deploy
  - Brings up Deploy dialog



  - Fill in password (for ssh access) and “Deploy Directory” field

    - Press the “Deploy” button
    - Creates a startup script (grip), a headless jar file (grip.jar) and project file (project.grip) in the directory specified by the “Deploy Directory” field
- Run headless grip with project file
    - $ cd projects/webcam-blue/headless/
    - $ grip
    - See same command line output as above (but no UI)

Comments

Pros

1. Intuitive user interface for testing the effect of common image processing operations
2. JavaFX UI Can be built and deployed on both Linux and Windows (Mac?)
3. Can run “headless” on Raspberry Pi, Roborio or Jetson (purportedly)
4. Supports ROS (gazebo) and networkTables (First/WPI) publication protocols
5. Can extract and publish geometry information from contour lists (center, width etc.)

Cons

1. Current implementation doesn't support generalized object recognition through “learning” algorithms
    - Uses basic image processing functions like “blur”, edge extraction etc.
    - Similar to ni-vision (IMAQ)
2. GRIP documentation says GPU isn't supported on Jetson board
3. On Raspberry Pi, problems with image publication require complicated workaround
4. Requires installing Java runtime on target platform
5. Java based (vs compiled binary), so processing speed may be an issue (?)

Problems

1. Can't figure out how to combine contours and original image
    - Want to see contours or convex hulls super-imposed on original image (e.g. for teleop

mode)

- Combining Original image with threshold output (as mask) is black everywhere except where mask is white
- UI doesn't permit connection between original image and “contours” output as second image for any CV operations (bit-wise xor etc.)
- No obvious tool to convert “Contours” or “Hulls” display back into RGB to make CV combine operators work

2. Image publication for display on Linux doesn't seem to work
    - May need GRIP code changes

TODO

1. Interface GRIP input source to Gazebo camera messages (DONE)
2. build and deploy on Raspberry Pi (DONE)
3. build and deploy on Jetson (ARM processor only)
4. Modifications WPI GRIP source project
    - Add CV block Combine images (annotate images)
    - Make URL source for mpeg entry block editable
    - Add block to use object recognition data (advanced OpenCV feature)
    - Modify Image Publish block to send output images to SmartDashboard or web browser on Linux

OpenCV

Version: OpenCV 3.1

target system: Ubuntu 14.04

references

http://www.askaswiss.com/2016/01/how-to-install-opencv-3-1-python-ubuntu-14-04.html

Building and Installation

1. Obtain source code via download
    - go to download site: http://opencv.org/downloads.htm
    - Select OpenCV for Linux/Mac

- Download opencv-3.1.0.zip (e.g. to ~)
- $ cd ~/
- $ unzip opencv-3.1.0.zip

2. Build
   - $ cd opencv-3.1.0
   - $ mkdir build && cd build
   - $ cmake ..
   - make -j6
3. Install
   - $ sudo make install
   - $ sudo /bin/bash -c 'echo "/usr/local/lib" > /etc/ld.so.conf.d/opencv.conf'
   - $ sudo ldconfig

Alternate build using opencv git rep

1. Obtain active source code
   - $ git clone https://github.com/Itseez/opencv
   - cd opencv
   - git checkout tags/3.1.0 -b 3.1.0
     - note: master branch is actually at 2.4 which produces compile errors unless CUDA is disabled
2. configure
   - mkdir build && cd build
   - ccmake ..
     - select options
       - YES: WITH_CUDA, WITH_OPENGL, WITH_MP ..
     - press c (configure)
     - press g (generate)
3. build

- make -j6 (takes ~ 1hr to build)

Test Installation

1. Build samples
   - $ cd ../samples
   - mkdir test && cd test
   - $ ccmake ../
     - set OPENCV_DIR to ~/opencv/build
     - set OPENCV_FOUND to YES
   - $ make -j6
2. Test
   - fix data paths so that samples will work using defaults
     - cd ~/opencv
     - cp -R data/* samples/data (training data needed for face-detect)
     - $ cd cpp
     - ln -s ~/opencv/samples/data ../data
   - $ cpp-example-facedetect data/lena.jpg
     - should see woman's head with face and eyes circled

Mjpg Streamer

Mjpg_steamer is a utility application that can be used to convert a set of images or the output of a USB WebCam into a TCP/IP data stream that can be input to an external application, web browser or GRIP (as an “ip camera”). It is a github project that can be build from source on most hardware/software platforms (including the Raspberry Pi)

1. Installation (Linux)
   - `Obtain libraries and source code`
     - `sudo apt-get install libjpeg8-dev`
     - `git clone https://github.com/jacksonliam/mjpg-streamer.git`
   - `Build from source and install`

- cd mjpg-streamer/mjpg-streamer-experimental
- make clean all

- Install
  - sudo make install

2. Tests
   1. pipe some jpg files in a directory to a web address
      - mjpg_streamer -i "input_file.so -f /home/dean/test/images -r -d 0.1" -o "output_http.so -w www -p 1180"
      - In Web browser (e.g. Firefox) capture and display images or image stream
        - e.g. Set web address to http://Ubuntu14.local:1180/?action=stream

Gazebo Simulation with GRIP

Setup

1. Modify simulation FRC Robot sdf file
   - Add a “save” element in the camera sensor block e.g:

```
<save enabled="true">
   <path>/tmp/shooter-camera</path>
</save>
```

   - When the simulation is run this will generate a set of jpg files to appear in the indicated directory

Run Simulation

2. start mjpg-streamer with arguments that will generate an mpeg stream using the files in the indicated directory
   - rm -fr /tmp/shooter-camera
   - mkdir -p /tmp/shooter-camera
   - mjpg_streamer -i "input_file.so -f /tmp/shooter-camera -r -d 0.1" -o "output_http.so -w www -p 1180"

3. Start GRIP as described above and set input source to a new ip camera with local URL

   e.g. http://Ubuntu14.local:1180/?action=stream

4. Start Gazebo Robot simulation as described previously (__ref__)

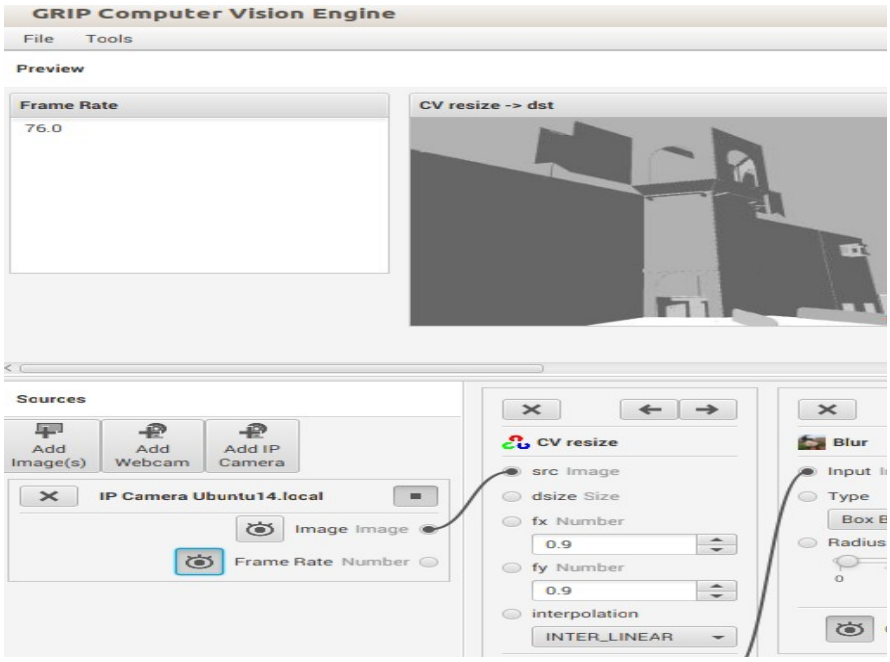
   - cd to FRC Robot project and run startup script e.g.:

     ```
     # start gazebo

     export SWMODEL=$SWEXPORTS/2016-Robot_Exported

     export GAZEBO_MODEL_PATH=$SWMODEL:
     $WPILIB/simulation/models:/usr/share/gazebo-6.5/models

     gazebo --verbose $SWMODEL/2016-Robot-field.world
     ```

   - Connect Joystick or Gamepad and start sim_ds
     - $sim_ds
   - Start FRCUserProgram
     - $ cd <path-to-robot-project>/linux_simulate
     - $ ./FRCUserProgram

First Results

1. Screenshot of Gazebo simulation of shooter camera on team 159's 2016 robot in FRC Stronghold Challenge field



Improvements

Add colored tape strips around tower targets

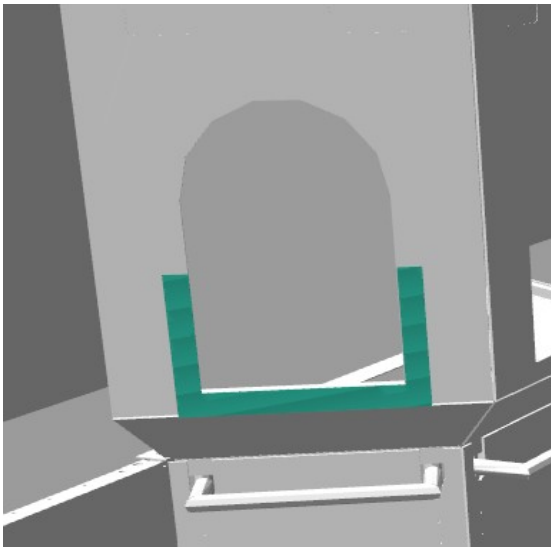
1. Create a tape pattern sdf file using solidworks exporter
   - create a sketch bottom = 18”, sides=12”, tape width=2”
   - Create a part then assembly from sketch
   - extrude sketch a slight amount (e.g. ~0.1”)
   - generate reference properties for origin, baseplane and axis
   - Use gazebo exporter tab “edit model” to set properties
     - Set name to TapePattern
     - set origin, baseplane and axis using reference objects created previously
     - Select extruded sketch as base joint
     - set color
   - Export model
2. Modify sdf file (hand edit)
   - remove collision and inertia elements and set gravity=0
     - otherwise object will drop to the ground when physics is enabled
   - edit ambient and diffuse colors if needed
     - for test purposes first try setting color to pure green(rgb=0,1,0)
3. Add TapePattern object to Robot simulation world file
   - Start Gazebo with world file containing robot and FRC field
   - Stop physics simulation
   - Insert a “TapePattern” object into gazebo
   - Use Position and Rotate modes to move TapePattern object to correct position below window in Turret
   - Select the TapePattern item in the world tree and Copy down

“pose” settings

- Exit gazebo
- Edit the robot world file to add a tape pattern object with the correct pose e.g.:

```
<include>
  <uri>model://TapePattern</uri>
  <pose>0.125288 -7.62138 2.11 0.000148 0.000257 0.52429</pose>
</include>
```

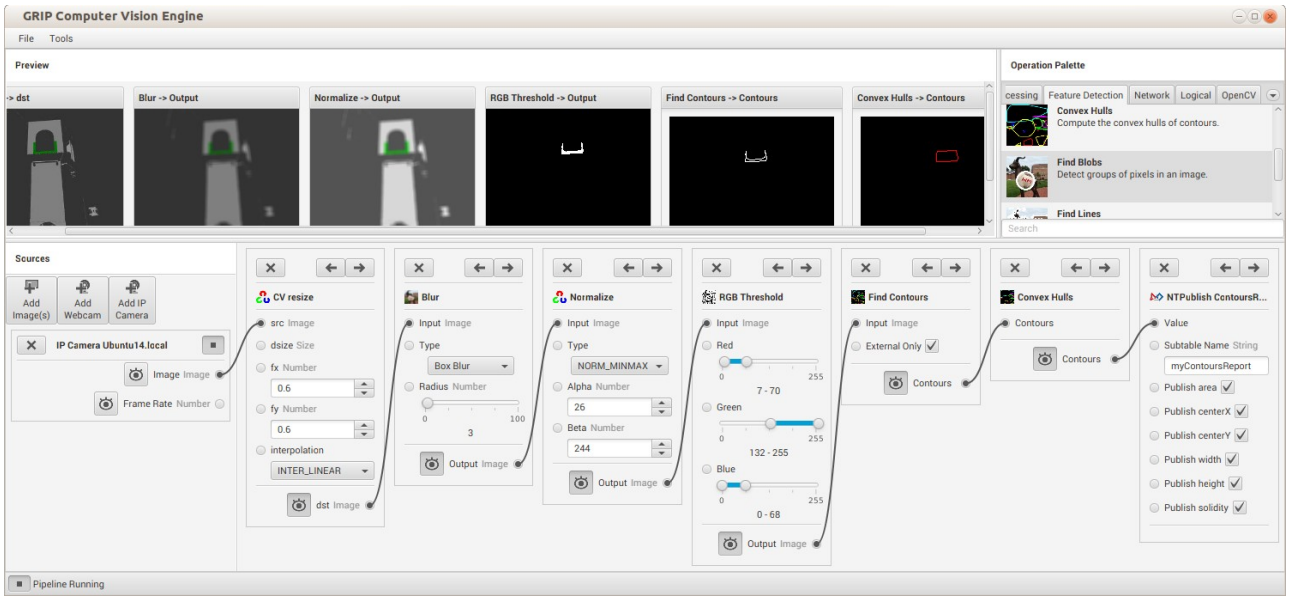
- Screenshot of tape pattern in left turret



Optimize GRIP Processing blocks to identify tape pattern target

1. Start up mgpg-streamer (monitor image dump directory)
2. Start up Gazebo Robot-FRC field simulation
3. start up GRIP and set source to ip camera (use local host as URL)
4. In Gazebo teleop mode drive robot to location of tower and adjust shooter angle until target is visible in GRIP input window
5. Set processing blocks in GRIP

- IP Camera input source (receiving data from mjpg-streamer)
- resize (optional)
- blur (kernel size=3)
- normalize (default settings)
- RGB Threshold (optimized for pure green tape pattern)
- find contours (external only)
- convex hulls
- NT Publish Contours



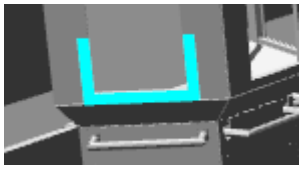
6. Deploy headless GRIP project on Raspberry Pi and compare with GRIP running natively on Ubuntu
   - Connect up Raspberry Pi to network
   - Open up net-tables viewer as client or sfx
   - In GRIP UI Deploy headless project to raspberry Pi
     - In tools settings dialog set network Tables server address to ip or domain name of Ubunto host (e.g. Ubuntu14.local)
     - set deploy address to raspberrypi.local
     - set user name to pi

- enter password (raspberry)
- Press tools->Deploy

- Monitor network tables client window and verify that values for area centerX etc. get updated as before (i.e. when running GRIP from UI in Ubuntu)
  - qualitatively it seems to be the same
  - need to figure out how to measure bandwidth

Optimize lighting to match FRC field environment

1. Is it possible to add a light source to the Robot ?
   - No: not with latest version of Gazebo
   - But is a feature request
2. Is illumination supported as an sdf material property ?
   - Yes: set “emissive” property in material element
   - e.g. <emissive>0.0 1.0 1.0 1</emissive>

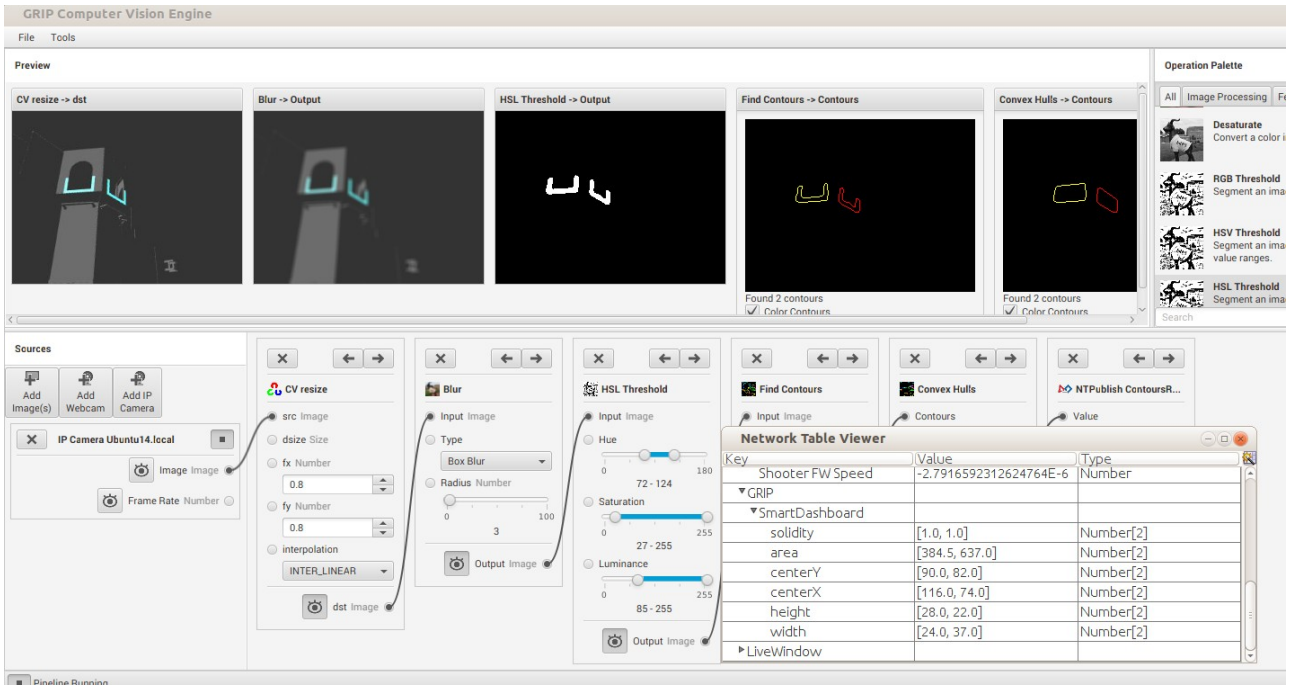


3. Determine best color for tape pattern
   - ambient and diffuse =1,1,1,1
   - <emissive>0.2 1.0 1.0 1</emissive>
4. Is it possible to modify the simulated camera properties such as “exposure” and “brightness” ?
   - Need to check
   - Can emulate effect by darkening scene lighting
5. Modify Gazebo ambient lighting properties be better emulate inside environment
   - Disabled global shadows
   - darkened field environment by changing rgb diffuse color values from 0.8 to 0.5

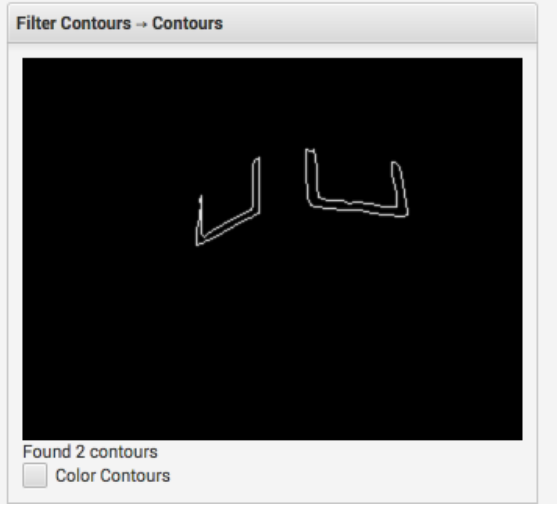
Improved Results

1. Added second tape pattern to center turret

2. Processing blocks
    - Got better contour isolation by using HSL vs RGB threshold block
    - Removed Normalize block
3. Screenshot of GRIP UI with improved target lighting



4. Comparison with real FRC field results
    - reference: http://wpilib.screenstepslive.com/s/4485/m/50711/l/481750-using-grip-for-the-2016-game



TODO

1. `Modify Solidworks Gazebo exporter plugin to provide entry fields for jpeg output path to avoid having to hand edit sdf files`

Raspberry Pi

Raspberry Pi - Setup

1. Hardware version
    - Raspberry Pi 3 (model B)
    - purchased from NewEgg ($35)
2. Software version
    - Raspbian Jessie-Lite
3. Cables & other hardware
    - micro SD Card reader
    - micro SD card
        - started with a 4G card but ran out of room when installing OpenCV so got a 16 G card
        - recommend 16G or greater
    - USB –> micro USB cable
    - Standard Ethernet cable

Installation of Operating system (from Ubuntu host)

1. Download Raspbian Jessie Lite OS
    - source: https://www.raspberrypi.org/downloads/raspbian/
    - download file: 2016-05-27-raspbian-jessie-lite.zip
    - $ unzip 2016-05-27-raspbian-jessie-lite.zip
    - creates: 2016-05-27-raspbian-jessie-lite.img (size:1387266048)
2. Install OS on SD card
    - Instructions: https://www.raspberrypi.org/documentation/installation/installing-images/linux.md

- Determine the partition number for the SD card reader
    - Remove the SD card reader
    - $ df -h (note: the base configuration info)
    - Insert the SDCard reader with u-Sdcard installed
    - $ df -h
    - These are the new devices which appear over the base configuration

      /dev/sdc1 818M 5.7M 812M 1% /media/dean/D8DE-D3EE

      /dev/sdc2 976M 135M 791M 15% /media/dean/ecabb50d-36f7-4ec3-b449-11c76e8467f5
    - The full partition for the card is therefore “sdc” (following the instructions in the ref)
        - note: when this procedure was repeated using a 16G card the partition was “sdg” instead

3. Burn the image onto the SD card
    - $ umount /dev/sdc1
    - $ umount /dev/sdc2 (if present)
    - $ sudo dd bs=4M if=2016-05-27-raspbian-jessie-lite.img of=/dev/sdc
    - $ sync
4. Check validity of image on card (optional)
    - $ sudo dd bs=4M if=/dev/sdc of=from-sd-card.img
        - ls -l: -rw-r--r-- 1 root root 3980394496 Jun 3 09:35 from-sd-card.img
    - $ sudo truncate --reference 2016-05-27-raspbian-jessie-lite.img from-sd-card.img
        - ls -l: -rw-r--r-- 1 root root 1387266048 Jun 3 09:37 from-sd-card.img
    - $ diff -s 2016-05-27-raspbian-jessie-lite.img from-sd-card.img
        - output: Files 2016-05-27-raspbian-jessie-lite.img and from-sd-card.img are identical
5. Connect up the raspberry Pi
    - Insert micro SD card in slot underneath Pi card
    - connect Ethernet cable from Pi to Host (or switch)

- Connect USB power cable from Pi to USB port on host
    - The board should now boot the OS

6. Test connection from Host to Pi
    - $ ping raspberrypi.local
        - should get responses
    - $ ssh -l pi raspberrypi.local
        - enter “raspberry” as password
        - Should see command prompt: pi@raspberrypi:~ $
7. Configure Pi so that a password isn't required on login

    On Localhost:

    - `$ ssh-keygen`
        - `enter a passphase`
    - `$ cat ~/.ssh/id_*.pub | ssh pi@raspberrypi.local 'cat > .ssh/authorized_keys'`
        - `note: first make sure that Pi already has a .ssh directory`
    - $ ssh -p pi@raspberry.local
        - On first access a dialog asks for passphrase (use same as entered in ssh-keygen)
        - On subsequent logins passphase isn't required

Raspberry Pi - GRIP

1. Reference
    - https://github.com/WPIRoboticsProjects/GRIP/wiki/Running-GRIP-on-a-Raspberry-Pi-2
2. Obtain required library files by pressing the links in the above reference
    - libntcore.so,libstdc++.so.6
3. Download system libraries
    - Create destination directory on pi
        - log onto Pi (ssh -l pi raspberrypi.local)
        - pi@raspberrypi:~ mkdir vision
        - pi@raspberrypi:~ cd vision

- pi@raspberrypi:~ mkdir grip

- Install Java 8 on Pi
  - pi@raspberrypi:~ sudo apt-get update && sudo apt-get install oracle-java8-jdk
- scp libraries from Host to Pi (in a separate Host command shell)
  - $ scp libntcore.so pi@raspberrypi.local:/home/pi/vision/grip/libntcore.so
  - $ scp libstdc++.so.6 pi@raspberrypi.local:/home/pi/vision/grip/libstdc++.so.6

4. Build and Install **mpeg-streamer** to fix reported bug with running USB cameras on Pi
   - Obtain libraries and source code
     - Log onto Pi (ssh -l pi raspberrypi.local)
     - pi@raspberrypi:~ sudo apt-get update && sudo apt-get install git
     - pi@raspberrypi:~ sudo apt-get update && sudo apt-get install cmake
     - pi@raspberrypi:~ sudo apt-get update && sudo apt-get install libjpeg8-dev
     - pi@raspberrypi:~ cd vision
     - pi@raspberrypi:~ git clone https://github.com/jacksonliam/mjpg-streamer.git
   - Build from source and install
     - pi@raspberrypi:~ cd mjpg-streamer/mjpg-streamer-experimental
     - pi@raspberrypi:~ make clean all
     - pi@raspberrypi:~ sudo make install
   - Test USB Camera connection
     - Connect USB camera to one of 4 USB slots on Pi
     - should see “video0” when camera is connected
   - Run mpeg-streamer from a command line to test
     - pi@raspberrypi:~ mjpg_streamer -o "output_http.so -w ./www

```
-p 1180" -i "input_uvc.so -d /dev/video0 -f 15 -r 640x480
-y -n"
```

```
Output:

 MJPG Streamer Version.: 2.0

 i: Using V4L2 device.: /dev/video0

..
```

- View Stream from web browser
  - Open a tab in Firefox and enter the following URL: http://raspberrypi.local:1180/?action=stream
  - Should see an image from the camera in the tab
- View Stream on host using a Python script
  - file stream-server.py:

```
#!/usr/bin/python
import cv2
import urllib
import numpy as np
stream=urllib.urlopen('http://raspberrypi.local:1180/?action=stream')
bytes=''
while True:
    bytes+=stream.read(1024)
    a = bytes.find('\xff\xd8')
    b = bytes.find('\xff\xd9')
    if a!=-1 and b!=-1:
        jpg = bytes[a:b+2]
        bytes= bytes[b+2:]
        i = cv2.imdecode(np.fromstring(jpg,dtype=np.uint8),cv2.CV_LOAD_IMAGE_COLOR)
        cv2.imshow('camera',i)
        if cv2.waitKey(1) == 27:
            exit(0)
```

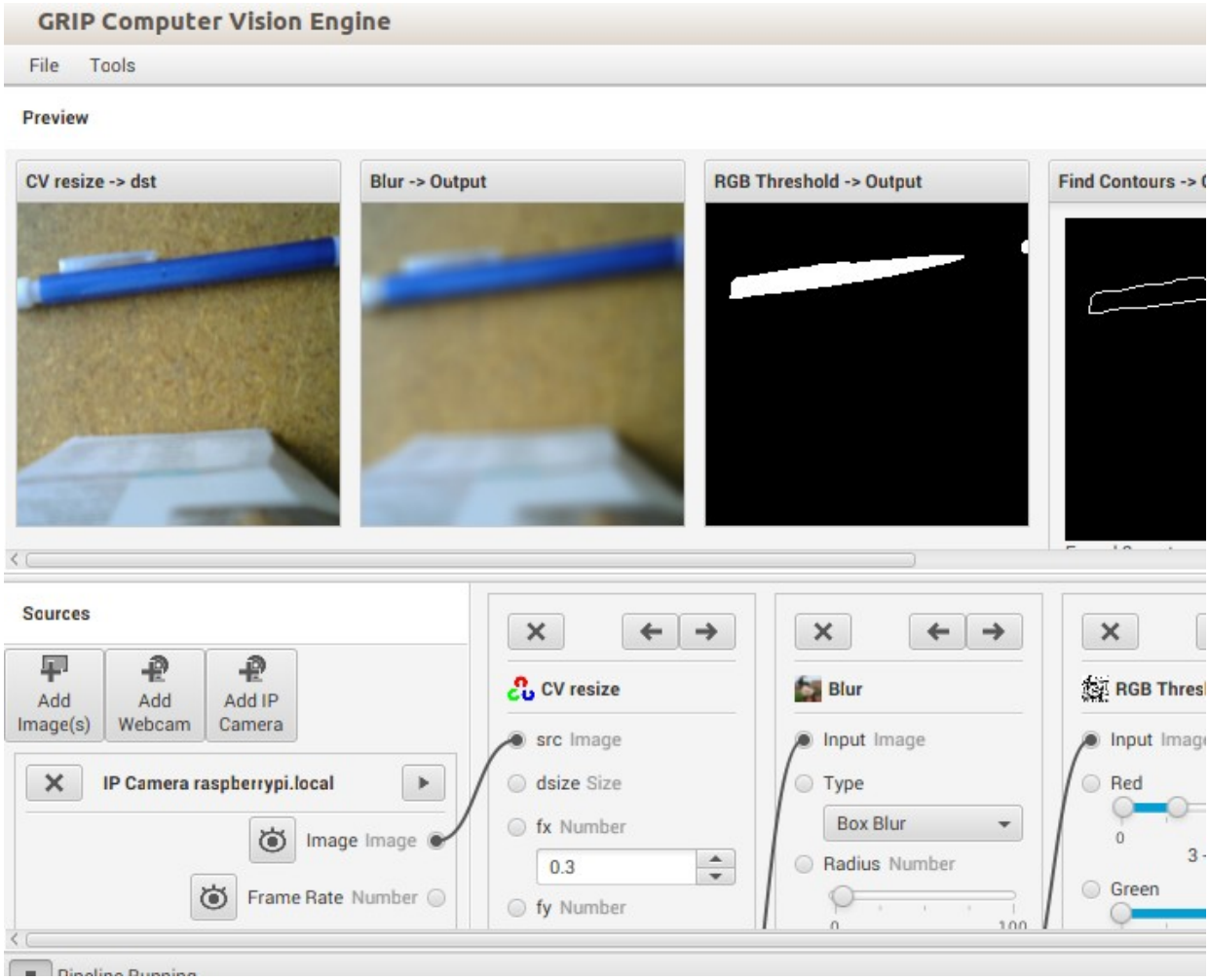
  - On Pi, start mpeg-streamer
  - On Host, $ python stream-server.py

5. Test Image stream from Pi running GRIP on host

   On Pi start up mpeg streamer as described in the previous section

   - pi@raspberrypi:~ mjpg_streamer -o "output_http.so -w ./www -p 1180" -i "input_uvc.so -d /dev/video0 -f 15 -r 640x480 -y -n"

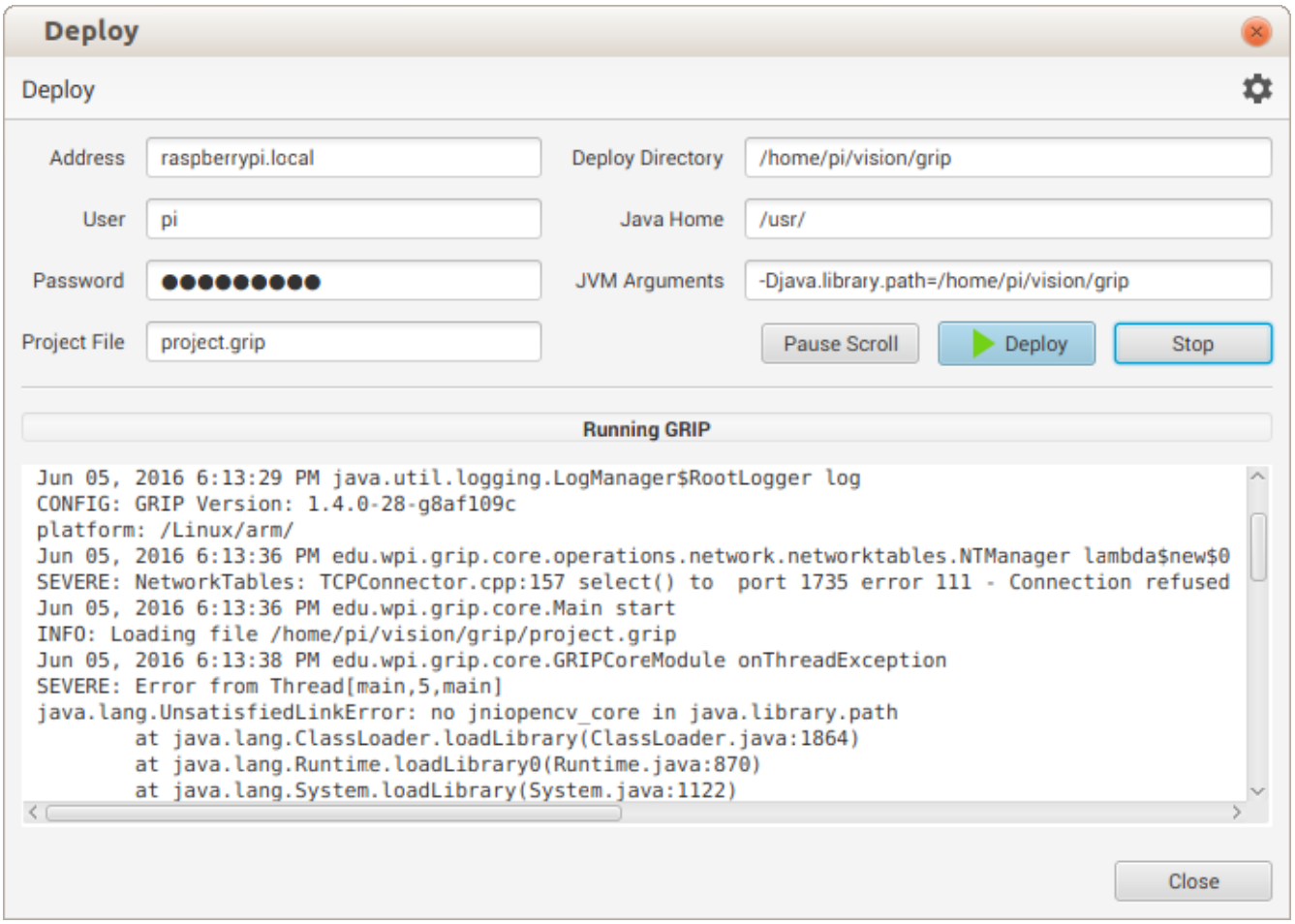
- On Host, start up GRIP with project using one of the methods described above
  - ui/build/install/ui/bin/ui projects/webcam-blue/webcam-blue.grip
- In GRIP UI, replace the WebCam input with “Add IP Camera” input
  - Set URL in Ip Camera dialog to: http://raspberrypi.local:1180/?action=stream
- Should see same processing behavior as when USB camera was connected directly to the Host



6. Verify and try to fix USB camera bug on Pi
   - Was able to get normal USB “WebCAM” GRIP input source to work after installing `fswebcam as described in:` https://www.raspberrypi.org/documentation/usage/webcams/
   - `Couldn't see images but got ContoursReport updates in ntables`

- When trying to run mpeg-streamer and a headless GRIP project that was using the webcam simultaneously mpeg-streamer aborted with “illegal image format error”
  - mpeg-streamer worked again once GRIP project was killed

7. Deploy Host GRIP project on Pi (following instructions in ref)
   - On Host start up GRIP and load project
   - Change Source to IP Camera (URL=http://raspberrypi.local:1180/?action=stream) as described in the previous section
   - Set Deploy options as shown



- Pressed Deploy button
  - copies grip.jar (1.4.0-28-g8af109c) and project.grip to ~/vision/grip on Pi
  - fails with error: “no jniopencv_core in java.library.path”

        - without -Djava.library.path get unsatisfied link error for ntcore
    - Same problem observed when running from command line logged into pi
    - libjniopencv_core.so etc. appears to be part of javacv (and javacpp)
    - But haven't been able to locate a pre-built binary for raspberry (arm) so will try to build them from source

8. Build javacpp and javacv (method 1) FAILS
    - reference: https://github.com/t-oster/VisiCam/wiki/Raspberry-Pi-installation-on-Raspbian
    - `pi@raspberrypi:~` sudo apt-get install maven sed
    - git clone https://github.com/bytedeco/javacpp.git
    - git clone https://github.com/bytedeco/javacv.git
    - build javacpp
        - mvn clean install -f javacpp/pom.xml -Dplatform.name=linux-arm
        - build completes
    - build javacv
        - mvn clean install -f javacv/pom.xml -Dplatform.name=linux-arm
        - build fails
9. Build javacpp and javacv (method 2) WORKS
    - reference: http://alltechanalysis.blogspot.com/2015/09/installing-opencv-30-and-javacv-on.html
    - `pi@raspberrypi:~ cd`
    - `pi@raspberrypi:~` sudo apt-get install maven
    - `pi@raspberrypi:~` git clone https://github.com/bytedeco/javacpp-presets
    - `pi@raspberrypi:~` git clone https://github.com/aravindrajasekharan/JavaCpp-OpenCv-Linux-Arm.git
    - `pi@raspberrypi:~` cd JavaCpp-OpenCv-Linux-Arm
    - `pi@raspberrypi:~ cp` cppbuild.sh `../javacpp-presets/opencv`
    - `pi@raspberrypi:~` cd `../javacpp-presets`

- pi@raspberrypi:~ export PLATFORM=linux-arm
  - instruction is missing in reference
- pi@raspberrypi:~ git checkout tags/1.1 -b 1.1
  - instruction is missing in reference
  - compatible with opencv version 3.0
  - note: latest version (1.2) fails to build
- pi@raspberrypi:~ ./cppbuild.sh
  - builds local opencv 3.0 (succeeds)
  - takes ~1/2 hour
- pi@raspberrypi:~ mvn clean install -DskipTests -Dplatform.name=linux-arm
- After ~ 1hour
  - [INFO] JavaCPP Presets ..... SUCCESS [45.900s]

    [INFO] JavaCPP Presets for OpenCV ..... SUCCESS [48:05.816s]

    [INFO] JavaCPP Presets for FFmpeg ..... FAILURE [0.922s]
  - [ERROR] Failed to execute JavaCPP Builder: Could not parse "libavutil/avutil.h": File does not exist
  - But found "libavutil/avutil.h" in several sub-directories
    - e.g. ./opencv/cppbuild/linux-arm/opencv-3.0.0/3rdparty/include/ffmpeg_/libavutil/avutil.h
- However,jni libraries seem to have been built anyway
  - e.g. ./opencv/target/classes/org/bytedeco/javacpp/linux-arm/libjniopencv_core.so exists etc.
- copied /opencv/target/classes/org/bytedeco/javacpp/linux-arm/*.so /usr/lib
- re-ran grip launcher script (start_grip.sh) from ~/vision
  - export LD_LIBRARY_PATH=/home/pi/vision/grip; java -jar

/home/pi/vision/grip/grip.jar
/home/pi/vision/projects/project.grip &

- no longer get “no jniopencv_core in java.library.path”
- but now get the following errors
  - SEVERE: The CV resize operation did not perform correctly.

    java.lang.RuntimeException: /home/pi/javacpp-presets/opencv/cppbuild/linux-arm/opencv-3.0.0/modules/imgproc/src/imgwarp.cpp:3208: error: (-215) ssize.area() > 0 in function resize

    several of these then ..
  - Jun 09, 2016 3:45:09 AM edu.wpi.grip.core.Main onExceptionClearedEvent

    INFO: Exception Cleared Event

    Jun 09, 2016 3:45:09 AM edu.wpi.grip.core.operations.network.networktables.NTManager lambdanew0

    SEVERE: NetworkTables: TCPConnector.cpp:157 select() to 127.0.0.1 port 1735 error 111 - Connection refused

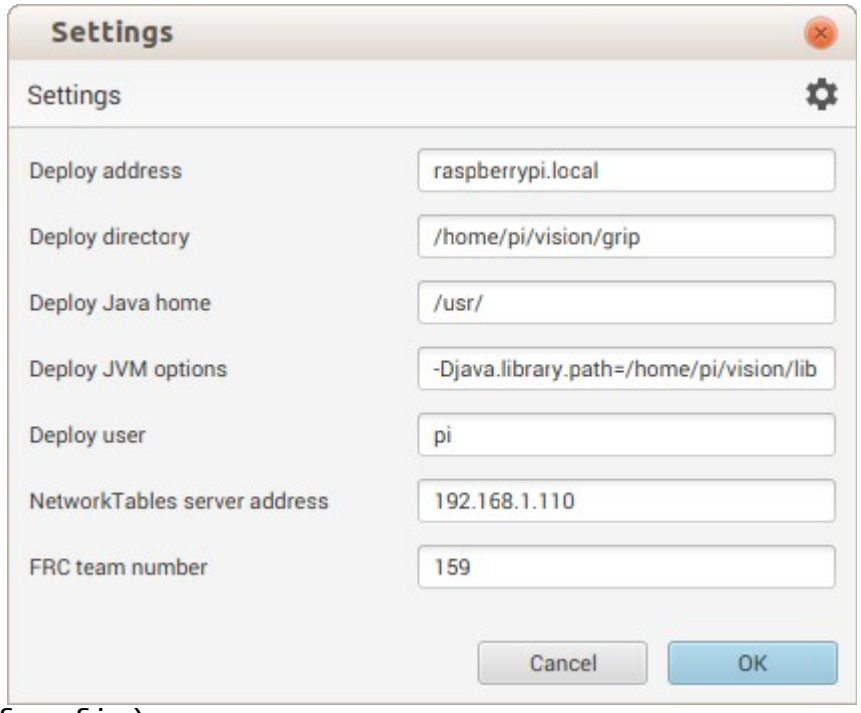
    this error keeps repeating until process is killed

- Got “Deploy” in host grip GUI to run (but generate similar errors) by changing JVM Arguments to:

  -Djava.library.path=/home/pi/vision/grip:/usr/lib
  - note: missing jni libraries are now in /usr/lib
- Fix for “connection refused” error
  - On Pi modified project.grip file to change ip of “PublishAddress” element to that of Linux host (as determined by ifconfig)
  - started Network Tables server on host
  - started mpeg-streamer on Pi
  - started start_grip.sh on Pi

- Now, no longer get connection refused errors but Network Tables GUI shows invalid data in fields “[]”
- problem was that blue pencil was missing from camera view
- Repositioned pencil and now see values being updated in Network Tables window !

- Set PublishAddress from GRIP GUI
  - Open “settings” dialog and change “Network Tables Server Address to ip of Host (i.e as determined by running ifconfig)



  - can then directly deploy the GRIP project to Pi without needing to hand-edit the project.grip file
  - When the “Deploy” button is pressed network tables are updated by the “headless” GRIP project running on the Pi and the image “pipeline” to the UI GRIP running on the host is stopped
  - After stopping the headless GRIP project from the “Deploy” dialog window press the small button labled “Pipeline Stopped” in the lower-left corner of the GRIP interface to resume processing from the UI Grip application

10. Summary of how to set up and run a headless GRIP project on the Raspberry Pi

    1. Get Pi resources
        - Install raspian-jessie-lite on Pi SD card
        - Install java 8
        - Obtain or build libjniopencv_core.so etc. for linux-arm
        - place these and downloaded arm versions of libntcore.so,libstdc++.so.6 somewhere accessible from java.library.path (e.g. /usr/lib ~/vision/grip)
        - Build or obtain mpeg_streamer utility and install it
    2. Debug GRIP project on Host
        - Connect USB camera to Pi and point it at a suitable target
        - Run mpeg-streamer on Pi (see above for arguments)
        - Start GRIP (with GUI) on Host
        - Add IP camera block as GRIP input
            - URL=http://raspberrypi.local:1180/?action=stream
        - Add grip filters etc. to isolate regions of interest in camera images
        - Add Contours block
        - Start up Network Table Server on Host (or run FRCUserProgram which will instantiate a server)
        - Add Publish Contours report as last block in GRIP chain
    3. From GRIP on Host, Deploy GRIP project to Pi
        - Open “settings” dialog and change “Network Tables Server Address to ip of Host (i.e as determined by running ifconfig)
        - Set Deploy arguments as shown above and press “Deploy”
    4. (Alternative) Run headless GRIP project on Pi
        - Log onto Pi and run “start_grip.sh”
    5. Route network info to Eclipse program running on host

- Use Eclipse GRIP Test program described 9.5 above
- Stop Network Tables server (otherwise get the following error)
  - NT: ERROR: bind() failed: Address already in use (TCPAcceptor.cpp:102)
- Run project executable from command line (linux_simulate/FRCUserProgram) or eclipse (run as linux_simulate ..)
  - Appears to start up a new Network Tables server
- (Optional) Start up Network Tables as Client to view contours report publications in external window

6. Addendum:
   - To ease the pain of setting up a new SD card that includes support for GRIP I've placed a number of tar files in the Team159:MentoRepository at

Raspberry Pi – OpenCV

1. Version
   - 3.0 (or 3.1)
2. Reference
   - http://www.pyimagesearch.com/2015/10/26/how-to-install-opencv-3-on-raspbian-jessie/
3. Install dependent libraries
   - Log onto Pi (`ssh -l pi raspberrypi.local`)
   - From reference instructions
     - pi@raspberrypi:~ sudo apt-get update
     - pi@raspberrypi:~ sudo apt-get upgrade
     - pi@raspberrypi:~ sudo rpi-update
     - pi@raspberrypi:~ sudo reboot
     - pi@raspberrypi:~ sudo apt-get install libjpeg-dev libtiff5-dev libjasper-dev libpng12-dev
     - pi@raspberrypi:~ sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev libv4l-dev
     - pi@raspberrypi:~ sudo apt-get install libxvidcore-dev libx264-dev
     - pi@raspberrypi:~ sudo apt-get install libatlas-base-dev gfortran

- pi@raspberrypi:~ sudo apt-get install python2.7-dev python3-dev
- pi@raspberrypi:~ sudo apt-get install libgtk2.0-dev (do we need this for headless processing ?)

4. Obtain opencv source

   (This procedure differs from reference instructions which uses a tar file)

   - Log onto Pi
   - pi@raspberrypi:~ git clone https://github.com/Itseez/opencv.git
   - pi@raspberrypi:~ cd opencv
   - pi@raspberrypi:~ git checkout tags/3.0.0 -b 3.0.0
     - note: for 3.1 version use: :~ git checkout tags/3.1.0 -b 3.1.0

5. Setup Python (2.7)

   - pi@raspberrypi:~ wget https://bootstrap.pypa.io/get-pip.py
   - pi@raspberrypi:~ sudo python get-pip.py
   - pi@raspberrypi:~ sudo pip install virtualenv virtualenvwrapper
   - pi@raspberrypi:~ sudo rm -rf ~/.cache/pip
   - added to bottom of ~/.profile

     export WORKON_HOME=$HOME/.virtualenvs
     source /usr/local/bin/virtualenvwrapper.sh
   - pi@raspberrypi:~ source ~/.profile
   - pi@raspberrypi:~ mkvirtualenv cv
     - command prompt should now be: (cv)pi@raspberrypi~
     - note: after rebooting etc. to get back into the “cv environment” enter “workon cv” (ctrl-d to exit)
   - (cv)pi@raspberrypi:~ pip install numpy
     - takes about 15-20 minutes to complete

6. Build opencv

   - (cv)pi@raspberrypi:~ mkdir release && cd release
   - (cv)pi@raspberrypi:~ ccmake ..

- kept most defaults except turned off CUDA and enabled openMP

- (cv)pi@raspberrypi:~ make -j4
  - took ~1hr to build
  - note: after build disk usage increased from 49->93% on 4G SD card -(need bigger card)
    - So purchased a 16G uSD card ($11) and repeated everything up to this point
    - df now shows 25% usage

7. Finishing the installation
   - (cv)pi@raspberrypi:~ cd ~/.virtualenvs/cv/lib/python2.7/site-packages/
   - (cv)pi@raspberrypi:~ ln -s /usr/local/lib/python2.7/site-packages/cv2.so cv2.so
8. Verify installation
   - (cv)pi@raspberrypi:~ python

     >>> import cv2

     >>> cv2.__version__

     '3.0.0'

     >>>
   - Use Ctrl-d to exit Python
9. Python Utility functions
   - Display image data sent from Pi in host window

```
#!/usr/bin/python
# run on Ubuntu host
# opens a window and updates whenever a new image is sent from client

import socket
import cv2
import numpy

def recvall(sock, count):
    buf = b''
    while count:
        newbuf = sock.recv(count)
        if not newbuf: return None
        buf += newbuf
        count -= len(newbuf)
    return buf

TCP_IP = 'Ubuntu14.local' # DHCP hostname or static ip of host
```

```
TCP_PORT = 5001

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((TCP_IP, TCP_PORT))
s.listen(True)
cv2.namedWindow("SERVER")
while True:
  conn, addr = s.accept()
  length = recvall(conn,16)
  stringData = recvall(conn, int(length))
  data = numpy.fromstring(stringData, dtype='uint8')
  decimg=cv2.imdecode(data,1)
  cv2.imshow('SERVER',decimg)
  if cv2.waitKey(25) == 27:
    break
s.close()
cv2.destroyAllWindows()
```

- Send opencv image (captured from USB camera)to Host for display

```
#!/usr/bin/python
# runs on Pi
import socket
import cv2
import numpy

TCP_IP = 'Ubuntu14.local' # DHCP hostname or static ip of host
TCP_PORT = 5001

sock = socket.socket()
sock.connect((TCP_IP, TCP_PORT))

capture = cv2.VideoCapture(0)
ret, frame = capture.read()

encode_param=[int(cv2.IMWRITE_JPEG_QUALITY),90]
result, imgencode = cv2.imencode('.jpg', frame, encode_param)
data = numpy.array(imgencode)
stringData = data.tostring()

sock.send( str(len(stringData)).ljust(16));
sock.send( stringData );
sock.close()
cv2.waitKey(0)
```

Robot Image processing Code

Vision interface class

specifications

1. Front end (API for Robot program) should be the same for whatever back end processing engine is used (e.g. GRIP, ni-

vision, native openCV, advanced image recognition tools etc.)

2. Front end API should not be different when compiling for simulation or deploy targets

Front end Interface API

1. Configure API with camera and target property information (width, height etc)
    - It may be possible to obtain some camera information automatically
2. Receive target information from back-end contained in an array of structures or class objects containing:
    - Identity code for each object recognized
    - distance from camera (in feet or meters), left right offset angle, inclination angle (in degrees or radians)

Back end

1. Capture net tables data generated from processing engine (e.g. GRIP)
2. produce target information data structure(s) that contain real world distance and position information etc.
    - Use Configuration information to convert between screen space and real world dimensions

Simulation Tasks

1. Determine distance from target based on size of target in image
    - verify using lidar data
2. Determine target angle adjustment based on vertical position of target in image
3. determine robot position adjustment based on horizontal position of target in image
4. Use target data structure to set robot position, shooter angle and flywheel speed to place high goal in (simulation) autonomous mode