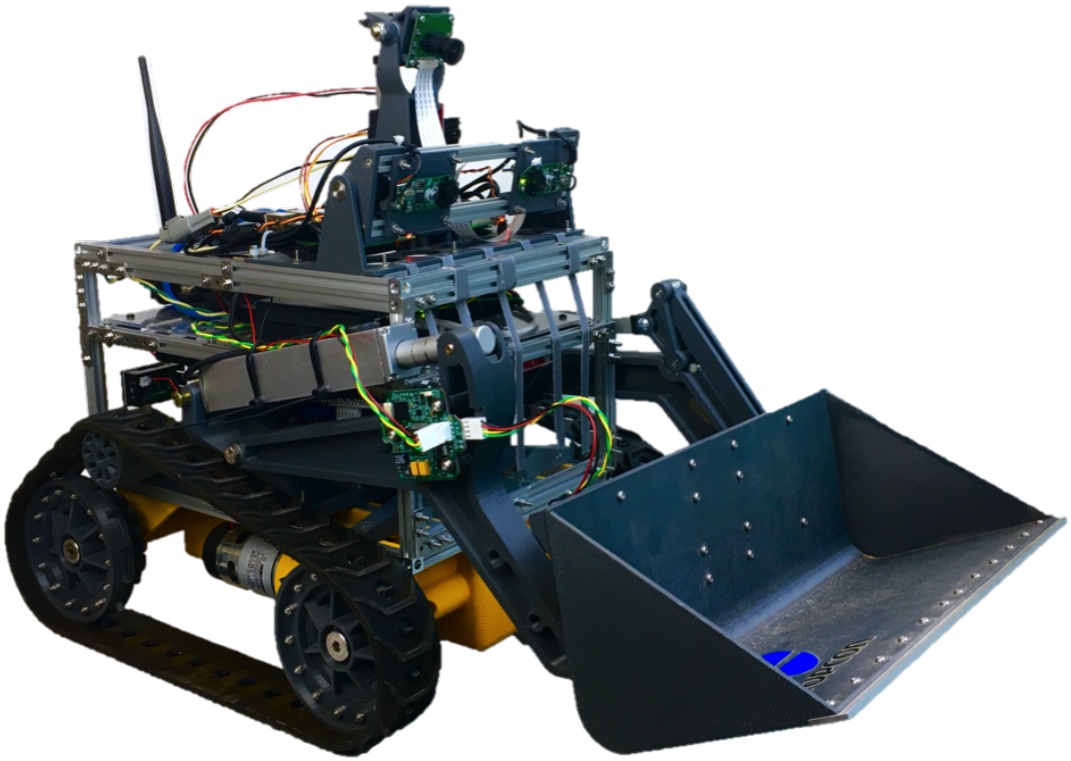
Autonomous Bulldozer Development

Project Sambuca • Technical Report

James M Trombadore, Donald Avansino, David Kooi, Kiefer Selmon



A UC Santa Cruz Baskin School of Engineering Senior Design Project Sponsored by Topcon

TOPCON Baskin Engineering UC SANTA CRUZ

July 1, 2018

Contents

List of Figures

1 Introduction

The purpose of this project is to develop a series of algorithms for the digging operation of an autonomous bulldozer. The project is under the sponsorship of Topcon Corp. The intended audience are the managers, engineers, and future interns of the Topcon Corporation. The Sambuca Project relies primarily on computer vision for it's digging operation.

The following tasks were automated using computer vision:

- Identifing a stockpile
- Verification of stockpile material
- Aligning to the a stockpile
- Navigating toward a stockpile
- Calculating the optimal entry to a stockpile
- Verification of a bucket fill
- Traversal to a marked dumpsite

Using these developed features, we demonstrate the Housecat repeatedly digging a stockpile and moving its debris to a demarkated dumpsite. These developed features are also geared towards job efficiency. Using these developed features, a bulldozer can sense a stockpile's mass and will always attempt to penetrate the densest part of the stockpile when digging. The bulldozer will only traverse to a designated dumpsite when a full bucket of stockpile material has been achieved.

This report will first cover hardware used for this project. Then in Section 2 each task listed above will be covered. These tasks will be explained in the context of our final demonstration.

2 File System and ROS Overview

2.1 Packages

The project uses ROS package conventions. The packages are shown in Figure 1.

```
david@s27:~/Workspace/sambuca/ROS/catkin_ws/src$ ls
beginner_tutorials  gscam  nmea_navsat_driver  sambuca_pcl
CMakeLists.txt      movex  pcl
```

Figure 1: Sambuca ROS Packages

2.1.1 beginner_tutorials

Includes various ROS created tutorials. See http://wiki.ros.org/ROS/Tutorials[3] for more information.

2.1.2 gscam

GStreamer Camera driver package. Handles the incoming image frames. These frames are published to the topics boxed in Figure 2.

```
<group ns="camera">
 <env name="GSCAM_CONFIG" value="tcpclientsrc host=192.168.0.4 port=5000 ! gdpdepay
 <node name="center" pkg="gscam" type="gscam">
   <remap from="camera/image_raw" to="center/image_raw"/>
 </node>

 <env name="GSCAM_CONFIG" value="tcpclientsrc host=192.168.0.3 port=5003 ! jpegdec
 <node name="left" pkg="gscam" type="gscam">
   <remap from="camera/image_raw" to="left/image_raw"/>
 </node>

 <env name="GSCAM_CONFIG" value="tcpclientsrc host=192.168.0.4 port=5004 ! jpegdec
 <node name="right" pkg="gscam" type="gscam">
   <remap from="camera/image_raw" to="right/image_raw"/>
 </node>

</group>
```

Figure 2: ROS Camera Launch File

The **remap** tag indicates a re-mapping of topic name. See http://wiki.ros.org/roslaunch/XML/remap for more information.

2.1.3 nmea_navsat_driver

A "ROS driver to parse NMEA strings and publish standard ROS NavSat message types." **Not used** within this project.

2.1.4 sambuca_pcl

Optimal Approach calculation functions. The package contains two files shown in Figure 3.

```
david@s27:~/Workspace/sambuca/ROS/catkin_ws/src/sambuca_pcl/src$ ls
heap_analysis.cpp  heap.ply
```

Figure 3: sambuca_pcl Files

3 2D Environment Analysis

Our implementation of the 2D process can be found in the following file in the process_2D directory:

`ROS/catkin_ws/src/movex/py_src/state_impl/AUTO_DIG/process_2D/impl_states.py`

All corresponding files initializing the neural networks can be found in the weights directory:

`ROS/catkin_ws/src/movex/weights/`

The following sections will discuss methods of installation, training, testing, and implementation in the 2D process from the directory above. The impl_states.py file controls two separate states, "APPROACHING_2D_State" and "CONE_SEARCHING_2D_State". The two states are similar in the usage of identifying targets using the same network topology, except the approaching state uses an additional texture analyzing network.

3.1 Object Recognition Using Yolo

In order to implement the YOLO algorithm, we use a variant of Tensorflow named Darkflow which combines Tensorflow network implementation with the YOLO algorithm. We use Darkflow to implement transfer learning, by taking the pre-trained network, reinitializing the last two convolutional layers, and training them with our compiled dataset.

3.2 Darkflow

Instructions for setting up Darkflow and training can be found on the Github by thtrieu: `https://github.com/thtrieu/darkflow`[8]. The following sections describe some of the specific commands we used in more detail.

3.2.1 Training using Darkflow

Images can be pulled from websites such as Google images, and by using a program called LabelIMG we can annotate the data to generate a .xml file for each image containing the (x,y) points of a bounding box for a target. See the following link for details: `https://github.com/tzutalin/labelImg`[9]. Images and annotations are separated into respective folders. The file names in the images and annotations directory must match to avoid any error.

The initial model .cfg configuration file and .weights file used for transfer learning can be found on the YOLO website: `https://pjreddie.com/darknet/yolo/`[5]. We used tiny-yolo because of its smaller network size and faster compute time, but conversely also results in lower accuracy. With Darkflow installed and an annotated dataset compiled, we run the following command to begin training:

```
flow --model cfg/tiny-yolo-voc-1c.cfg --load bin/tiny-yolo-voc.weights
--train --annotation dataset_dir/annotations/ --dataset dataset_dir/images/
--epoch 100
```

The number of epochs correspond to the number of times a full batch of samples passes through the network once. Training time is correlated to dataset size.

We found through experimentation that training to around 850 epochs with a loss of around 0.04 was the most effective at creating a network that could perform well on any new test images without overfitting to the training set.

In order to load the most recently trained model, we use the following command which loads from the "ckpt/" directory:

```
flow --model cfg/tiny-yolo-voc-1.cfg --load -1 --savepb
```

This creates a "built_graph/" directory containing the model.pb and model.meta files we will use to load and run the network. To load any model from an earlier checkpoint, simply replace the number 1 in "–load -1" with the checkpoint number.

3.2.2 Testing Darkflow

In order to test the network, we use the following command on a directory of sample images:

```
flow --imgdir sample_img_dir --metaLoad graph.meta --pbLoad graph.pb
--threshold 0.2 --json
```

The threshold is set at 0.2, which will allow the network to display anything above a confidence of 20%. The "–json" command is optional, and is used to print the confidences to a json file rather than output the resulting images. Examples of testing the network on novel images can be seen in Figures 4 and 5.

Figure 4: Prediction on Sand and Gravel

Figure 5: Prediction on Topcon Site

3.2.3 Running Darkflow in the 2D Process

Within the "runState", we initialize the network in the code block in Figure 6. The "root" variable is set inside the local computer's .bashrc to be the path to

```
if self.firstRun:
    print "Initializing Mulchpile Searching"
    root = os.environ['SAMBUCAROOT']
    self.options = {
        # ## MULCH ##
        'pbLoad': root + "/ROS/catkin_ws/src/movex/weights/mulchpile_5_gray_875/tiny-yolo-voc-1.pb",
        'metaLoad': root + "/ROS/catkin_ws/src/movex/weights/mulchpile_5_gray_875/tiny-yolo-voc-1.meta",

        'threshold': 0.20,
        'gpu': 0.7
    }
    self.tfnet = TFNet(self.options)
```

Figure 6: Initializing the Darkflow Network

the ROS directory.

The getCenterFrame() function is called to return a frame from the center camera topic. We then call the following function which takes the input frame, passes it through the network, and stores a list of predictions in the results variable:

```
results = self.tfnet.return_predict(frame)
```

We then cycle through the list of results and find the max confidence in code block in Figure 7. After the loop we we will have the maximum confidence

```
# cycle through each prediction and print predictions onto frame
for result in results:
    tl = (result['topleft']['x'], result['topleft']['y'])
    br = (result['bottomright']['x'], result['bottomright']['y'])
    label = result['label'] + ' : ' + str(result['confidence'])
    # draw prediction's bounding box on frame with correct label
    frame = cv2.rectangle(frame, tl, br, self.color, 7)
    frame = cv2.putText(frame, label, tl, cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 0), 2)

    # check for prediction with highest confidance
    # highest confidance prediction becomes target that we want to drive towards
    if result['confidence'] > max_confidence:
        target = result
        # target_center = self.getCenterOfTarget(target)
        max_confidence = target['confidence']
```

Figure 7: Looping through Network Results

bounding box which is above minimum threshold, which we will then use to do texture analysis.

3.3 Texture Analysis using Tensorflow

For texture analysis we based our network on a binary image classifier, which can be referenced here: `http://cv-tricks.com/tensorflow-tutorial/training-\convolutional-neural-network-for-image-classification/`[7]. The original code can be found on the following Github: `https://github.com/sankit1/cv-tricks.com/tree/master/Tensorflow-tutorials/tutorial-2-image-classifier`.

3.3.1 Training the Texture Network

To train the network, we compile hundreds of images for both a positive and negative set. Images are taken directly from the target environment from the bot's center camera, and segmented into N x N images for training

To generate training data, use the attached script "targetedMince.py". This script will open an image, then cut the area you select up into a grid of small tiles of a width and height specified on the command line. Select areas of the texture you would like to train on and press '1' and the script will save the images to a folder named after the texture.

Command line usage: python targetedMince.py file texture width height

3.3.2 Running the Texture Network in the 2D Process

When we want to verify that our pile is not a false positive, or that our bucket is full, we use a neural network in order to perform texture analysis. We cut up the image into an N x N array of smaller images each of which get fed into the texture analyzer, and from there we process the array of predictions.

The function that "cuts up" the image into an array of images is called getGrid() and can be seen here:

```
def getGrid(self, target, colorFrame, n, image_size, num_channels):
    # create texture img
    # revise for global variables x, y
    x = (target['topleft']['x'], target['bottomright']['x'])
    y = (target['topleft']['y'], target['bottomright']['y'])
    image = colorFrame[y[0]:y[1], x[0]:x[1]]
    images = []

    #image = cv2.resize(texture_img, (image_size, image_size),0,0, cv2.INTER_LINEAR)
    box_height = int(image.shape[0]/n)
    box_width  = int(image.shape[1]/n)

    # split yolo box and append to images array
    for i in range(n):
        for j in range(n):
            box = image[i*box_height : (i+1)*box_height, j*box_width : (j+1)*box_width]
            box = cv2.resize(box, (image_size, image_size),0,0, cv2.INTER_LINEAR)
            images.append(box)
    return images
```

Figure 8: getGrid()

4 Stereo Camera Calibration

To calibrate the stereo set of cameras, the following resources were used: `http://wiki.ros.org/camera_calibration/Tutorials/StereoCalibration`[4] and `http://wiki.ros.org/stereo_image_proc/Tutorials/ChoosingGoodStereoParameters`[6].

Calibrating the stereo cameras is done in the following steps:

1. Run the stereo camera calibration tool:

   ```
   rosrun camera_calibration cameracalibrator.py --size 7x5 --square 0.675
   right:=/camera/right/image_raw left:=/camera/left/image_raw
   right_camera:=/camera/right left_camera:=/camera/left --approximate=0.1
   ```

   "–size" and "–square" refer to the measured checker sizes for the board used to calibrate.

2. Run the camera_test launch file:

   ```
   roslaunch camera_test
   ```

3. Run stereo_image_proc:

   ```
   ROS_NAMESPACE=camera rosrun stereo_image_proc stereo_image_proc _approximate_sync:=true
   ```

4. Run rqt_reconfigure to tune the stereo algorithm parameters live:

   ```
   rosrun rqt_reconfigure rqt_reconfigure
   ```

   Chosen parameters can be saved and loaded from a saved path in a given launch file.

5. Run stereo_view to display resulting disparity map:

   ```
   rosrun image_view stereo_view stereo:=camera image:=image_rect _approximate_sync:=true
   ```

5 Stereo Image Pipeline (ROS Nodelets)

ROS Nodelets were used to create the stereo image pipeline shown in Figure 9. Green blocks indicate nodelets. Nodelets provide a system with faster communication between nodes. Nodelets are attached to a **NodeletManager**. The NodeletManager wraps all child nodelets into one process. Communication is performed via shared memory. This is faster than conventional ROS nodes. Nodelets can be viewed at: **http://wiki.ros.org/nodelet**.

Conventional ROS nodes communicate via local network. I.e each node has it's own IP address and performs interprocess communication over the local network.

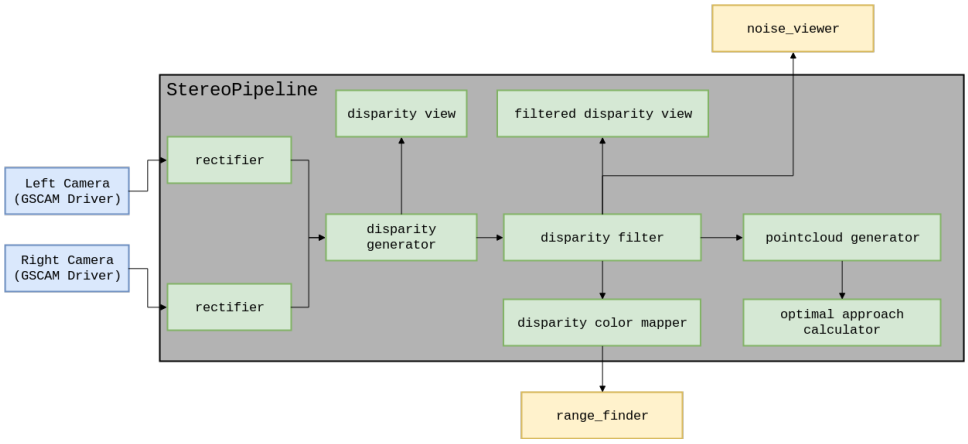


Figure 9: Stereo Pipeline using ROS Nodelets

The following nodelets are part of the ROS library:

- rectification
- disparity generation
- point cloud generation

The following nodelets were developed by our team:

- disparity filtering
- disparity color mapper
- optimal approach calculation

5.1 Stereo Image Process

ROS provides a **Stereo Image Process**: **http://wiki.ros.org/stereo_image_proc**. This process is a black box where raw images go in and a disparity image comes

out. We needed to add filtering and optimal approach calculation to this process so we took the individual pieces of the **Stereo Image Process** and created our own image pipeline.

5.2 Disparity Filtering

The disparity map created by the rectified images was low quality and needed to be smoothed before point cloud generation. Disparity filtering is performed in **../ROS/catkin_ws/src/movex/src/disp_filter_nodelet.cpp**.

5.3 Disparity Color Mapper

ROS does not have a public way to convert a grayscale disparity image into a colored disparity map. Normally, ROS performs the grayscale to color mapping in the **Stereo Image Process**. But since we made our own image pipeline we also had to create our own color mapper.

5.4 Optimal Approach Calculator

This nodelet is detailed in Section 6.2.3

6 3D Environment Analysis

The process_3D contains the optimal approach logic.

`ROS/catkin_ws/src/movex/py_src/state_impl/AUTO_DIG/process_3D/impl_states.py`

Within the *impl_states.py* file, the angle of maximum area is calculated from the point cloud. This was covered in Section 2.1.4

`ROS/catkin_ws/src/movex/src/optimal_approach_nodelet.cpp`

6.1 Installation

The 3D process requires the installation of the Point Cloud Library.[2] Ubuntu users can add the PointCloudLibrary repository and install via apt-get:

```
sudo add-apt-repository ppa:v-launchpad-jochen-sprickerhof-de/pcl
sudo apt-get update
sudo apt-get install libpcl-all
```

Figure 10: Ubuntu PointCloudLibrary Installation

For other systems see: **http://www.pointclouds.org/downloads/linux.html**[1].

6.2 Calculating the Optimal Approach

heap_analysis.cpp holds all the functions used for the optimal approach calculation. The functions are linked into a shared library for use by other packages.

Additionally, *heap_analysis.cpp* is compiled into an executable to demonstrate the optimal approach calculation on a perfect data set. The main function of *heap_analysis.cpp* consumes the heap.ply point cloud file. This data set was created using a 3D model and "sprinkling" points over the surface. The 3D model creation was done in the browser application Vectary. The output format was a mesh object.

See https://www.vectary.com/ for more information.

The point cloud "sprinkling" was done in the application **Meshlab** with the **Poisson Disk Sampling Filter**. The sampling of the mesh object created the *heap.ply*.

See http://www.meshlab.net/ for more information.

heap_analysis.cpp is compiled into an executable called **consume_ply**. It performs some transformations on the point cloud and runs the optimal approach algorithm. The visual output is shown in Figure 11. The intersection points are marked by gray spheres. The reference plane is seen in the background. Note that the intersection points are spread out. This is because the optimal approach algorithm in tuned for stereo camera data.

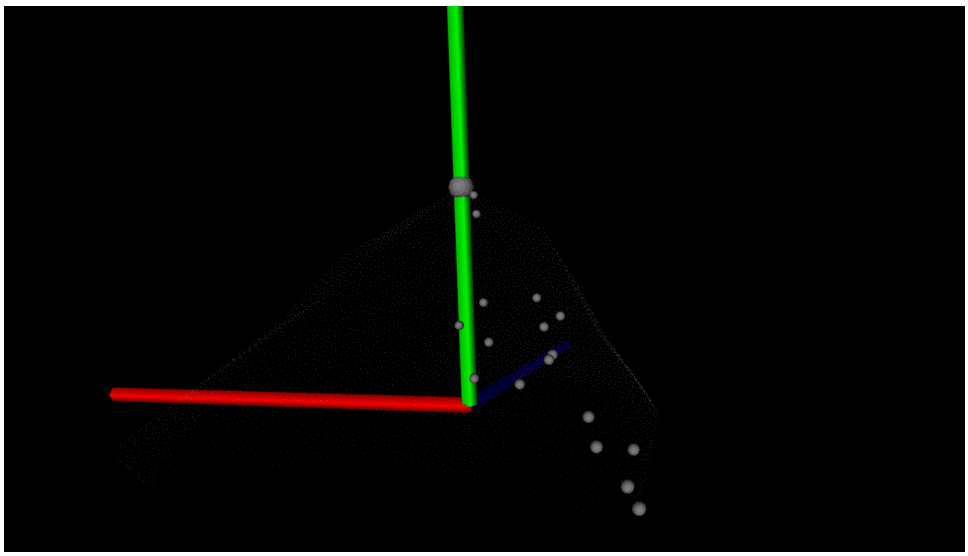


Figure 11: consume_ply Output

6.2.1 heap_analysis.cpp

heap_analysis.cpp contains all the functions used for optimal approach. They are listed in the header file: **/ROS/catkin_ws/src/sambuca_pcl/include/sambuca_pcl/heap_analysis.h**

Their functions are presented in Figures 12 and 13.

```
36 /* Find the maximum Y value where X = 0 and Z > 0*/
37 void computeMaxPointAhead(pcl::PointCloud<pcl::PointXYZ>& pc,  pcl::PointXYZ& maxPoint);
38
39 /* Given a pointcloud compute the minimum X,Y,Z values.*/
40 void computeMinPoints(pcl::PointCloud<pcl::PointXYZ>& pc, float& minX,
41                                                           float& minY,
42    ▮                                                      float& minZ);
43
44
45
46 /* Given a pointcloud compute the minimum X,Y,Z values and the index within
47  * the pointcloud of those values. */
48 void computeMaxPoints(pcl::PointCloud<pcl::PointXYZ>& pc, float& maxX,
49                                                           float& maxY,
50                                                           float& maxZ,
51                                                           uint16_t& idxX,
52                                                           uint16_t& idxY,
53                                                           uint16_t& idxZ);
54
55 /* Takes original value o and normalizes it. */
56 float calculateNormal(float o, float min, float max);
57
58 /* Normalizes a pointcloud in-place. Requires an array of minimum and
59  * maximum [X,Y,Z] values. All normalized values are multiplied by scale. */
60 void normalizePointCloud(pcl::PointCloud<pcl::PointXYZ>& pc, float min[3],
61                                                              float max[3],
62                                                              uint16_t scale);
63
64 /*  Creates a reference plane.
65  *
66  *  E.g: width = height = 10, xScale = yScale = 10, forms square of 10x10 points
67  *  with max length position X = 10 and max height position Y = 10.
68  *
69  *  E.g: width = 20, height = 10, xScale = 1, yScale = 2, forms a rectangle
70  *  of 20X10 points with a max length position of X = 1 and max height
71  *  position Y = 2.
72  *
73  * */
74 void createReferencePlane(pcl::PointCloud<pcl::PointXYZ> *refPlane,
75                           uint32_t width, uint32_t height, uint32_t xScale, uint32_t yScale);
76
```

Figure 12: heap_analysis.h Part 1

```
80
81 /*
82  * Get the intersection between the heapCloud and the refPlane.
83  * http://docs.pointclouds.org/trunk/kdtree_2include_2pcl_2kdtree_2impl_2io_8hpp_source.html#l00068
84  *
85  * Nearest neighbor thresholds are set within the function.
86  * Vectors of indices and PointXYZ's indicate heapCloud points on the intersection.
87  * */
88 void getIntersection(pcl::PointCloud<pcl::PointXYZ>::Ptr heapCloudPtr,
89                      pcl::PointCloud<pcl::PointXYZ>::Ptr refPlanePtr,
90                      std::vector<int> &indices,
91                      std::vector<pcl::PointXYZ> &intersectionPoints);
92
93 /* Calculates the area beneath the points. */
94 float areaBeneathIntersection(std::vector<pcl::PointXYZ> &intersectionPoints,
95                               pcl::PointXYZ& peakPoint);
96
97
98
99 /*
100  * Scans from 0 - PI and returns the angle of maximum area
101  * useViewer should be set to 0 during continuous operation.
102  * usePly should be set to 1when running the PLY demo
103  * */
104 float scanForEntry(pcl::PointCloud<pcl::PointXYZ>::Ptr heapCloudPtr,
105                    pcl::PointCloud<pcl::PointXYZ>::Ptr refPlanePtr,
106                    pcl::PointXYZ& peakPoint,
107                    boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer,
108                    int useViewer,
109                    int usePly);
```

Figure 13: heap_analysis.h Part 2

6.2.2 Linking Multi-Package C++ in ROS

Using the **heap_analysis.hpp** functions requires linking the file into a shared library. Figure 14 shows the creation of the **consume_ply** executable and the creation of the shared library. The full file is located in: **ROS/catkin_ws/src/sambuca_pcl/CMakeLists.txt**.

```
51 add_executable(consume_ply src/heap_analysis.cpp) #src/pcd_write.cpp )
52 target_link_libraries(consume_ply ${catkin_LIBRARIES})
53
54
55 ## Declare a C++ library
56 add_library(${PROJECT_NAME}
57         src/heap_analysis.cpp
58 )
59 ## Specify libraries to link a library or executable target against
60 target_link_libraries(${PROJECT_NAME} ${catkin_LIBRARIES})
61
62 ##############
63 ## Install ##
64 ##############
65
66 # all install targets should use catkin DESTINATION variables
67 # See http://ros.org/doc/api/catkin/html/adv_user_guide/variables.html
68
69 ## Mark executable scripts (Python etc.) for installation
70 ## in contrast to setup.py, you can choose the destination
71 # install(PROGRAMS
72 #   scripts/my_python_script
73 #   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
74 # )
75
76 ## Mark executables and/or libraries for installation
77 install(TARGETS ${PROJECT_NAME}
78 ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
79 LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
80 RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
81 )
82
83 ## Mark cpp header files for installation
84 install(DIRECTORY include/${PROJECT_NAME}
85 DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
86 FILES_MATCHING PATTERN "*.h"
```

Figure 14: sambuca_pcl CMakeLists.txt Excerpt

After the **heap_analysis** library is created, it must be referenced in the **movex** package. This is shown in Figures 15 and 16. Line 10 shows the **sambuca_pcl** package included into the **movex** build. Line 112 shows the **sambuca_pcl** library linked to the optimal approach nodelet.

This file is located: **ROS/catkin_ws/src/movex/CMakeLists.txt**

```
 1 cmake_minimum_required(VERSION 2.8.3)
 2 project(movex)
 3
 4 find_package(catkin REQUIRED COMPONENTS rospy roscpp std_msgs genmsg
 5         actionlib_msgs actionlib message_generation cv_bridge
 6         nodelet pluginlib pcl_conversions pcl_ros)
 7
 8 find_package(SDL2 REQUIRED)
 9 find_package(OpenCV)
10 find_package(sambuca_pcl REQUIRED)
```

Figure 15: Including the sambuca_pcl Package

```
109 add_library(optimal_approach_nodelet src/optimal_approach_nodelet.cpp)
110 add_dependencies(optimal_approach_nodelet ${catkin_EXPORTED_TARGETS})
111 add_dependencies(optimal_approach_nodelet movex_generate_messages_cpp)
112 target_link_libraries(optimal_approach_nodelet ${catkin_LIBRARIES}
113         ${sambuca_pcl_LIBRARIES})
114
```

Figure 16: Including the sambuca_pcl Package

6.2.3 optimal_approach_nodelet.cpp

The Optimal Approach Nodelet located in: **/ROS/catkin_ws/src/movex/src/optimal_approach_nodelet**

The Optimal Approach Nodelet waits for a processing request. Once received the Optimal Approach Nodelet consumes one point cloud and performs the calculation. The processing request is handled by a service thread and the calculation is handled by another thread.

Figure 17 shows the class definition. Node the inheritance of **nodelet**. **onInit** specifies the nodelet's operation when loaded into the nodelet manager. Lines 72-74 define several process variables. These provide synchronization between the service thread and the calculation thread. Line 87 subscribes the nodelet to the pointcloud. Line 91 creates an approach service where clients may request an optimal approach.

```
66 class OptimalApproachNodelet : public nodelet::Nodelet{
67
68
69
70
71         // Process variables
72         uint8_t processPC = 0; // Whether or not to process point cloud
73         uint8_t dataReady = 0; // Whether or not the result is ready
74         float   optimalAngle = 0;
75
76         ros::Subscriber    pointCloudSub;
77         ros::ServiceServer approachService;
78
79         public:
80             OptimalApproachNodelet(){};
81             ~OptimalApproachNodelet(){};
82
83             virtual void onInit(){
84                     NODELET_INFO_STREAM("OPTIMAL APPROACH onInit\n");
85                     ros::NodeHandle &nh = getMTPrivateNodeHandle();
86
87                     pointCloudSub = nh.subscribe("/sambuca/points2", 1,
88         ▮                                &OptimalApproachNodelet::pointcloudCallback, this);
89
90
91                     approachService = nh.advertiseService("/sambuca/approachService",
92                                     &OptimalApproachNodelet::handleApproachService, this);
93
94             }
95
```

Figure 17: Optimal Approach Class Definition

Figure 18 shows the service handling the approach requests. When a request is received, the request is blocked until data is ready. (See line 83).

When the calculation is done the calculation thread sets **dataReady** and **optimalAngle**. **optimalAngle** is then handed to the response object.

```
96              bool handleApproachService(movex::Appch::Request  &req,
97                                         movex::Appch::Response &res){
98
99
00                      NODELET_INFO_STREAM("OPTIMAL APPROACH waiting for pc\n");
01                      processPC = 1; // Let pointcloudCallback know it can start processing
02                      dataReady = 0; // Let pointcloudCallback toggle when it is finished
03                      while(!dataReady){
04                          //pass
05                      }
06                      processPC = 0; // Tell pointcloudCallback to stop processing
07
08
09                      res.distance = 0;
10                      res.angle    = optimalAngle;
11                      return true;
12
13              }
```

Figure 18: Optimal Approach Service

Figure 19 shows the start of the optimal approach calculation. After passing several synchronization checks, it is ready for processing.

```
116             void pointcloudCallback(const sensor_msgs::PointCloud2ConstPtr& input){
117                 static int isRunning = 0;
118                 NODELET_INFO_STREAM("OPTIMAL APPROACH attempting\n");
119                 // Only process if we get a go-ahead from the service
120                 if(!processPC){
121                     NODELET_INFO_STREAM("OPTIMAL APPROACH process check\n");
122                     return;
123                 }
124
125
126                 // Don't process if we've already begun
127                 if(isRunning){
128                     return;
129                 }
130                 isRunning = 1;
131
132                 NODELET_INFO_STREAM("OPTIMAL APPROACH received pointcloud\n");
```

Figure 19: Optimal Approach Calculation(1)

Figure 20 shows the next step: Converting the ROS point cloud to a PCL point cloud and making the cloud workable. On line 142 we create a PCL point cloud with XYZ and RGB values. We are only interested in the XYZ values to we copy the XYZRGB point cloud into a XYZ only cloud on line 158.

Next we remove all the NaNs. TThe calculation will fail if NaNs are not removed.

Finally, a smart pointer is created to the cloud.

```
141                 // Aquire the cloud
142                 pcl::PointCloud<pcl::PointXYZRGB> inputCloud;
143                 pcl::fromROSMsg(*input, inputCloud);
144
145                 // Remove the rgb values
146                 // We are working with xyzCloud from now on
147                 pcl::PointCloud<pcl::PointXYZ>      xyzCloud;
148
149                 //  pcl::PointCloud<pcl::PointXYZ>::Ptr xyzCloudPtr(&xyzCloud);
150                 pcl::copyPointCloud(inputCloud, xyzCloud);
151
152                 // Remove all Nans
153                 std::vector<int> indices;
154                 pcl::removeNaNFromPointCloud(xyzCloud, xyzCloud, indices);
155
156                 pcl::PointCloud<pcl::PointXYZ>::Ptr xyzCloudPtr(new pcl::PointCloud<pcl::PointXYZ>());
157                 pcl::copyPointCloud(xyzCloud, *xyzCloudPtr);
158                 cout << "Copy Size: " << xyzCloudPtr->size();
159
```

Figure 20: Optimal Approach Calculation(2)

Figure 21 shows the reference plane being created on line 234. The reference plane is set to the origin, I.e the peak of the pile.

Finally, **optimalAngle** is calculated in scanForEntry on line 267.

```
227                 // Create reference plane
228                 uint32_t width = 500;
229                 uint32_t height = 75;
230                 pcl::PointXYZ def(0,0,0); // Default value
231                 pcl::PointCloud<pcl::PointXYZ> refPlane(width, height, def);
232                 //pcl::PointCloud<pcl::PointXYZ>::Ptr refPlanePtr(&refPlane);
233                 cout << "Size: " << refPlane.size();
234                 createReferencePlane(&refPlane, width, height, 75, peakPoint.y+1);
235                 pcl::PointCloud<pcl::PointXYZ>::Ptr refPlanePtr(new pcl::PointCloud<pcl::PointXYZ>());
236                 pcl::copyPointCloud(refPlane, *refPlanePtr);
237
238
239
240                 // Setup reference plane at origin
241                 float new_x, new_z = 0;
242                 float new_y = -1;
243                 //new_x = peakPoint.x;
244                 //new_y = peakPoint.y;
245                 //new_z = peakPoint.z;
246
247                 Eigen::Affine3f transform2 = Eigen::Affine3f::Identity();
248                 transform2.translation() << -new_x, new_y, -new_z;
249                 //pcl::transformPointCloud(refPlane, refPlane, transform2);
250
251
252                  // Create visualizer
253
254
255                 boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer;
256             //  viewer = simpleVis(xyzCloudPtr);
257                 std::cout << "Visualizer Created" << std::endl;
258
259
260
261
262                 if(xyzCloudPtr->size() == 0){
263                     NODELET_INFO_STREAM("OPTIMAL APPROACH empty point cloud\n");
264                     return;
265                 }
266                 // Scan for optimal entry
267                 optimalAngle = scanForEntry(xyzCloudPtr,
268                           refPlanePtr,
269                             peakPoint,
270                             viewer, 0, 0);
```

Figure 21: Optimal Approach Calculation(3)

6.2.4 Translating the Dozer

With a given angle of optimal approach, we use a trigonometric method of translating the bot based on an isosceles triangle which can be seen in Figure 22. In this example, we take the angle β as the magnitude of the angle from the bot's orientation. We then calculate the angle α along with an estimate of the bots rotational and translational velocity to maneuver the bot toward the optimal vector.

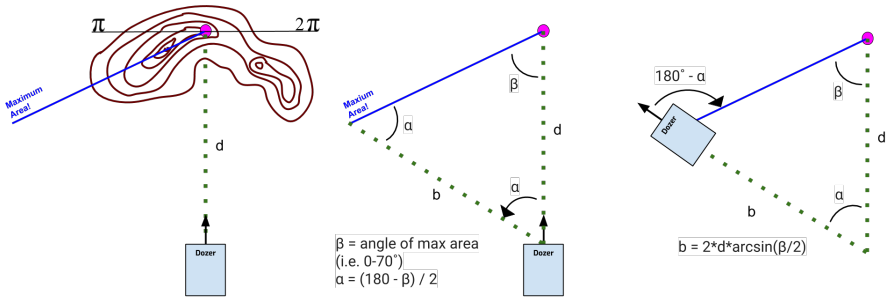


Figure 22: Translating the Dozer to the Optimal Vector

7 Closed-Loop Bucket Control

Bucket control was achieved using the IMU's on the Houscat. The **ActionServer** contains logic to operate the bucket control either as timed movements or as setpoint movements. If a magnitude is set on a **ActionGoal** the bucket will move to a set point. But if a duration is included on a **ActionGoal** the bucket will move for a duration. Figure 23 shows the difference.

```
22      # Action 1: Bucket Setpoint
23      goal1 = movementGoal()
24      goal1.requester = "Action Tester"
25      goal1.liftRequest.isActive = True
26      goal1.liftRequest.magnitude = 45
27
28      # Action 2: Bucket Setpoint
29      goal2 = movementGoal()
30      goal2.requester = "Action Tester"
31      goal2.grabRequest.isActive = True
32      goal2.grabRequest.magnitude = 1
33      goal2.grabRequest.duration = 3
```

Figure 23: Setpoint and Duration ActionGoals

Action 1 has a magnitude only and will move the bucket arm to postive 45 degrees. Action 2 will raise the bucket 3 seconds because the duration is three and the magnitude is positive.

The closed loop control is handled within **action_server.py**. When a bucket command is received, a **PoseObserver** thread is created. This thread runs asynchronously from the main thread and consumes IMU data. Once the set point is reached the **PoseObserver** notifies the **ActionServer** to send a stop command.

```
223                 # Create and start a pose observer thread
224                 poseObserver = PoseObserver(self, cmdName, SambucaPoseTopics[cmdName],\
225                                             request.magnitude)
226                 poseObserver.start()
227
```

Figure 24: Creating a PoseObserver

Figure 24 shows the **PoseObserver** thread being created.

```
70      def handlePose(self, pose):
71          angle = pose.data
72          rospy.loginfo("{} {}".format(self.poseTopic, angle))
73
74
75          # Run the motors toward the reference angle
76          if(self.cmdName == SambucaCmdNames.GRAB):
77              if(self.refAngle - angle > 0):
78                  SambucaCmdFunc[self.cmdName](1)
79              else:
80                  SambucaCmdFunc[self.cmdName](-1)
81
82          if(self.cmdName == SambucaCmdNames.LIFT):
83              if(self.refAngle - angle > 0):
84                  SambucaCmdFunc[self.cmdName](1)
85              else:
86                  SambucaCmdFunc[self.cmdName](-1)
87
88          # If the reference angle and the current angle are close
89          # enough, stop the arm movement
90          if(abs(angle - self.refAngle) < self.threshold):
91
92              self.sub.unregister()
93              self.actionServer.notifyStop(self.cmdName)
94              self.runFlag.clear()
95              rospy.loginfo("Reference angle {} reached".format(self.refAngle))
96              # time.sleep(1)
```

Figure 25: PoseHandler Callback Method

Figure 25 shows the **PoseHandler** pose callback method. This method consumes IMU data and adjusts the arm or bucket while outside the setpoint. Once within a threshold, the **PoseObserver** clears it's own **runFlag** and notifies the **ActionServer** to stop the movement.

The action server is located in: **ROS/catkin_ws/src/movex/nodes/services/action_server.py**.

8 Hardware

For this project we were given a scaled replica of a full-size bulldozer, which allows a close approximation of identical behavior in a lab setting. The small scale replica is named the Housecat MkII, while the full-size is named the Movex M-48.

Both the Housecat and the Movex consist of a stereo pair of cameras, as well as a singular camera mounted top-center, both of which can be seen in Figures 26 and 27. On board are two Raspberry Pi's which are used to relay the camera video frames to a host computer with the use of an on-board Wifi router. Movement and bucket control is done through the use of an Arduino which relays control signals to the drive motors and pair linear actuators for lift and tilt operations. The system is equipped with IMU's (inertial measurement unit) which allow closed loop feedback operation of the bucket position.

Development was done mostly on the Housecat. The software was proven scalable by successfuly running it on the Movex M48.

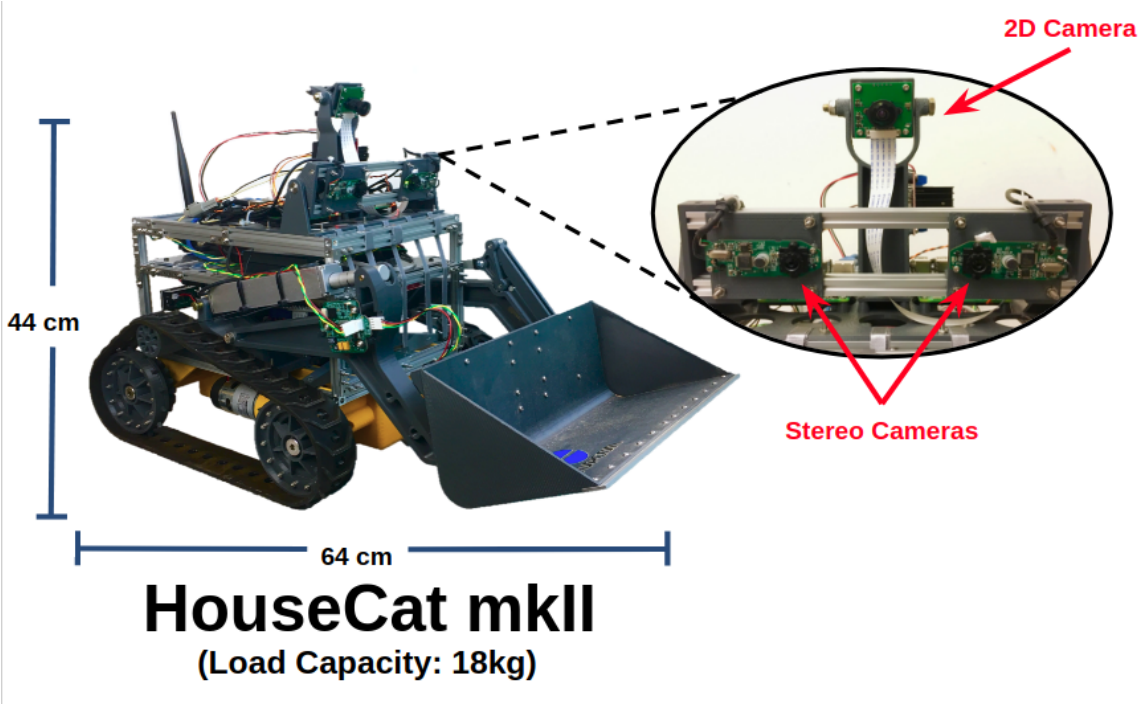


Figure 26: The HouseCat mkII

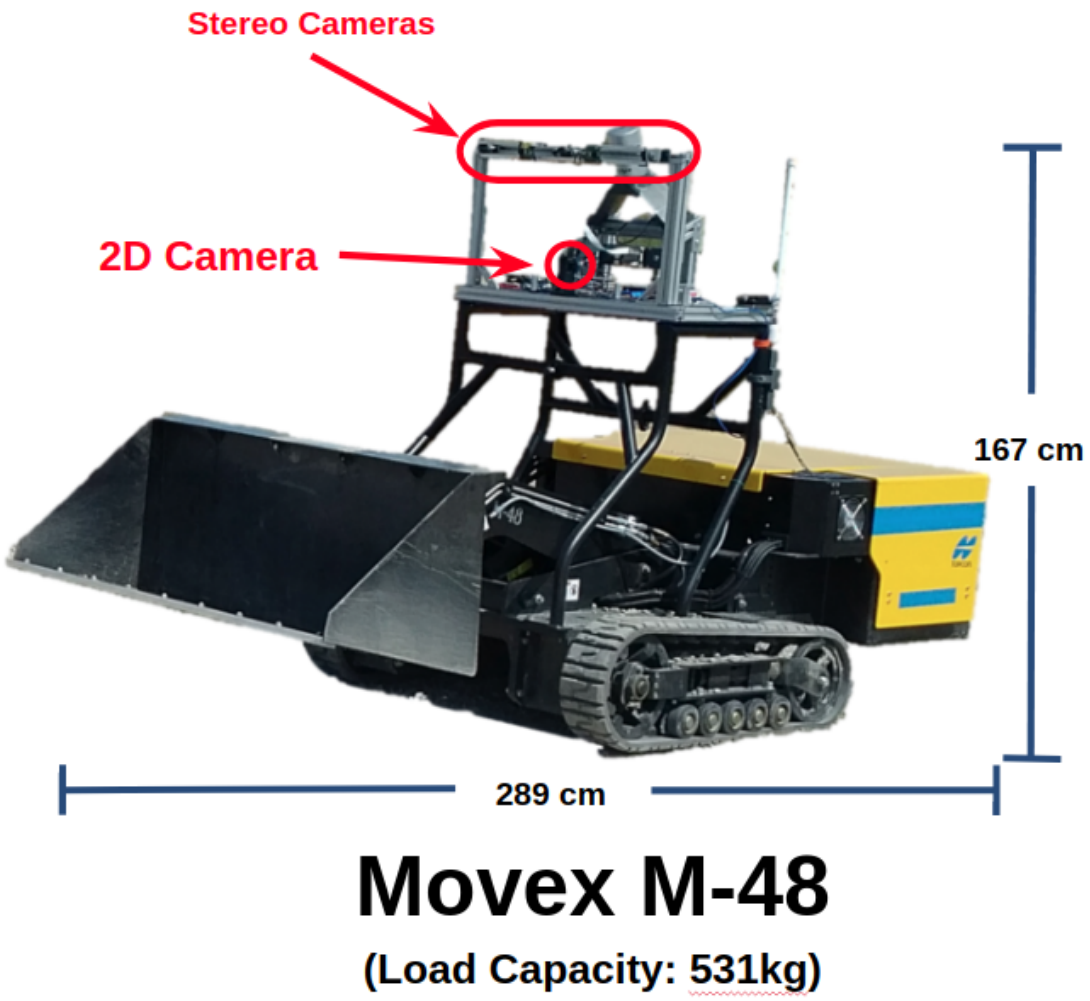


Figure 27: The Movex M48

References

[1] Installing pcl, 2018. http://www.pointclouds.org/downloads/linux.html.

[2] Pcl webpage, 2018. http://pointclouds.org/.

[3] Ros tutorials, 2018. http://wiki.ros.org/ROS/Tutorials.

[4] Z. Klapow I. Saito. How to calibrate a stereo camera. `http://wiki.ros.org/camera_calibration/Tutorials/StereoCalibration`, 2017.

[5] J. Redmon. You only look once: Unified, real-time object detection, 2016. https://arxiv.org/abs/1506.02640.

[6] Open Robotics. Choosing good stereo parameters. `http://wiki.ros.org/stereo_image_proc/Tutorials/ChoosingGoodStereoParameters`, 2018.

[7] A. Sachan. Tensorflow tutorial 2: image classifier using convolutional neural network, 2018. http://cv-tricks.com/tensorflow-tutorial/training-convolutional-neural-network-for-image-classification/.

[8] thtrieu. Darkflow, 2018. https://github.com/thtrieu/darkflow.

[9] tzutalin. Labelimg. `https://github.com/tzutalin/labelImg`, 2018.