**Autonomous Ultrasound Probe Placement and Photoacoustic Needle Tracking for Spinal Surgeries**

Produced by: Joshua Shubert

Mentor: Dr. Muyinatu Bell

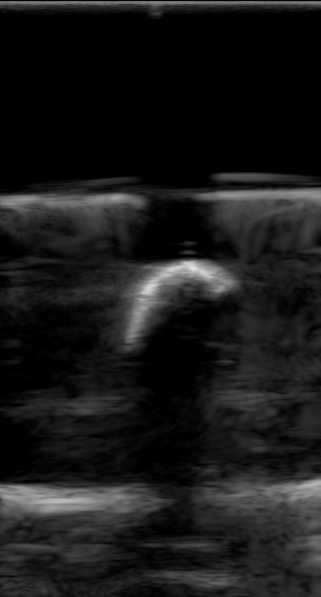
1. Abstract

A system was designed and implemented to address the need to visualize surgical tools during spinal operations without the use of repeated X Ray images. The system consists of two main components, hereafter referred to as Phase 1 and Phase 2. Phase 1 consists of autonomous preoperative placement of the ultrasound probe onto the patient’s spine. Phase 2 consists of autonomous intraoperative tracking and visualization of the needle as it is inserted into the spine. Together these phases form a complete surgical system that has the realistic potential to eliminate or greatly reduce the need for ionizing radiation based imaging during the procedure. In practice Phase 1 was able to successfully place an Ultrasound probe onto a patient’s spine however future work is needed for successful placement on specific vertebrae. Phase 2 was able to successfully visualize the location of the needle tip on a preoperative CT image of the vertebra. The visualized trajectory on the CT image closely matched the trajectory of the needle insertion showing promise for using photoacoustic tool tracking as an alternative to repeated C-arm X Ray or fluoroscopy guidance.

1. Background

Spinal surgeries are some of the most common operations performed. In particular, 150,000 Spinal Fusions are performed each year [1] and 90 out of every 100,000 Medicare enrollees have had a Vertebroplasty [2] with the number increasing every year. Currently these and many other spinal procedures are primarily guided by fluoroscopy or repeated C-arm X-rays

This creates a large radiation dose for the patient and perhaps more importantly the surgeon [3][4] Additionally, X-Ray modalities cannot visualize soft tissues such as nerves and blood vessels without the use of contrast agents. It is reported that 3% of all spinal fusion surgeries (or about 5,000 each year) result in nerve damage. Additionally spinal fusion surgery sometimes results in blindness when the combination of the placement of the fusion mass and the patient position during surgery cause an ischemic injury [6].

For Vertebroplasty as well as the similar procedure, Kyphoplasty, it is reported that the radiation exposure from intraoperative fluoroscopy can cause an additional 410 cases of cancer per 1 million patients [7] with the risk increased for younger patients or patients who may have children in the future as the radiation is predicted to increase the risk of birth defects.  
So there is a clear need for intraoperative imaging that is simultaneously able to visualize critical structures such as nerves and blood vessels while also not adding to the patient and surgeons cumulative radiation dose. Ultrasound imaging has the benefit of being low cost and radiation-free, however the high acoustic reflection caused by the bone interface makes it effectively impossible to see the inside of vertebrae. Additionally, 2D ultrasound images can be confusing. For example the following image is of the transverse process of a vertebra which is not at all obvious from the single picture, but rather requires developing structural context by scanning nearby locations with the probe. Additionally, ultrasound has no easy way to discriminate critical structures from the rest of the image. The requirements of visualizing critical structures, while also not creating ionizing radiation, point to using photoacoustic imaging instead of traditional ultrasound.

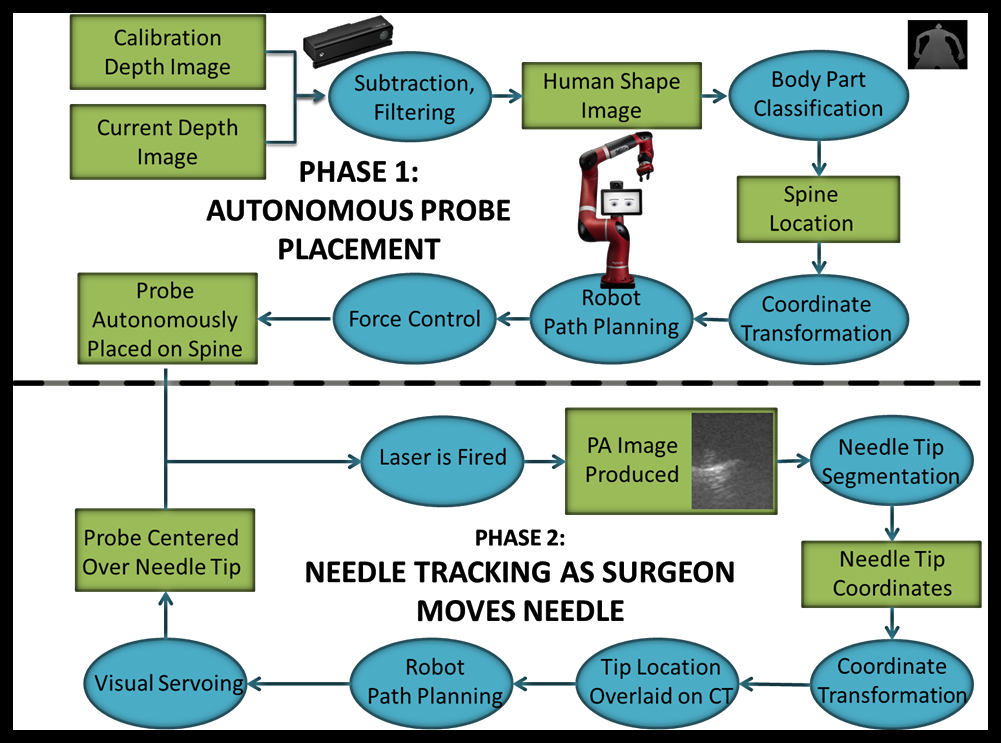
Photoacoustic Imaging is a new imaging modality that is adept at visualizing these critical structures as well as surgical tools while also having no harmful side effects (so long as the laser energy is limited) so there is great potential for it to aid in surgical tool tracking and visualization during spinal surgeries.

1. Introduction

To address the issues highlighted in the preceding section, I developed two systems. The goal of the first system, Phase 1, is to be able to autonomously and safely place an ultrasound probe onto a patient’s spine. The goal of the second system, Phase 2, is to track and follow a needle as it is inserted into a patient’s vertebra, and visualize the position of the needle tip to the surgeon in a useful way such that radiation based imaging modalities do not need to be used.   
In general the first system consists of a Kinectv2 depth sensor, a Sawyer Robot with integrated force sensing, a Linear Array Ultrasound Probe, and custom control software for the Kinectv2 and Sawyer Robot.   
The second system consists of a Sawyer Robot, Linear Array Ultrasound Probe, Alpinion Ultrasound Scanner, Photoacoustic Laser source, thoracic vertebra sample, and a hollow needle with an optical fiber temporarily fixed inside, as well as Needle Tip Segmentation and custom Robot Control software.

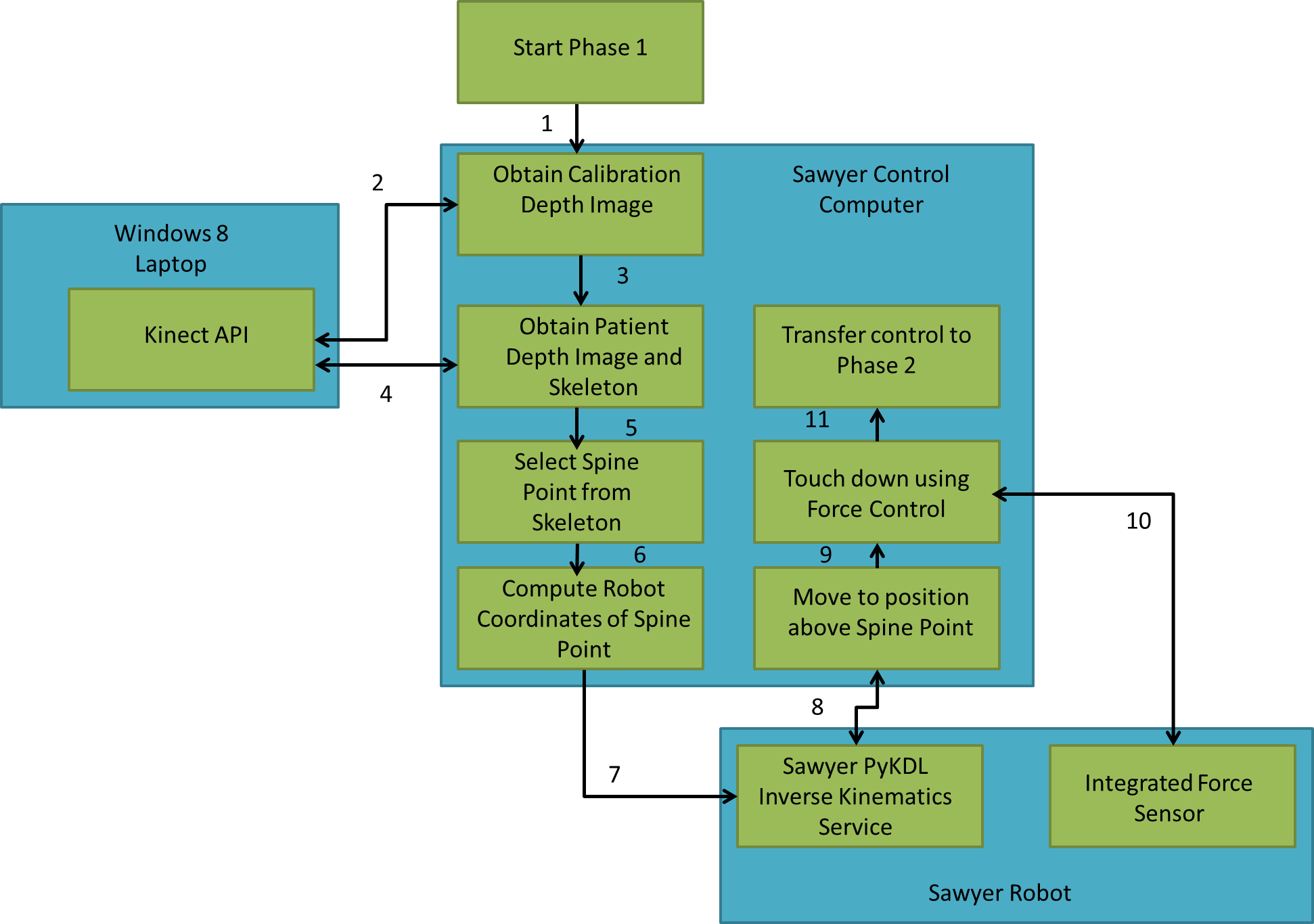
The architecture of the overall system, as well as the architecture of selected software components is elaborated on in the following section.

1. System Architecture



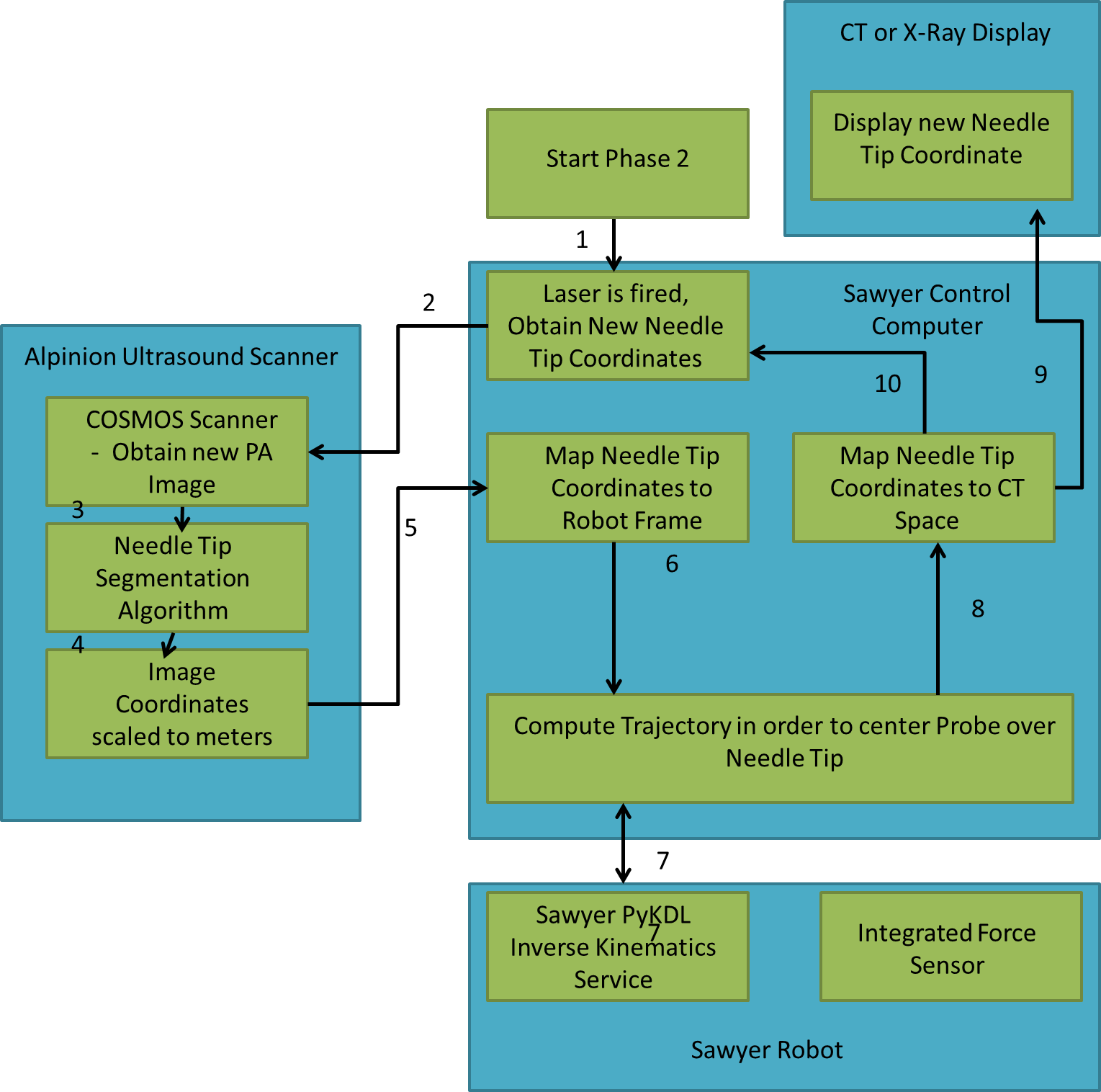
Overall System Diagram

The overall system is displayed in the figure above. A majority of Phase 1 is centered around a program titled PulsePlace.py that handles everything except computing the pose skeleton of the patient and the inverse kinematics of probe placement. The skeleton and depth images are provided through the official Kinectv2 C ++ API. I implemented communication between PulsePlace.py and the Kinectv2 API that transfers data over a TCP/IP socket connection and also parses the skeleton object for display on the PulsePlace.py GUI and also selects the middle spine point for probe placement. The software for Phase 2 is split between the Sawyer control computer and an Alpinion E-cube 12 research ultrasound scanner. The scanner performs needle segmentation locally and ships the results out.



Phase 1 Detailed Architecture

Expanding on what was stated previously, the Inverse Kinematics solutions are computed using the platform-agnostic Orocos KDL package. The company that manufactures the Sawyer robot has an official wrapper for the KDL package for their Baxter robot titled Baxter PyKDL but nothing for Sawyer, so I had to implement my own wrapper which I titled Sawyer PyKDL that is capable of position and velocity inverse kinematics. Once joint angles are computed on the control computer they get sent to Sawyer’s embedded joint controllers for execution.



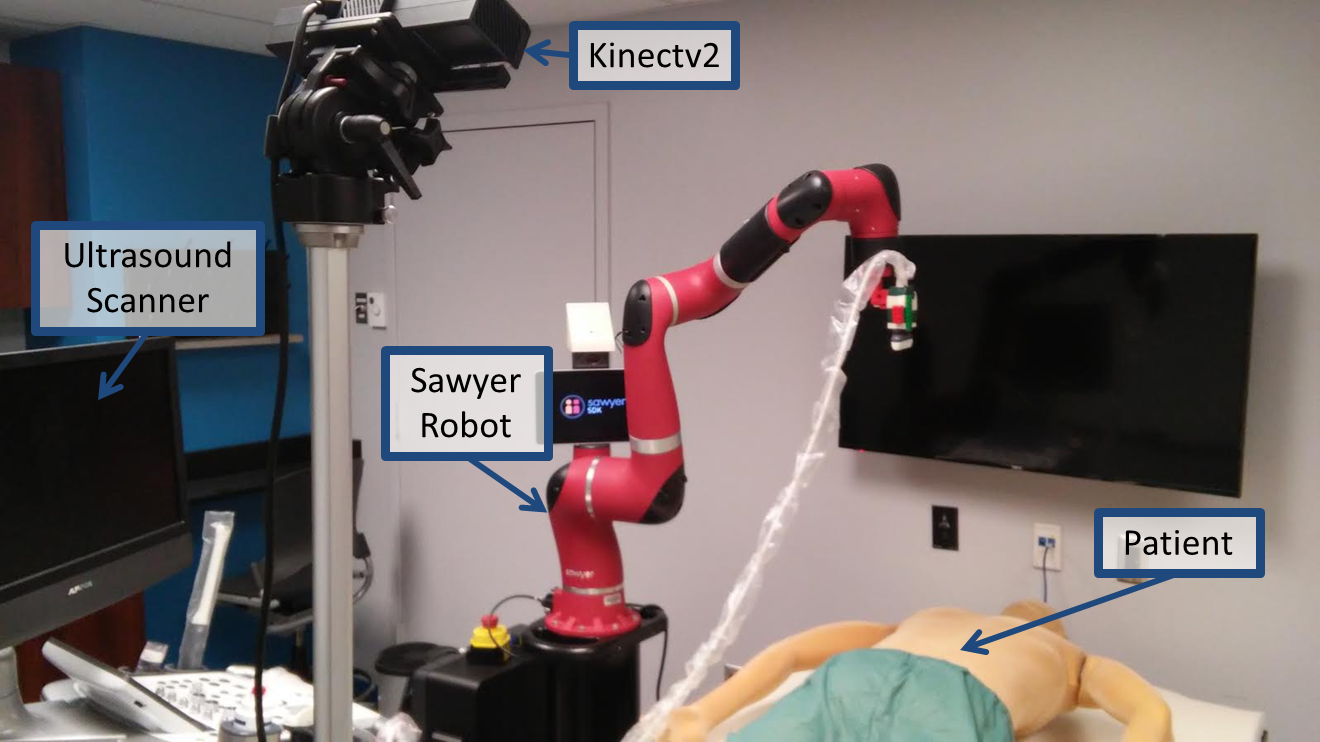
Phase 2 Detailed Architecture

Phase 2 is different from Phase 1’s structure in that the processing is fairly evenly split between Sawyer’s control computer and the Alpinion Ultrasound Scanner. Each time the laser is fired, a script running on the scanner called NeedleSegment.py grabs the photoacoustic image and extracts the needle tip coordinates from it. It then scales these coordinates to meters and sends it to a script running on the Sawyer control computer called RobotController.py that takes the needle tip coordinates and transforms them into robot space coordinates, computes the trajectory to center the probe over the coordinates, and commands the robot. As of right now Needle tip coordinates are placed onto the CT volume offline but the process could easily be made real time.

1. Methods

*Phase 1 – Autonomous Ultrasound Probe Placement*

Phase 1 starts with the user starting up PulsePlace.py and pressing the ‘Calibrate’ button to obtain a depth image from the Kinectv2 API that contains no patient and the same physical setup that will be used for the operation. Then the operator would instruct the patient to get onto the bed face down. At this point the operator would press the ‘Start’ button to obtain another depth image from the Kinectv2 API that does contain the patient. This starts up the segmentation and skeleton tracking process. PulsePlace.py requests the most recently tracked skeleton from the Kinectv2 and parses it for the coordinates of the middle spine point. This point is then mapped onto the Kinectv2 depth image using the built in Kinectv2 camera calibrations. After that happens, PulsePlace.py subtracts the calibration depth image from the patient depth image to get a rough outline of the patient. This is then refined by using morphological operators and connected component labeling to remove all objects from the depth image except the patient’s outline. For this system as is only the depth value from the middle spine point is needed, but there is potential to expand functionality here. This depth point is then mapped into Robot coordinates by using a registration transformation obtained using hand-eye calibration. The Kinectv2 is positioned on the top of a tripod and the base of the patient bed whereas the Sawyer robot is positioned at the side of the patient bed.



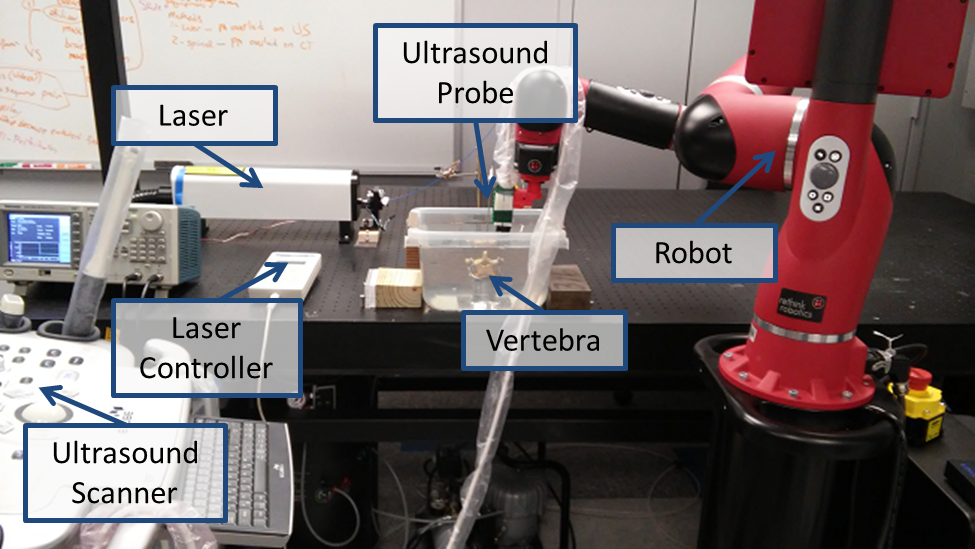
Phase 1 Set-up

Calibration is performed by having a user place a checkerboard on the patient bed and touching Sawyer’s end-effector to each of the checkerboard squares in order. Then the user finds the coordinates of the same checkerboard squares in the same order in the depth image. Then the registration transformation is computed using Arun’s method for point cloud to point cloud registration.  
Once the location of the middle spine point is known in Robot coordinates it is now necessary to move the probe to a position 10 cm above the middle spine point. To this end, PulsePlace.py sends the desired location in Cartesian coordinates to Sawyer PyKDL which computes the necessary joint angles. The Sawyer control computer then sends these to Sawyer’s joint controllers for execution. Once the desired location is reached, PulsePlace.py goes into force control mode. On the GUI there is a slider for the operator to select the desired probe placement force.   
In force control mode the robot moves the probe downward until the force it reads on its integrated force sensor matches the desired force. Once this condition is met, PulsePlace.py terminates and calls RobotController.py, the core of Phase 2

*Phase 2 – Autonomous Visual Servoing and Needle Tracking*

First I will describe the software and algorithms used in Phase 2. Then I will explain the experiment performed to visualize the needle tip.  
As mentioned previously, the core of Phase 2 is RobotController.py. This program takes as input needle tip coordinates in image space and maps them to robot space using the transformations from ultrasound image plane to robot end effector which was obtained from ultrasound probe calibration [8], and the transformation from robot end effector to robot base, which is provided by ROS. Then RobotController.py computes the distance in robot base frame that the probe must be moved in order to be centered over the needle tip. The joint angles to realize this are computed using Sawyer PyKDL and then executed by the robot’s joint controllers.  
However, in order to obtain these needle tip coordinates they must be first segmented from the photoacoustic image. This is done by NeedleSegmenter.py which runs on the Alpinion scanner. This program captures the current photoacoustic image. Then it thresholds the image on intensity values to remove most of the background noise and leave only the needle tip as well as some scattered high intensity noise bits. Then the image is thresholded on blob size to remove the scattered bits and leave only the needle tip. The centroid of this remaining blob is said to be the needle tip location. However that is not enough to be robust. In addition to the segmentation algorithm, NeedleSegmenter.py compares consecutive images and needle tip locations for spatial consistency before sending the average of the most recent segmentation results to RobotController.py.   
Additionally, these needle tip locations are saved in a log that is then used to plot all the needle tip locations onto a CT volume of the vertebra. This is done by applying a robot to CT registration transformation. Registration between robot and CT was performed by first obtaining the CT coordinates of 7 distinct features on the vertebra which include the top of the spinal process, the ends of the two transverse processes, and 4 CT markers. Then ultrasound images were obtained of these 7 features. The locations of the features were manually selected and converted to robot base coordinates using both the ultrasound image plane to robot end effector transformation and then the robot end effector to robot base transformation for each orientation. Then the 7 point pairs were used to compute a registration transformation using Arun’s method for point cloud to point cloud registration.  
MATLAB was used for loading the DICOM files for the vertebra and for plotting the points onto the volume.

The system was evaluated by performing a mock spinal operation wherein a hollow biopsy needle with an optical fiber inside of it is inserted into the cancellous tissue of a thoracic vertebra. As the needle is inserted, a pulsed laser source is fired. This creates a photoacoustic signal at the needle tip that is imaged using a nearby ultrasound probe held by the Sawyer robot.

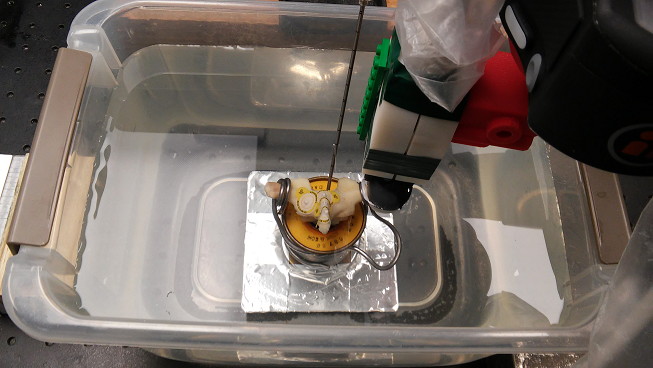


Experimental Set-up

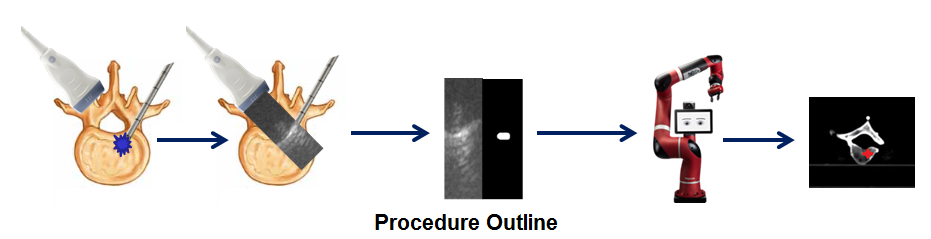
This signal is then segmented using NeedleSegmenter.py on the Alpinion scanner and recorded for later mapping to CT coordinates.   
Initially visual servoing was attempted, however it continually failed, even though it worked well on other tissue samples such as liver and steak samples. This will be more thoroughly discussed later.



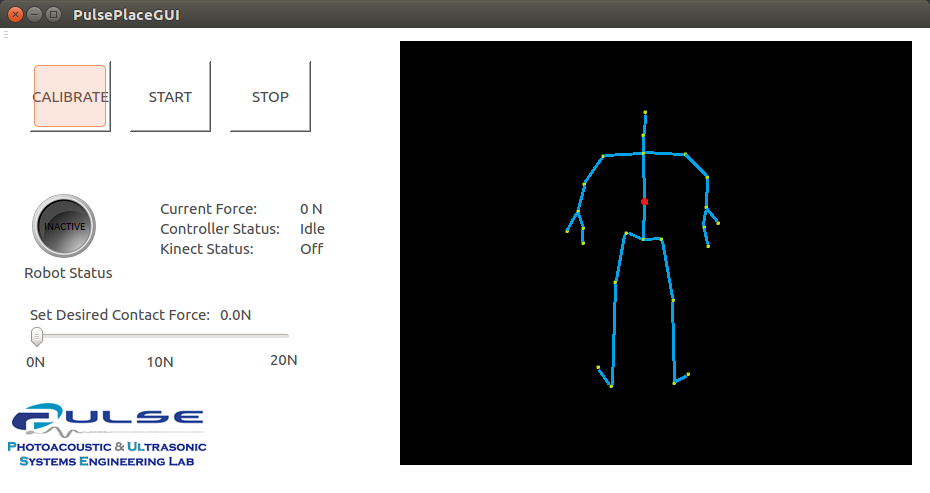
Close-up of the Set-up



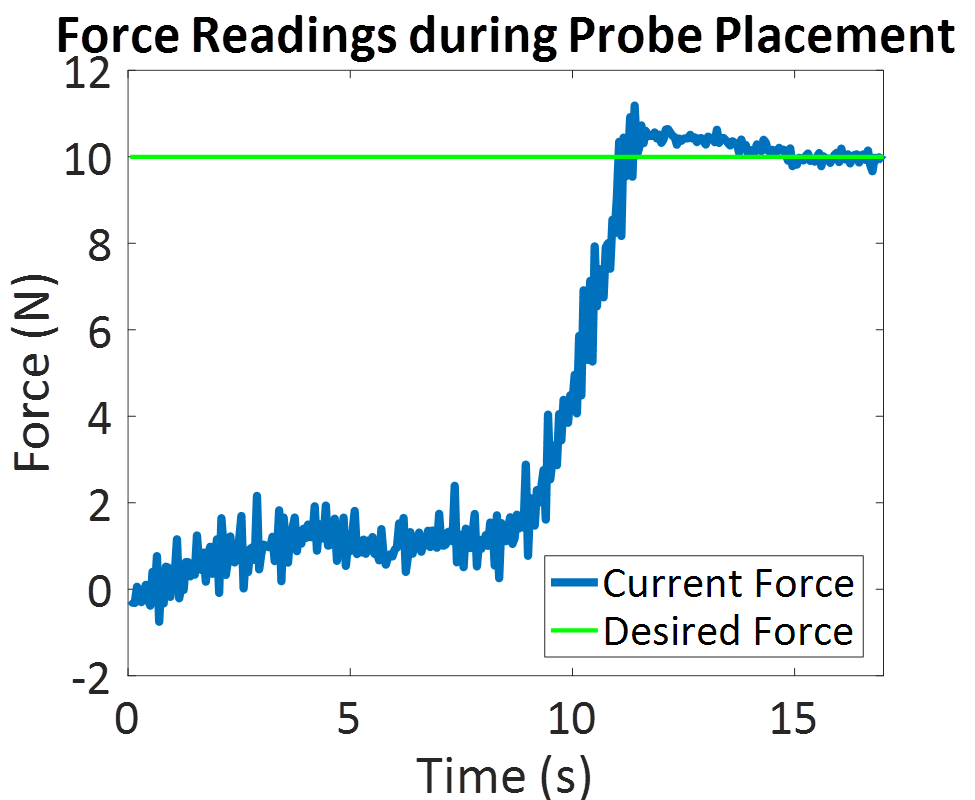
Top-Down Close-up



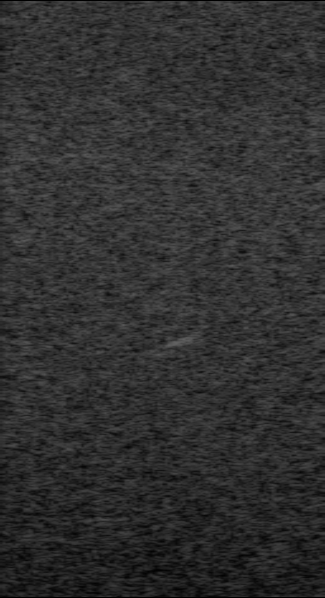
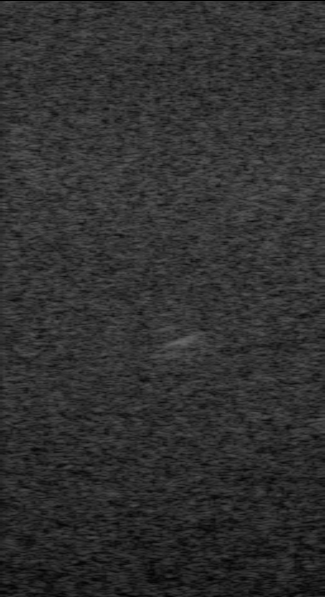
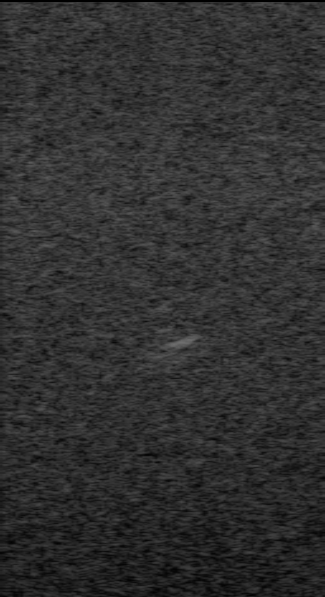
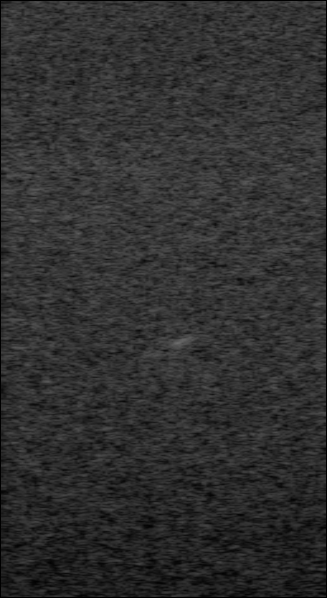
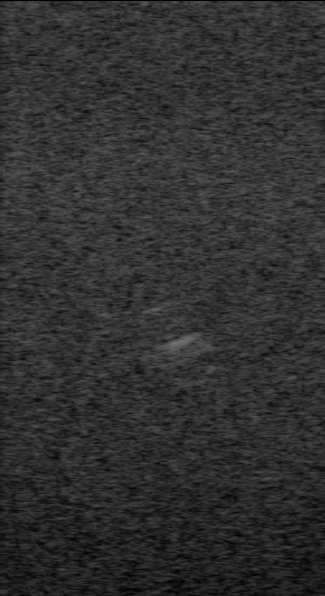
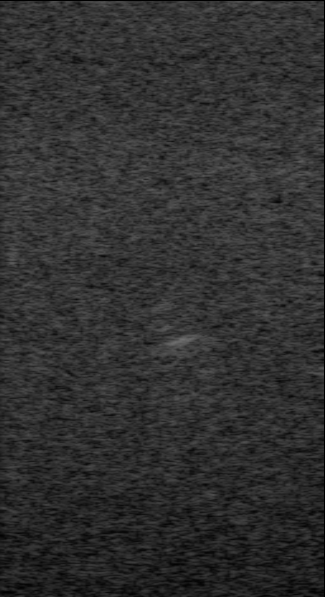
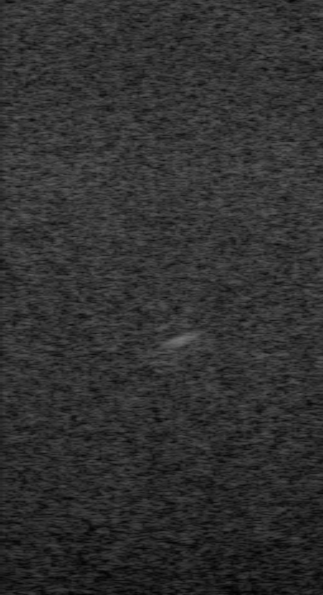
1. Results

The following image shows the GUI that was created to realize Phase 1. Samples of the large volume of code used to produce Phase 1 are in the appendix at the end of the report.

The following graph illustrates a sample of force readings during a placement of the ultrasound probe onto a patient. In this trial 10 N was the desired force and as the plot shows the robot came to settle at 10 N within a range of +/- 0.2 N.

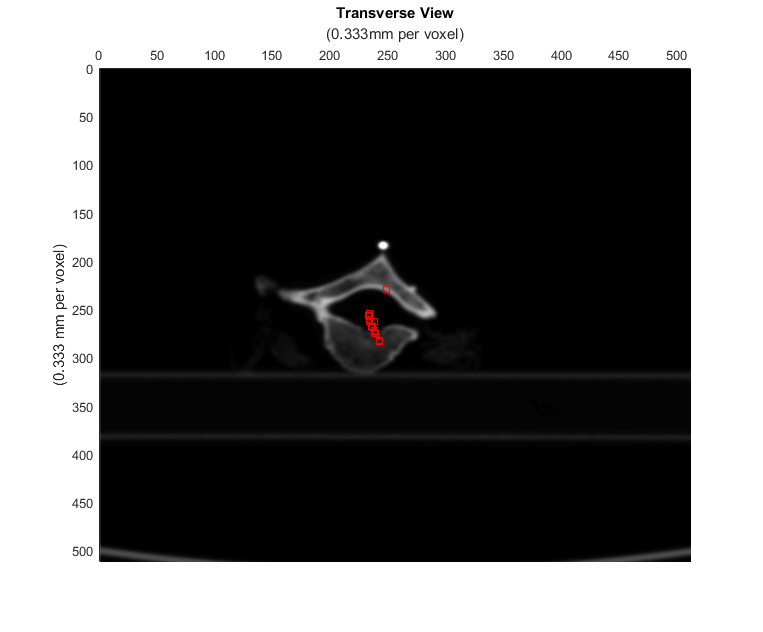


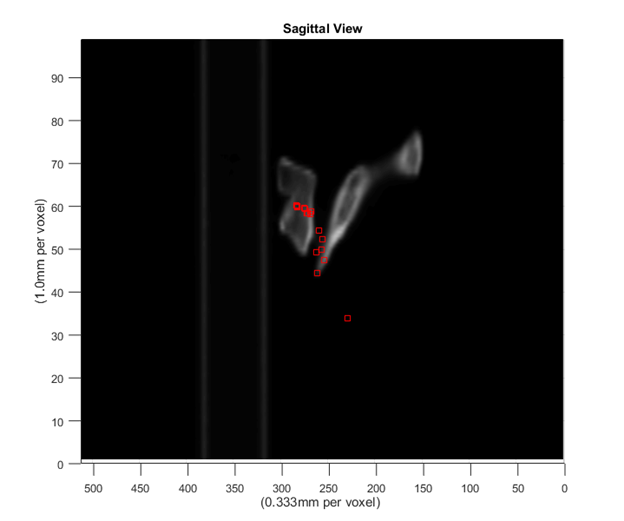
The following set of photoacoustic images all show a signal that is generated from the inside of the vertebra. The images vary in energy from being similar to energies that are used to generate photoacoustic signals in normal tissues such as chicken, liver, and steak all the way to high energies that damage your optical fiber so that you can’t get energy values to put into your final report. (So if you’re wondering why there aren’t any specific energy values, that is why)  
The laser allows the user to modify the delay between pulses, with a minimum delay of 190us. At 350 us the energy output of the fiber used in this experiment was 2.9mJ. The images below range from 210 us in the top left corner and increase by 10us up to 300us in the bottom right corner. As can be seen the image is faint at 300us but definitely still visible.

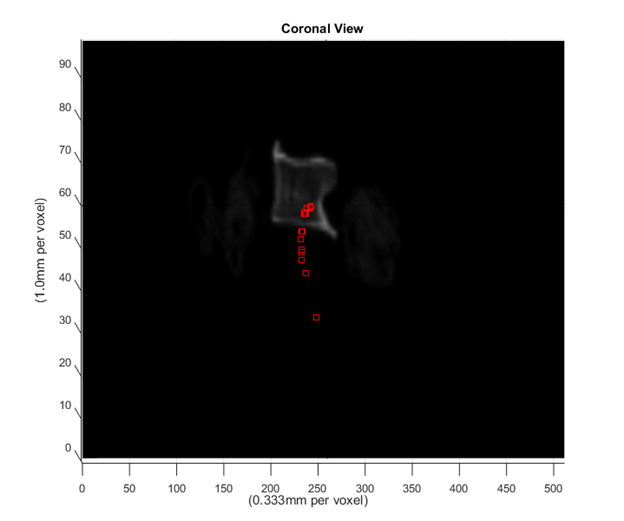


Energy Levels decrease top to bottom, left to right

Below is the result of a needle insertion trial visualized on the three primary slices of a CT image. A hole was drilled into the cancellous tissue so that the needle could be inserted. It was not drilled through the pedicles like would be seen in most spinal operations. However, drilling under the spinal process was easier since I am not an orthopaedic surgeon and I did not want to risk cracking my vertebra sample.







1. Discussion

The biggest problem experienced during this project was the failure of visual servoing when tested on the vertebra. This is was caused by the probe moving to a position that either had two or more bone interfaces, or had significantly thicker bone between the probe and the needle tip. As seen in the below diagram (where red lines are signal paths and blue lines are bone interfaces that the signals pass through), acoustic signals generated from inside the cancellous tissue will attenuate less if they pass through the pedicle than if they pass upward toward the spinal process because there will be a second acoustic reflection, greatly attenuating the signal. However it is even better if the probe can be positioned such that it can image the cancellous tissue directly and bypass the spinal process, perhaps by having the patient lean forward to increase the spacing between their vertebrae.

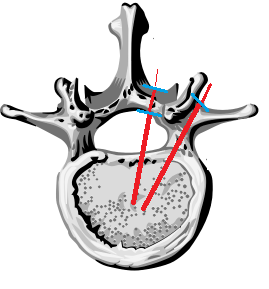


Diagram of Attenuation at Bone Interfaces

In my experiments I was never able to visualize anything if I placed the probe above the spinal process. I had to angle it near the articular facets in order to get a signal. This is not an unrealistic viewing angle, especially since the patient can be made to lean in directions to optimize signal acquisition.

1. Conclusions and Future Work

Overall the system is not perfect however it is a great foundation for future improvements.

Phase 1 can successfully place the probe onto the spine, however it still needs to be able to move to the correct vertebra to be clinically useful. This could be done by registering a CT of the patient’s spine to the segmented patient depth map or by performing a deformable registration of a statistical atlas to the segmented depth map. Then various manual techniques or automated segmentation algorithms could be applied to label the individual vertebrae.

Also, an Ultrasound -> X-Ray registration could be performed allowing the photoacoustic signals from Phase 2 to be mapped onto a projection X-Ray of the spine, lowering the radiation dose to the patient further. In the experiment performed in this report a CT image of the vertebra was used due to the increased usefulness of a CT volume for visualizing the points. However, a trained surgeon would certainly be able to make use of a display consisting of only a 2 D X Ray image, particularly if this display showed recent needle tip locations in addition to the current segmented location.

However there still remains the problem of performing visual servoing of the ultrasound probe to remain centered over the needle tip. In the experiment discussed in this report the robot was not able to successfully center over the signal and maintain vision of the signal because often it would move to a position where there was increased bone thickness or a second bone interface, greatly attenuating the signal. Switching to a beamformer that is not amplitude based (such as a short lag spatial coherence beamformer) could allow for visualizing signals through greater bone thicknesses, thus allowing visual servoing to work on signals from within the vertebra [5].

1. Management Summary

When I started this project I planned to produce the following deliverables:  
Minimum:

* Robot Control Software with
  + Human Shape Segmentation  
    Inverse Kinematics for Probe Placement  
    Force Feedback  
    A Nice GUI
* Demonstration of probe placement and segmentation algorithm

Expected:

* Exploration of whether or not it was possible to get photoacoustic signals from within a vertebra
* Develop a Needle Tip segmentation algorithm for extracting the needle tip location from these vertebral photoacoustic images

Maximum:

* Robot Control Software with
  + Needle Segmentation Algorithm
  + Visual Servoing to track a needle
  + Visual Display of the Photoacoustic Image coregistered to Ultrasound
* Demonstration of the full package on a Spine Phantom

However during the course of the project my direction changed a little bit. I completed my Expected deliverable perhaps the earliest. I purchased a lamb spine and was able to obtain photoacoustic images by shining the laser onto the inner surface of a cut-in-half vertebra. I further fulfilled this when I obtained a set of photoacoustic images at different energies from deep within the cancellous tissue of a human thoracic vertebra sample. I also was able to apply my needle segmentation algorithm onto the vertebral photoacoustic images with success.  
Next came my minimum deliverable. This one took longer because of my hesitation to use the windows kinectv2 API which required a Windows 8 or newer OS. I spent a large amount of time exploring alternative methods but in the end I cracked and just used a separate windows 8 laptop to be able to access the kinectv2 API. I was able to complete the large amount of programming required to make the software package as set out in my original proposal, however I did not really test it on people other than myself because it needs the improvements described in the future work section before it will be clinically useful.

Perhaps the greatest change came in my maximum deliverable. I opted for performing my experiment on a vertebra sample in a water bath instead of creating a human shaped phantom with an embedded spine. Also I originally said I would display my segmentation results onto an ultrasound image. That changed when it became apparent that that would not be enough to visualize the needle tip in a precise and useful manner. It turns out ultrasound of bone is very hard to understand due to the high acoustic reflection of the bone interface. Instead I opted to make a concession and allow one preoperative radiation based image that I would then use to overlay my photoacoustic signals onto. This way the radiation dose to the patient is still kept to a minimum and the surgeon still gets a useful way to visualize the location of the needle tip.

1. References

[1] Mastrangelo, Giuseppe, et al. "Increased cancer risk among surgeons in an orthopaedic hospital." *Occupational Medicine* 55.6 (2005): 498-500.

[2] Lipson, Stephen J. "Spinal-fusion surgery—advances and concerns." New England Journal of Medicine 350.7 (2004): 643-644.

[3] Kallmes, David F., et al. "A randomized trial of vertebroplasty for osteoporotic spinal fractures." New England Journal of Medicine 361.6 (2009): 569-579.

[4] Radiation Dose in X-ray and CT Exams, https://www.radiologyinfo.org/en/pdf/safety-xray.pdf

[5] Bell, Muyinatu A. Lediju, et al. "Short-lag spatial coherence beamforming of photoacoustic images for enhanced visualization of prostate brachytherapy seeds." Biomedical optics express 4.10 (2013): 1964-1977.

[6] Deyo, Richard A., and Sohail K. Mirza. "Spinal-fusion surgery-the case for restraint." The New England journal of medicine 350.7 (2004): 722.

[7] Perisinakis, Kostas, et al. "Patient Exposure and Associated Radiation Risks from Fluoroscopically Guided Vertebroplasty or Kyphoplasty 1." Radiology 232.3 (2004): 701-707.

[8] Kim, Chunwoo, et al. "Ultrasound probe and needle-guide calibration for robotic ultrasound scanning and needle targeting." IEEE Transactions on Biomedical Engineering 60.6 (2013): 1728-1734.

XI. Code Appendix

**NeedleSegmenter.py:**  
  
import os

import numpy as np

import cv2

import ImageGrab

import pickle

import copy

import time

import datetime

import math

from socket import \*

class NeedleSegmenter:

def \_\_init\_\_(self, robot\_controller\_ip="127.0.0.1", save\_video=False, verification\_limit=5, termination\_condition=1000, contrast=False):

self.robot\_controller\_ip = robot\_controller\_ip

self.save\_video = save\_video

self.verification\_limit = verification\_limit

self.termination\_condition = termination\_condition

self.contrast = contrast

self.x1 = 785

self.x2 = 1112

self.y1 = 153

self.y2 = 753

self.width = self.x2 - self.x1

self.height = self.y2 - self.y1

self.double\_width = self.width \* 2

self.log\_file = open("log.txt", 'wb')

if self.save\_video:

self.fourcc = cv2.VideoWriter\_fourcc(\*'DIVX')

self.frame\_count = 0

self.vid\_count = 0

self.video = cv2.VideoWriter('needlesegment' + str(self.vid\_count) + '.avi', self.fourcc, 20.0,

(self.double\_width, self.height))

self.scale\_matrix = pickle.load(open('scale\_matrix', 'rb'))

self.scale\_matrix[0][0] = 0.000108024

self.scale\_matrix[0][0] = 0.00011744966

print self.scale\_matrix

self.isocenter = np.array([self.width/2, 0.0])

self.port = 13000

self.addr = (self.robot\_controller\_ip, self.port)

self.UDPSock = socket(AF\_INET, SOCK\_DGRAM)

self.nnt\_count = 0

def run(self):

while self.nnt\_count < self.termination\_condition:

#start = time.time()

print self.frame\_count

if self.save\_video and self.frame\_count >= 50:

self.video.release()

self.video = cv2.VideoWriter('needlesegment' + str(self.vid\_count) + '.avi', self.fourcc, 20.0, (self.double\_width, self.height))

self.frame\_count = 0

self.vid\_count += 1

print "SAVING VIDEO"

result\_queue = []

coordinate\_queue = []

C = 0

CNR = 0

SNR = 0

# Perform segmentation on 'verification\_limit' consecutive frames

for i in range(0, self.verification\_limit):

# screengrab

img = ImageGrab.grab((self.x1, self.y1, self.x2, self.y2))

img = cv2.cvtColor(np.array(img), cv2.COLOR\_RGB2GRAY)

#if self.save\_video:

img\_orig = copy.deepcopy(img)

# segment needle tip

found, coords, img\_seg = self.needle\_segment(img)

if self.save\_video:

img\_orig = cv2.cvtColor(np.array(img\_orig), cv2.COLOR\_GRAY2RGB)

img\_fused = np.concatenate((img\_orig, img\_seg), axis=1)

self.video.write(img\_fused)

self.frame\_count += 1

# append segmentation results to their corresponding queues

result\_queue.append(found)

coordinate\_queue.append(coords)

# If all is well, do visual servoing as planned

if self.all\_true(result\_queue) and self.coordinates\_spatially\_consistent(coordinate\_queue):

self.nnt\_count = 0

data = np.mean(coordinate\_queue, 0) # Average the most recent needle tip locations [pixels]

if self.save\_video:

#img\_orig = cv2.cvtColor(np.array(img\_orig), cv2.COLOR\_GRAY2RGB)

cv2.circle(img\_seg, (int(data[0]),int(data[1])), 5, (0, 255, 0), 2)

cv2.imshow('needle', img\_seg)

cv2.waitKey(1)

img\_fused = np.concatenate((img\_orig, img\_seg), axis=1)

self.video.write(img\_fused)

self.frame\_count += 1

data = data - self.isocenter

data\_m = self.scale\_matrix[0:1, 0:1] \* data # Convert pixel coordinates into [m]

print data\_m

self.UDPSock.sendto("val\_" + str(data\_m), self.addr)

print "sent"

cv2.waitKey(1000)

# If we are finding needle tips but they are all over the place, assume they are all bad

elif self.all\_true(result\_queue) and not self.coordinates\_spatially\_consistent(coordinate\_queue):

self.nnt\_count += 1

print "Scatter: not found!"

data = "msg\_" + "not\_found" + str(self.nnt\_count)

# If we do not find the needle tip consistently, we count how many times this has happened

# and we send a message to the robot controller to move the probe back and forth to try and find the needle tip

elif not self.all\_true(result\_queue):

self.nnt\_count += 1

if self.nnt\_count > self.termination\_condition:

data = "msg\_" + "shut\_down"

self.UDPSock.sendto(data, self.addr)

break

data = "msg\_" + "not\_found" + str(self.nnt\_count)

self.UDPSock.sendto(data, self.addr)

cv2.waitKey(500)

print "Results: ", found, coords, data, self.nnt\_count

if self.contrast:

self.log\_file.write("Results: " + str((found, coords, data, self.nnt\_count, C, CNR, SNR)) + " " + str(

datetime.datetime.utcnow()) + "\n")

else:

self.log\_file.write("Results: " + str((found, coords, data, self.nnt\_count)) + " " + str(

datetime.datetime.utcnow()) + "\n")

self.log\_file.flush()

self.UDPSock.close()

self.log\_file.close()

os.\_exit(0)

def all\_true(self, results\_queue):

'''

Purpose:

Determines if the results in the results queue are all True, meaning that a needle tip has been reliably found in

last few frames, and False otherwise

Inputs:

results\_queue -> list of booleans, recording the results of the last batch of needle segmentations

Outputs:

Boolean -> True if all the results are True, False otherwise

'''

for i in results\_queue:

if i is not True:

return False

return True

def coordinates\_spatially\_consistent(self, coordinate\_queue, k=0.10):

'''

Purpose: Determines if all the needle tip coordinates in the batch are close to each other within some boundary.

This helps because if the segmented needle tip coordinates are all over the place, it is likely they are all wrong

so we want to do nothing as a defensive measure

Inputs:

coordinate\_queue -> list of 2-element tuples, where each tuple is the x,y coordinate of a needle tip location

k -> parameter used to determine what it means for coordinates to be 'close'

e.g. all needle tip locations must be with +/- k\*100 percent of the average needle tip location

Outputs:

Boolean -> True if all needle tip coordinates are close to each other, False otherwise

'''

coordinate\_average = np.mean(coordinate\_queue, 0)

for i in coordinate\_queue:

if not self.is\_close(i[0], coordinate\_average[0], k) or not self.is\_close(i[1], coordinate\_average[1], k):

return False

return True

def is\_close(self, a, b, k):

'''

Purpose:

Return true if 'a' is between 'b' +/- k\*100 percent

Inputs:

a -> number to be tested

b -> number 'a' is to be tested against

k -> parameter to determine what 'close' means as described in the Purpose section

Outputs:

Boolean -> True is a is close to b, False otherwise

'''

if b \* (1 - k) <= a <= b \* (1 + k):

return True

def needle\_segment(self, img, k=30, num\_chunks=20):

'''

Purpose:

Segments a Photoacoustic Image for the location of a needle tip

Inputs:

img -> typical numpy array image (only the photoacoustic image, assumes the image has been cropped)

k -> a constant used for thresholding. Basically if the difference between the max and avg intensity for a slice

is less than k we black out that slice. This is a good intensity invariant way of ignoring background slices

num\_chunks -> the number of horizontal slices to make in the image for the binary thresholding portion

Outputs:

Boolean -> True if a needle tip is detected, False if no needle tip detected

return\_coords -> (x,y) coordinate of centroid of needle tip region

Notes:

This algorithm makes a few assumptions on how to find the needle tip.

1. The needle tip is relatively bright in the image

2. If many possible needle tips are found, and there is no strong outlier, none are the needle tip

In general if you have a strong laser and the needle tip is in the ultrasound plane the algorithm will find it,

and if it is not in the plane at all the algorithm will correctly determine the needle tip is absent.

The ambiguities come in when the needle tip is on the very edge of the ultrasound plane or the laser is not

very powerful

'''

img\_height = len(img)

chunk\_size = int(img\_height / num\_chunks)

# Go through the photo acoustic image from top to bottom in 'chunks'

# Perform binary thresholding chunk by chunk

im\_max = img.max()

ret, img[0:chunk\_size, :] = cv2.threshold(img[0:chunk\_size, :], 255, 255, cv2.THRESH\_BINARY)

for i in range(0, num\_chunks):

chunk\_max = img[chunk\_size \* i:chunk\_size \* (i + 1), :].max() # Highest intensity in chunk

chunk\_average = np.average(img[chunk\_size \* i:chunk\_size \* (i + 1), :]) # Avg intensity in chunk

if chunk\_max - chunk\_average < k: # If Max and Avg intensity are very close, this is a background chunk

threshold = chunk\_max # and we make the threshold equal to the maximum value in the chunk

else:

threshold = img[chunk\_size \* i:chunk\_size \* (i + 1),

:].max() - 10 # Else, threshold around the max value

threshold = img.max() - 10

#print img.max()

ret, thresh = cv2.threshold(img[chunk\_size \* i:chunk\_size \* (i + 1), :], threshold, 255, cv2.THRESH\_BINARY)

img[chunk\_size \* i:chunk\_size \* (i + 1), :] = thresh

# Apply thresholding to original image

# Dilate and then erode to connect very close 'blobs'

kernel = np.ones((3, 3), np.uint8)

img = cv2.dilate(img, kernel, iterations=1)

img = cv2.erode(img, kernel, iterations=1)

ret, img[img\_height - 20:img\_height, :] = cv2.threshold(img[img\_height - 20:img\_height, :], 255, 255,

cv2.THRESH\_BINARY)

# Perform connected component labelling

connectivity = 4

output = cv2.connectedComponentsWithStats(img, connectivity, cv2.CV\_32S)

num\_labels = output[0]

labels = output[1]

stats = output[2]

centroids = output[3]

# If we found nothing to label (except the background) return False

if num\_labels is 1:

img = cv2.cvtColor(np.array(img), cv2.COLOR\_GRAY2RGB)

cv2.line(img, (self.width/2, 0), (self.width/2, 1000), (0, 255, 0), 1)

cv2.imshow('needle', img)

cv2.waitKey(1)

mask = None

return False, 0, img

# If we found lots of potential needle tips but there is no strong outlier, return False

if num\_labels > 20 and not self.strong\_outlier(stats[1:, cv2.CC\_STAT\_AREA]):

img = cv2.cvtColor(np.array(img), cv2.COLOR\_GRAY2RGB)

cv2.line(img, (self.width/2, 0), (self.width/2, 1000), (0, 255, 0), 1)

cv2.imshow('needle', img)

cv2.waitKey(1)

mask = None

return False, 0, img

# If we found a needle tip, return the coordinates of its centroid

argmax = np.argmax(stats[1:, cv2.CC\_STAT\_AREA]) + 1

return\_coords = tuple(centroids[argmax].astype(dtype=np.uint32))

img = cv2.cvtColor(np.array(img), cv2.COLOR\_GRAY2RGB)

#cv2.circle(img, return\_coords, 2, (0, 0, 255), 1)

#cv2.circle(img, return\_coords, 5, (0, 0, 255), 1)

cv2.line(img, (self.width/2, 0), (self.width/2, 1000), (0, 255, 0), 1)

cv2.imshow('needle', img)

cv2.waitKey(1)

return True, return\_coords, img

def strong\_outlier(self, areas, k=3):

'''

Purpose: Looks in an array of values to see if there is exactly one that surpasses the average

area by a factor of k, a 'strong outlier'

Inputs:

areas -> an array of values

k -> a factor that determines what is considered a strong outlier. e.g. a strong outlier

must be k times bigger than the average value

Outputs:

Boolean -> True if there is one strong outlier, False if no strong outliers or multiple strong outliers

'''

avg = np.average(areas)

if areas.max() < 5:

return False

num\_str\_outliers = 0

for i in areas:

if i > k \* avg:

num\_str\_outliers += 1

if num\_str\_outliers == 1:

return True

return False

def compute\_contrast(self, img, mask):

"""

Purpose: Computes the Contrast, Contrast to Noise Ratio, and Signal to Noise ratio at the depth a previously

segmented point of interest is located

:param img: Numpy array-like Ultrasound or Photoacoustic Image

:param mask: Numpy array-like mask image that is 255 where the point of interest is, and 0 everywhere else

:return: C -> contrast

CNR -> contrast to noise ratio

SNR -> signal to noise ratio

"""

foreground = 0 # Intensity sum for foreground pixels

background = 0 # Intensity sum for background pixels

n\_f = 0 # Number of foreground pixels

n\_b = 0 # Number of background pixels

Si = 0 # Mean intensity of foreground pixels

So = 0 # Mean intensity of background pixels

sigi = 0 # Standard deviation of foreground pixels

sigo = 0 # Standard deviation of background pixels

signal\_record = [] # List of rows in image where point of interest is (allows for speed up)

for i in range(0, len(img)):

signal = False

temp\_bg = 0 # We only want to add in background pixels from rows where atleast one foreground

temp\_n\_bg = 0 # pixel is located, thus we store our results in temporary variables

for j in range(0, len(img[0])):

if mask[i][j] == 0:

temp\_bg += img[i][j]

temp\_n\_bg += 1

elif mask[i][j] == 255:

foreground += img[i][j]

n\_f += 1

signal = True

if signal == True: # If we found atleast one foreground pixel in this row, we save our results

n\_b += temp\_n\_bg

background += temp\_bg

signal\_record.append(i)

Si = foreground / float(n\_f)

So = background / float(n\_b)

ssumi = 0 # sum of square foreground (img - mean) values, used for standard deviation

ssumo = 0 # sum of square background (img - mean) values, used for standard deviation

for i in signal\_record: # Only go through rows known to have foreground pixels

for j in range(0, len(img[0])):

if mask[i][j] == 0:

ssumo += (img[i][j] - So) \*\* 2

if mask[i][j] == 255:

ssumi += (img[i][j] - Si) \*\* 2

sigi = math.sqrt(ssumi)

sigo = math.sqrt(ssumo)

print Si, So, background, n\_b, foreground, n\_f

C = 20\*np.log10(float(Si) / float(So))

CNR = math.fabs(Si - So) / math.sqrt(float(sigi)\*\*2 + float(sigo)\*\*2)

SNR = So / sigo

print "C: ", C, " CNR: ", CNR, " SNR: ", SNR, "\n"

return C, CNR, SNR

if \_\_name\_\_ == '\_\_main\_\_':

ns = NeedleSegmenter(robot\_controller\_ip="169.254.98.243", save\_video=True, contrast=False)

ns.run()

**SawyerInteraction.py**:  
  
'''

Author: Joshua Shubert, PULSE Lab - Johns Hopkins University

Contact Info: jshubert1994@gmail.com / jshuber2@jhu.edu

Version: 1.1

Last Updated: 1/18/2017

Description:

Some simple wrapper functions for Sawyer's API with more intuitive functions

and quicker access to useful data.

E.g. if one wants to get a snapshot from the head camera, just call get\_camera\_image('head\_camera')

Its currently a WIP, always adding new functions as I find it necessary

Disclaimer: Some portions of this library are cannibalized versions of the example scripts on

the intera / Rethink Robotics wiki, though you could have assumed that.

Do not hesitate to contact me with questions!

'''

'''

List of Global Variables:

global\_tf\_listener

global\_camera\_image

'''

# --------------------------------------- #

# General libraries

# --------------------------------------- #

import time

import cv2

from cv\_bridge import \*

import numpy as np

import math

import argparse

import os

import pickle

import scipy.io

# --------------------------------------- #

# Robot libraries

# --------------------------------------- #

import tf

import rospy

import intera\_interface

from sensor\_msgs.msg import (

Image,

JointState,

)

from geometry\_msgs.msg import (

PoseStamped,

Pose,

Point,

Quaternion,

)

from std\_msgs.msg import Header

from sensor\_msgs.msg import JointState

from intera\_core\_msgs.srv import (

SolvePositionIK,

SolvePositionIKRequest,

)

# --------------------------------------- #

# Custom libraries

# --------------------------------------- #

#import eCube

def activate\_robot(sawyer\_ip="169.254.9.126"):

'''

Purpose:

Pings sawyer to ensure they are connected

and enables sawyer.

In general, call this function first!

Inputs:

sawyer\_ip -> your robot's IP address as reported when you run 'avahi-browse -a -r' from the terminal

Note: Ensure the IP address is correct by running

'avahi-browse -a -r' from the terminal

'''

print "Starting up..."

try:

(os.system("ping -c 1 " + sawyer\_ip) == 0)

os.system("rosrun intera\_interface enable\_robot.py -e")

rospy.init\_node('Hello\_Sawyer')

print "Sawyer says hello!"

return 1

except:

print "Error connecting and enabling Sawyer!"

return 0

def change\_display(img):

'''

Purpose:

Updates the image displayed on sawyer's face

Resolution is 1024 x 600

Larger images will be cut off

Smaller images will be displayed started at the top left corner of the screen

Inputs:

img -> An image in numpy array format e.g. from cv2.imread()

Note: could easily be modified to work with image file directory instead

'''

rospy.init\_node("Hello\_Sawyer")

msg = CvBridge().cv2\_to\_imgmsg(img, encoding="bgr8")

pub = rospy.Publisher('/robot/head\_display', Image, latch=True)

pub.publish(msg)

rospy.sleep(1)

return

def manually\_record\_angles(delay, n, mode='auto'):

'''

Purpose:

Records joint angles for play back or saving

The joint angles are recorded either periodically or on user command

Inputs:

delay -> amount of time (s) to wait between joint angle reads (~5s recommended) (ignore if mode='prompted')

n -> number of readings to take

mode -> method of data collection. Two options:

'auto' - the joint angles will be read every 'delay' seconds

'prompted' - the joint angles will be read only when the user presses enter

Outputs:

angle\_set -> set of joint angles

'''

limb = intera\_interface.Limb('right')

angle\_set = []

if mode == 'auto':

print "Reading joint angles once every " + str(delay) + " seconds..."

for i in range(0, n):

time.sleep(delay)

angles = limb.joint\_angles()

print "Pose " + str(i) + ": " + str(angles)

angle\_set.append(angles)

print "Joint angle set " + str(i + 1) + " recorded"

elif mode == 'prompted':

print "Reading joint angles when enter is pressed..."

limb = intera\_interface.Limb('right')

cv2.waitKey(0)

cv2.waitKey(0)

for i in range(0, n):

raw\_input("Press enter to read joint angles...")

angles = limb.joint\_angles()

# print "Pose " + str(i) + ": " + str(angles)

angle\_set.append(angles)

print "Joint angle set " + str(i + 1) + " recorded!"

print angles

return angle\_set

def repeat\_angles(angle\_set, speed=0.1, mode='nophoto'):

'''

Purpose:

Plays back motion from a set of angles

Inputs:

angle\_set -> list of sawyer joint angles

speed -> maximum speed for joint playbacks. speed > 0.3 gets jerky

mode -> determines if ultrasound images are captured. Two options:

nophoto - no ultrasound images are captured during playback

photo - images are captured at each terminal robot position

The image received depends on SS.py running on the alpinion system

Outputs:

imgs -> List of images

Note: ignore and comment out the 'photo' portion if not using alpinion ultrasound system

'''

rospy.init\_node('Hello\_Sawyer')

limb = intera\_interface.Limb('right')

limb.set\_joint\_position\_speed(speed)

if mode == 'nophoto':

print "Following obtained joint positions..."

for pose in angle\_set:

limb.move\_to\_joint\_positions(pose)

time.sleep(2)

return

elif mode == 'photo':

imgs = []

print "Following obtained joint positions..."

for pose in angle\_set:

limb.move\_to\_joint\_positions(pose)

#imgs.append(eCube.pingAlpinionforSS())

print "Grabbed an image..."

time.sleep(2)

return imgs

return

def inv\_kin\_pos(p,q):

'''

Purpose:

Solves for the joint angles needed to move a robot to a position p with orientation q

Inputs:

p -> a 3x1 cartesian vector describing the end effector final position relative to the robot base

q -> a 4x1 quaternion vector describing the end effector final orientation relative to the robot base

Outputs:

joint\_angles -> a list of joint angles for joints 0 ... 7 to be applied to the robots joints to satisfy (x,q)

'''

limb = intera\_interface.Limb('right')

ns = "ExternalTools/right/PositionKinematicsNode/IKService"

iksvc = rospy.ServiceProxy(ns, SolvePositionIK)

ikreq = SolvePositionIKRequest()

hdr = Header(stamp=rospy.Time.now(), frame\_id='base')

poses = {

'right': PoseStamped(

header=hdr,

pose=Pose(

position=Point(

x=p[0],

y=p[1],

z=p[2],

),

orientation=Quaternion(

x=q[0],

y=q[1],

z=q[2],

w=q[3],

),

),

),

}

# Add desired pose for inverse kinematics

ikreq.pose\_stamp.append(poses['right'])

# Request inverse kinematics from base to "right\_hand" link

ikreq.tip\_names.append('right\_hand')

try:

rospy.wait\_for\_service(ns, 5.0)

resp = iksvc(ikreq)

except (rospy.ServiceException, rospy.ROSException), e:

rospy.logerr("Service call failed: %s" % (e,))

return False

# Check if result valid, and type of seed ultimately used to get solution

if (resp.result\_type[0] > 0):

# Format solution into Limb API-compatible dictionary

joint\_angles = dict(zip(resp.joints[0].name, resp.joints[0].position))

else:

rospy.loginfo("INVALID POSE - No Valid Joint Solution Found.")

return False

return joint\_angles

def move\_to(p, q, speed=0.1):

'''

Purpose:

Moves the robot end effector to position 'p' with orientation 'q' at a defined speed

Inputs:

p -> a 3x1 cartesian vector describing the end effector final position relative to the robot base

q -> a 4x1 quaternion vector describing the end effector final orientation relative to the robot base

speed -> the speed at which to complete the motion. Too small of a speed lowers accuracy of motion

Outputs:

success -> Boolean that is true if the motion was completed successfully

Note: This is essentially a wrapper for inv\_kin\_pos()

'''

limb = intera\_interface.Limb('right')

limb.set\_joint\_position\_speed(speed)

ik\_result = inv\_kin\_pos(p, q)

if not ik\_result:

return False

limb.move\_to\_joint\_positions(ik\_result)

return True

def quat2rot(q):

'''

Purpose:

Converts quaternions into rotation matrices

Input:

q -> x,y,z,w quaternion

Output:

R -> 3x3 Rotation Matrix

'''

x = q[0]

y = q[1]

z = q[2]

w = q[3]

mag = math.sqrt(x\*\*2 + y\*\*2 + z\*\*2 + w\*\*2)

x /= mag

y /= mag

z /= mag

w /= mag

R = np.array(([1 - 2\*y\*\*2 - 2\*z\*\*2, 2\*x\*y - 2\*z\*w, 2\*x\*z + 2\*y\*w],

[2\*x\*y + 2\*z\*w, 1 - 2\*x\*\*2 - 2\*z\*\*2, 2\*y\*z - 2\*x\*w],

[2\*x\*z - 2\*y\*w, 2\*y\*z + 2\*x\*w, 1 - 2\*x\*\*2 - 2\*y\*\*2]))

return R

def rot2quat(R):

'''

Purpose:

Converts 3x3 rotation matrices to their quaternion representation

Can fail on invalid rotation matrices

Input:

R -> 3x3 Rotation Matrix

Output:

q -> 4x1 quaternion vector

Note: Improve this later by checking if rotation matrix is valid

'''

w = math.sqrt(1 + R[0][0] + R[1][1] + R[2][2]) / 2

x = (R[2][1] - R[1][2])/(4\*w)

y = (R[0][2] - R[2][0])/(4\*w)

z = (R[1][0] - R[0][1])/(4\*w)

q = np.array(([w,x,y,z]))

return q

def manual\_transform\_test(n):

'''

Purpose:

Allows you to move the robot through n poses manually and then print the current transform by pressing enter

Inputs:

n -> the number of poses to go through

'''

tf\_init()

time.sleep(2)

for i in range(0, n):

print str(i) + ": "

raw\_input("Press enter to print transform...")

print get\_tf()

return

def activate\_camera\_feed(cam="right\_hand\_camera", canny=False):

'''

Purpose:

Activates one of Sawyer's cameras, and shows the video feed.

The individual image frames are accessible here outside of the ros node callback,

unlike in the default Sawyer script so image processing can be done and used elsewhere in your script

Inputs:

cam -> string specifying which camera to get the image from. "right\_hand\_camera" or "head\_camera"

canny -> boolean that determines whether to do edge detection on the camera images

Outputs:

img -> the captured image

Notes:

You may get a 'camera\_image' is not defined error if time.sleep(1) is not enough. It should be enough!

So the error may be caused if a frozen opencv window is hanging around.

'''

camera\_init(cam) # Activate the desired camera

time.sleep(1) # Wait for image frames to start being collected

while True:

cv2.namedWindow("camera\_feed", 0)

img = camera\_image # Make local copy of global variable

if canny:

# Do simple edge detection

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

blurred = cv2.GaussianBlur(gray, (3, 3), 0)

get\_edge = cv2.Canny(blurred, 10, 100)

img = np.hstack([get\_edge])

# refresh the image on the screen

cv2.imshow("camera\_feed", img)

cv2.waitKey(3)

return

def get\_camera\_image(cam='head\_camera', display=False, save=False):

'''

Purpose: Pings the specified camera for the current image frame

Inputs:

cam -> either 'right\_hand\_camera' or 'head\_camera', the camera to get an image from

display -> Boolean specifying whether or not to display the captured image in an opencv window

save -> Boolean specifying whether or not to save the image file

Outputs:

img -> the current image as opencv image type

Note: If you want to continually get images from the same camera, move camera\_init() outside

of this function and remove the sleep step and then just repeatedly call this function

(perhaps in a loop)

'''

camera\_init(cam)

time.sleep(0.1)

img = global\_camera\_image

if save:

cv2.imwrite('img.jpeg', img, )

if display:

cv2.imshow("camera\_feed", img)

cv2.waitKey(0)

return img

def camera\_init(cam):

'''

Purpose: Initializes the camera and uses the callback function to update

a global variable 'camera\_image' with the current frame

Inputs:

cam -> either 'right\_hand\_camera' or 'head\_camera', the camera to activate

Outputs:

global\_camera\_image -> global variable containing the most recent image frame

in the typical opencv image format

Notes:

This is called from activate\_camera\_feed, but if you want to call get\_camera\_image()

you should call camera\_init prior

'''

def camera\_callback(img\_data):

'''

Purpose: This function is automatically called each time a new frame of image

data is acquired if camera\_init() was run.

Inputs:

img\_data -> contains the current image frame as a ros image type

Outputs:

camera\_image -> contains the current image frame as an opencv image type (as a global!)

'''

bridge = CvBridge()

global global\_camera\_image

global\_camera\_image = bridge.imgmsg\_to\_cv2(img\_data, "bgr8")

return

def clean\_shutdown():

print("Shutting down camera\_display node.")

cv2.destroyAllWindows()

rp = intera\_interface.RobotParams()

valid\_cameras = rp.get\_camera\_names()

if not valid\_cameras:

rp.log\_message(("Cannot detect any camera\_config"

" parameters on this robot. Exiting."), "ERROR")

return

arg\_fmt = argparse.RawDescriptionHelpFormatter

parser = argparse.ArgumentParser(formatter\_class=arg\_fmt,

description=camera\_init.\_\_doc\_\_)

parser.add\_argument(

'-c', '--camera', type=str, default=cam,

choices=valid\_cameras, help='Setup Camera Name for Camera Display')

parser.add\_argument(

'-r', '--raw', action='store\_true',

help='Specify use of the raw image (unrectified) topic')

parser.add\_argument(

'-e', '--edge', action='store\_true',

help='Streaming the Canny edge detection image')

args = parser.parse\_args()

print("Initializing node... ")

rospy.init\_node('camera\_display', anonymous=True)

camera = intera\_interface.Cameras()

if not camera.verify\_camera\_exists(args.camera):

rospy.logerr("Invalid camera name, exiting the example.")

return

camera.start\_streaming(args.camera)

rectify\_image = False

camera.set\_callback(args.camera, camera\_callback,

rectify\_image=rectify\_image)

rospy.on\_shutdown(clean\_shutdown)

rospy.loginfo("Camera\_display node running. Ctrl-c to quit")

def tf\_init():

'''

Purpose:

Creates a tf.TransformListener object that will have a continually updated transformation tree

that can be probed using get\_tf()

Note: This MUST be called once before get\_tf()

'''

global global\_tf\_listener

global\_tf\_listener = tf.TransformListener()

rate = rospy.Rate(10.0) # [Hz]

time.sleep(2)

def get\_tf(source='base', destination='right\_hand'):

'''

Purpose:

Probes the global tf\_listener object created by tf\_init() for a coordinate transformation 'from here to there'

Inputs:

source -> the 'from here' frame

destination -> the 'to there' frame

Outputs:

tf\_p -> a 3x1 cartesian vector described the destination frame's distance from the source frame in [m]

tf\_q -> a 4x1 quaternion vector describing the rotation from source to destination

'''

tf\_p, tf\_q = global\_tf\_listener.lookupTransform(source, destination, rospy.Time())

return tf\_q, tf\_p

def get\_joint\_torque():

'''

Purpose:

WIP

'''

rospy.init\_node

return

def build\_frame(R, t):

'''

Purpose:

Converts a 3x3 rotation matrix and a 3x1 translation vector into a 4x4 homogenous frame transformation matrix

Inputs:

R -> 3x3 rotation matrix

t -> 3x1 cartesian matrix

Outputs:

F -> 4x4 homogenous frame transformation matrix

'''

F = np.matrix((

[R[0,0], R[0,1], R[0,2], t[0]],

[R[1,0], R[1,1], R[1,2], t[1]],

[R[2,0], R[2,1], R[2,2], t[2]],

[ 0, 0, 0, 1]

))

return F

def IK\_test(num\_trials=25, test\_speed=0.05):

'''

Purpose:

Moves sawyers end effector back and forth

Inputs:

num\_trials -> Number of times to move back and forth

test\_speed -> Speed at which to move Sawyer

'''

forward = np.array((0.01, 0.01, 0.01))

backward = np.array((-0.01, -0.01, -0.01))

tf\_init()

err = []

trial = 0

while trial < num\_trials:

q, p\_bf = get\_tf()

move\_to(p\_bf + forward, q, speed=test\_speed)

q, p\_af = get\_tf()

err.append(np.linalg.norm(np.subtract(p\_bf + forward, p\_af)))

print "Diff: ", np.linalg.norm(np.subtract(p\_bf + forward, p\_af))

print np.subtract(p\_bf + forward, p\_af)

time.sleep(1)

move\_to(p\_af + backward, q, speed=test\_speed) # Move back

time.sleep(1)

trial += 1

print "Avg error: ", np.mean(err)

def point\_cloud\_registration(a,b):

'''

Purpose:

Determines the rotation and translation components of the transformation

between the two provided point cloud frames a and b using Arun's Method.

Input: Two point cloud frames a and b, each a numpy array of 3D points.

e.g. [[x1,y1,z1],[x2,y2,z2],..[xn,yn,zn]]

Output: R, the 3x3 rotation matrix, and t, the 3x1 translation matrix, such that

R\*a + t = b.

'''

N = len(a)

# find the mean [x,y,z] values for a and b

a\_centroid = np.mean(a, axis=0)

b\_centroid = np.mean(b, axis=0)

# subtract centroid values to localize points a and b

a\_local = a - np.tile(a\_centroid, (N, 1))

b\_local = b - np.tile(b\_centroid, (N, 1))

# get dot product of localized a and b

H = np.dot(np.transpose(a\_local), b\_local)

# Use Single Value Decomposition to determine the rotation matrix

U, S, V = np.linalg.svd(H)

R = np.dot(V.T, U.T)

if np.linalg.det(R) < 0:

# Special case of Arun's Method for reflection matrices

V[2, :] \*= -1

R = np.dot(V.T, U.T)

# determine the translation matrix

t = np.add(np.dot(-R, a\_centroid.T), b\_centroid.T)

# round each component to 6 decimal points

R = R.round(6)

t = t.round(6)

# return the rotation and translation matrices

return (R, t)

**PulsePlace.py:**  
  
import os, sys, time, copy

import numpy as np

import cv2

from PyQt5 import QtCore, QtGui, QtWidgets

from gui\_src import Phase1GUI\_source as GUI\_1

from kinect\_api import KinectAPI

from body\_segmentation\_api import BodySegmentationAPI

from sawyer\_api import SawyerAPI as sawyer

from win\_kinect\_api import Win8KinectAPI

from sawyer\_pykdl import sawyer\_pykdl as sk

import intera\_interface

class Phase1GUI(QtWidgets.QMainWindow, GUI\_1.Ui\_PulsePlaceGUI):

def \_\_init\_\_(self):

# Force Control Variables

self.MASTER\_STOP = False

self.max\_force = 20.0

self.min\_force = 0.0

self.med\_force = (self.max\_force + self.min\_force) / 2.0

self.SAFETY\_FORCE\_MAX = 25.0

self.desired\_force = 0.0

self.current\_force = 0.0

self.calibration\_force\_reading = 0.0

self.v = np.array([0.0, 0.0, 0.002, 0.0, 0.0, 0.0])

self.v\_max = 0.1 # [m/s] Maximum end effector velocity

self.stop = {'right\_j6': 0.0, 'right\_j5': 0.0, 'right\_j4': 0.0, 'right\_j3': 0.0,

'right\_j2': 0.0, 'right\_j1': 0.0, 'right\_j0': 0.0}

self.tolerance = 0.2 # [N] The acceptable range around desired force

self.k = 0.00001 # Proportional constant for force control

self.MOTION\_CONTROLLER\_RATE = 50 # [ms] Rate for updating the robot trajectory

self.force\_control\_active = False

# Set up GUI window

QtWidgets.QMainWindow.\_\_init\_\_(self)

GUI\_1.Ui\_PulsePlaceGUI.\_\_init\_\_(self)

self.setupUi(self)

self.CALIBRATEButton.clicked.connect(self.CALIBRATE\_BUTTON\_WRAPPER)

self.STARTButton.clicked.connect(self.START\_BUTTON\_WRAPPER)

self.STOPButton.clicked.connect(self.STOP\_BUTTON\_WRAPPER)

self.DESIRED\_FORCE\_SLIDER.valueChanged.connect(self.FORCE\_SLIDER\_WRAPPER)

self.MAXIMUM\_FORCE\_LABEL.setText(str(int(self.max\_force)) + "N")

self.MINIMUM\_FORCE\_LABEL.setText(str(int(self.min\_force)) + "N")

self.MEDIUM\_FORCE\_LABEL.setText(str(int(self.med\_force)) + "N")

# Body Segmentation Variables

self.Kinect = Win8KinectAPI.Windows8LaptopInterface()

self.Kinect\_depth\_image = None

self.cal\_img = None

self.KINECT\_UPDATE\_RATE = 1000

self.body\_segmenter = BodySegmentationAPI.SegmentSpineLocationHandler()

self.bs\_depth\_img = None

self.bs\_depth\_mat = None

self.bs\_reg\_img = None

self.target = None

# Sawyer variables

self.limb = None

self.cuff = None

self.solver = None

self.handeye\_calibration = None

self.usprobe\_calibration = None

def CALIBRATE\_BUTTON\_WRAPPER(self):

"""

Purpose: Collects a kinectv2 depth image (where the patient is not present) to be used for human

shape segmentation. When pressed it will also recalibrate the torque sensor readings

"""

def CALIBRATE\_BUTTON\_FUNCTION(self):

self.UPDATE\_SAWYER\_STATUS\_LABEL('neutral')

sawyer.activate\_robot()

self.cuff = intera\_interface.Cuff('right')

self.limb = intera\_interface.Limb('right')

sawyer.tf\_init()

self.solver = sk.sawyer\_kinematics('right')

self.cal\_img = cv2.imread('calib.bmp', 0)

for i in range(0,100):

self.calibration\_force\_reading += self.limb.endpoint\_effort()['force'][2]

self.calibration\_force\_reading /= 100

print self.calibration\_force\_reading

return

CALIBRATE\_BUTTON\_FUNCTION(self)

def START\_BUTTON\_WRAPPER(self):

"""

Purpose: Segments the human shape from the kinectv2 depth image, extracts a point on the body to move to

Recalibrates the torque sensor, Moves the end effector 10cm above the target location,

Starts velocity/force controlled descent onto the patient.

"""

def START\_BUTTON\_FUNCTION(self):

self.bs\_depth\_img = self.Kinect.get\_depth\_image()

self.Kinect.get\_joint\_data()

self.kj = Win8KinectAPI.KinectJointLocations()

if self.bs\_depth\_img is not None:

self.target = self.body\_segmenter.run(self.bs\_depth\_img, self.cal\_img, self.kj)

self.target\_rc = self.handeye\_calibration \* self.target

self.target\_rc[2] += 0.010

self.p, self.q = sawyer.get\_tf()

if not self.MASTER\_STOP:

sawyer.move\_to(self.target, self.q, speed=0.05)

for i in range(0,100):

self.calibration\_force\_reading += self.limb.endpoint\_effort()['force'][2]

self.calibration\_force\_reading /= 100

print self.calibration\_force\_reading

self.MOTION\_CONTROLLER\_TIMER = QtCore.QTimer()

self.MOTION\_CONTROLLER\_TIMER.timeout.connect(self.MOTION\_CONTROLLER)

self.MOTION\_CONTROLLER\_TIMER.start(self.MOTION\_CONTROLLER\_RATE)

self.force\_control\_active = True

return

START\_BUTTON\_FUNCTION(self)

def STOP\_BUTTON\_WRAPPER(self):

"""

Purpose: Allow user to stop the robot and make it move back off the patient

"""

def STOP\_BUTTON\_FUNCTION(self):

self.p, self.q = sawyer.get\_tf()

self.p = np.array(self.p[0], self.p[1], self.p[2] + 0.10)

if not self.MASTER\_STOP:

sawyer.move\_to(self.target, self.q, speed=0.05)

self.MASTER\_STOP = True

return

STOP\_BUTTON\_FUNCTION(self)

def FORCE\_SLIDER\_WRAPPER(self):

"""

Purpose: Allow user to select desired force on GUI

"""

def FORCE\_SLIDER\_FUNCTION(self):

self.desired\_force = ((self.DESIRED\_FORCE\_SLIDER.value() + 1) / 100.0) \* self.max\_force

self.SELECTED\_FORCE\_LABEL.setText(str(self.desired\_force) + "N")

FORCE\_SLIDER\_FUNCTION(self)

def CHANGE\_IMAGE\_DISPLAY\_BGR(self, img):

img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

height, width, channel = img.shape

bytesPerLine = 3 \* width

qimg = QtGui.QImage(img.data, width, height, bytesPerLine, QtGui.QImage.Format\_RGB888)

self.IMAGE\_DISPLAY.setPixmap(QtGui.QPixmap(qimg))

def CHANGE\_IMAGE\_DISPLAY\_GRAY(self, img):

height, width = img.shape

bytesPerLine = width

qimg = QtGui.QImage(img.data, width, height, bytesPerLine, QtGui.QImage.Format\_Indexed8)

self.IMAGE\_DISPLAY.setPixmap(QtGui.QPixmap(qimg))

def UPDATE\_SAWYER\_STATUS\_LABEL(self, status):

"""

Purpose: Update the robot status light on the GUI

"""

if status == 'neutral':

self.STATUS\_LIGHT.setPixmap(QtGui.QPixmap(os.getcwd() + "/gui\_src/gui\_imgs/greenb.png"))

elif status == 'moving':

self.STATUS\_LIGHT.setPixmap(QtGui.QPixmap(os.getcwd() + "/gui\_src/gui\_imgs/yellowb.png"))

elif status == 'stopped':

self.STATUS\_LIGHT.setPixmap(QtGui.QPixmap(os.getcwd() + "/gui\_src/gui\_imgs/redb.png"))

elif status == 'off':

self.STATUS\_LIGHT.setPixmap(QtGui.QPixmap(os.getcwd() + "/gui\_src/gui\_imgs/grayb.png"))

def KINECT\_IMAGE\_UPDATER(self):

self.Kinect\_depth\_image = self.Kinect.get\_depth\_image()

if self.Kinect\_depth\_image is not None:

self.CHANGE\_IMAGE\_DISPLAY\_GRAY(self.Kinect\_depth\_image)

def UPDATE\_CURRENT\_FORCE(self, force\_value):

"""

Purpose: Updates the Force Reading on the GUI and also updates the internal current\_force variable

:param force\_value: Force in Newtons. type = float

"""

self.current\_force = force\_value

if force\_value >= 0:

update\_string = " " + "{:.4f}".format(force\_value) + "N"

else:

update\_string = "{:.4f}".format(force\_value) + "N"

self.CURRENT\_FORCE\_VALUE\_LABEL.setText(update\_string)

def FORCE\_UPDATER(self):

"""

Purpose: Updates the current force reading. If no force reading yet obtained, get the

force reading of just the ultrasound probe for calibration. The force is updated at a

rate specified by self.FORCE\_UPDATE\_RATE

"""

if self.initial\_force\_reading is 0:

self.initial\_force\_reading = self.limb.endpoint\_effort()['force'][2]

self.UPDATE\_CURRENT\_FORCE(self.limb.endpoint\_effort()['force'][2] - self.calibration\_force\_reading)

self.update()

def MOTION\_CONTROLLER(self):

"""

Purpose: This is the primary motion control function. It is called at a rate specified by

self.MOTION\_CONTROLLER\_RATE

At each call several checks are performed before motion occurs

1) self.MASTER\_STOP is checked. This check will fail if the user pressed the 'Stop' button on

the GUI, the limb reports a collision, the force sensor reads a value over self.FORCE\_SAFETY\_MAX

or the requested robot end effector velocity self.v becomes greater than self.v\_max

2) A check is performed to see if the large cuff button is squeezed. If it is squeezed, force control is

deactivated (by setting self.force\_control\_active to False)

3) A check is performed to see if the white cuff button is pressed. If it is pressed, force control is

activated (by setting self.force\_control\_active to True)

4) A check is performed to see if the robot has reported a collision. If it has, force control will be

deactivated by setting self.MASTER\_STOP to True. The function call then returns

5) A check is performed to see if self.force\_control\_active is True and self.MASTER\_STOP is False. If

the function passes these checks, the desired end effector velocity is computed proportional to the

difference between the current force and the desired force. If a valid solution is found, the robot

will be commanded with the computed joint velocities.

NOTE: Sawyer's typical safety checks are disabled when commanding it with 'set\_joint\_velocities'

This is why so many checks are performed to ensure the robot does not push too hard or move to fast

and cause injury.

"""

if not self.MASTER\_STOP:

if self.limb.has\_collided():

print "Collision detected!"

self.force\_control\_active = False

self.limb.set\_joint\_velocities(self.stop)

self.UPDATE\_SAWYER\_STATUS\_LABEL('stopped')

self.MASTER\_STOP = True

return

if self.force\_control\_active:

if not self.MASTER\_STOP:

e = self.desired\_force - self.current\_force

print "End Effector Force: ", self.current\_force, self.k \* e

# Check to see if error is small enough

if self.current\_force < 2.0:

self.v[2] = self.v[2]

elif math.fabs(e) > self.tolerance:

# error is not small, so we keep moving

self.v[2] += self.k \* e

elif math.fabs(e) <= self.tolerance:

# error is small, so we stop

self.v[2] = 0.0

self.limb.set\_joint\_velocities(self.stop)

self.MOTION\_CONTROLLER\_TIMER.stop()

# Change GUI to phase 2, either change the labels or call 2nd process and kill this one

if self.current\_force > self.SAFETY\_FORCE\_MAX or self.v[2] < -self.v\_max:

# If we fail safety checks, stop the robot and disable force control

print "Safety stop!"

self.v[2] = 0

self.limb.set\_joint\_velocities(self.stop)

self.MASTER\_STOP = True

self.force\_control\_active = False

self.UPDATE\_SAWYER\_STATUS\_LABEL('stopped')

return

theta = self.velocity\_ik(self.v)

self.limb.set\_joint\_velocities(theta)

self.update()

def velocity\_ik(self, v):

"""

Purpose: Given a 3x1 velocity vector, compute the joint velocities to realize end effector velocity v

:param v: 3x1 velocity vector. type = numpy array

:return: theta: Dict of joint velocities matched to the proper Sawyer joints

"""

q, p = sawyer.get\_tf()

R = sawyer.quat2rot(q)

end\_eff\_v = self.limb.endpoint\_velocity()

v\_ = R.dot(v[:3])

v = np.array([v\_[0], v\_[1], v\_[2], v[3], v[4], v[5]])

twist = self.vel2kdltwist(v\_)

theta = self.solver.inverse\_velocity\_kinematics(twist)

theta = dict(zip(self.limb.joint\_names(), np.array(theta).tolist()))

return theta

def vel2kdltwist(self, velocity):

"""

Purpose: Converts a 6x1 complete velocity vector to a PyKDL Twist() object for use in inverse kinematics

:param velocity: 6x1 velocity vector (cartesian x,y,z and rotational x,y,z)

:return: V: Velocity Twist type = PyKDL Twist() object

"""

V = PyKDL.Twist()

for i in range(len(velocity)):

V[i] = velocity[i]

return V

if \_\_name\_\_ == "\_\_main\_\_":

app = QtWidgets.QApplication(sys.argv)

Phase1GUI\_window = Phase1GUI()

Phase1GUI\_window.show()

sys.exit(app.exec\_())

**RobotController.py:**  
import os

import pickle

import datetime

import numpy as np

from socket import \*

import time

import SawyerInteraction as sawyer

class NeedleFollower:

def \_\_init\_\_(self):

'''

Purpose:

Robot-side script for automatic tracking of a needle tip with a robot

It acts as a server to the Alpinion Ultrasound scanner, waiting for new UDP packets to be sent

which either contain coordinates for where the robot needs to move or a message informing the

robot controller that the needle tip has not be found, as well as how long it has been since the needle

tip was last found.

'''

self.log\_file = open('log.txt', 'w')

sawyer.activate\_robot()

# Load up ultrasound probe calibration data

self.r\_cal, self.t\_cal = pickle.load(open('calibration\_data', 'rb'))

self.t\_cal /= 1000.0

self.cal\_frame = sawyer.build\_frame(self.r\_cal, self.t\_cal)

self.probe\_coordinates = np.array([0.0, 0.0, 0.0])

self.nt\_coordinates = np.array([0.0, 0.0, 0.0])

# Initialize the frame transform streaming object

sawyer.tf\_init()

# Set up UDP connection to ultrasound scanner

self.host = ""

self.port = 13000

self.buf = 1024

self.addr = (self.host, self.port)

self.UDPSock = socket(AF\_INET, SOCK\_DGRAM)

self.UDPSock.bind(self.addr)

# Some misc control variables

self.forward = True

self.f\_count = 0

self.b\_count = 0

self.k0 = 3000000

self.k1 = 3

self.not\_found\_count = 0

def run(self):

print "Ready to recieve data..."

while True:

# Get the most recent data from the scanner

(self.data, self.addr) = self.UDPSock.recvfrom(self.buf)

self.buf = 0

self.buf = 1024

print "Received a message: ", self.data, self.data[0:3]

self.probe\_coordinates = self.get\_probe\_location()

if self.data[0:3] == "msg":

# We have received a control message

self.process\_control\_msg()

elif self.data[0:3] == "val":

# We have received some sort of data...

self.nt\_coordinates = self.process\_data\_msg()

self.log\_file.write("probe: " + str(self.probe\_coordinates)+",\n"+ "needle tip: " + str(self.nt\_coordinates)+"\n")

def get\_probe\_location(self):

homog\_t\_cal = np.matrix([self.t\_cal[0], self.t\_cal[1], self.t\_cal[2], 1])

q, p = sawyer.get\_tf()

robo\_frame\_matrix = sawyer.build\_frame(sawyer.quat2rot(q), p)

probe\_coordinates = robo\_frame\_matrix \* homog\_t\_cal.T

return probe\_coordinates

def parse\_coord(self, str\_coord):

'''

Purpose:

Parses a string of numeric elements e.g. "[156.0, 255.7, 356.2]" and converts it into a list

e.g. [156.0, 255.7, 356.2]

This is needed because the coordinates of the needle tip are transferred as a string over the UDP connection.

Inputs:

str\_coord -> a string version of a list of numbers

Outputs:

parsed\_list -> a list of numbers, str\_coord converted to a list

'''

tup\_coord = tuple(str\_coord)

#print "tup: ", tup\_coord

num\_str = ""

parsed\_list = []

new\_num\_flag = False

for i in tup\_coord:

#print "parsed\_list: ", parsed\_list

if not new\_num\_flag and i in ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']:

new\_num\_flag = True

if i not in ['[', ',', ' ', ']']:

num\_str += i

if i is ',' or i is ' ' and new\_num\_flag:

#print "num\_str: ", num\_str

parsed\_list.append(float(num\_str))

num\_str = ""

new\_num\_flag = False

if i is ']' and not num\_str == "":

print "num\_str: ", num\_str

parsed\_list.append(float(num\_str))

return parsed\_list

return parsed\_list

def process\_data\_msg(self):

"""

Based on the data coordinates sent, move the robot end effector to center

the US probe over the needle tip coordinates

"""

self.f\_count = 0

self.b\_count = 0

coordinates = self.parse\_coord(self.data[4:])

print "Data acquired: ", str(coordinates)

homog\_coord = np.matrix([coordinates[0], 0.0, 0.0, 1.0])

homog\_t\_cal = np.matrix([self.t\_cal[0], self.t\_cal[1], self.t\_cal[2], 1])

q, p = sawyer.get\_tf()

robo\_frame\_matrix = sawyer.build\_frame(sawyer.quat2rot(q), p)

v1 = robo\_frame\_matrix \* self.cal\_frame \* homog\_coord.T

v1 = np.array((float(v1[0][0]), float(v1[1][0]), float(v1[2][0])))

v2 = robo\_frame\_matrix \* homog\_t\_cal.T

v2 = np.array((float(v2[0][0]), float(v2[1][0]), float(v2[2][0])))

v3 = v2 - v1

print 'v3: ', v3, np.linalg.norm(v3)

desired\_position = np.array(p) - v3.T

print ' dest: ', desired\_position, p

sawyer.move\_to(list(desired\_position), q, speed=0.025)

print "error: ", (desired\_position - sawyer.get\_tf()[1]), " " + str(datetime.datetime.utcnow())

# Return the robot-space coordinates of needle tip

homog\_coord = np.matrix([coordinates[0], coordinates[1], 0.0, 1.0])

nt\_coordinates = robo\_frame\_matrix \* self.cal\_frame \* homog\_coord.T

return nt\_coordinates

def process\_control\_msg(self):

"""

Either shut down, initiate back and forth searching, or do nothing based on

the content of the control message

"""

print self.not\_found\_count

print self.data[4:13]

if self.data[4:] == "shut\_down":

print "Shutting down!"

self.log\_file.close()

exit()

elif self.data[4:13] == "not\_found":

self.not\_found\_count = int(self.data[13:])

print "No needle tip found in the last ", self.not\_found\_count, " cycles."

if self.not\_found\_count > self.k0:

# Do search routine

print "Search routine active..."

print (not self.f\_count and self.b\_count < 2 \* self.k1)

q, p = sawyer.get\_tf()

#print p, q

if self.forward and self.f\_count < self.k1:

self.f\_count += 1

v1 = np.array((0.0, 0.005, 0.0))

R = sawyer.quat2rot(q)

v2 = R.dot(v1)

print "move1"

sawyer.move\_to(p + v2, q, speed=0.025)

elif self.f\_count >= self.k1 and self.b\_count < 2 \* self.k1:

self.b\_count += 1

v1 = np.array((0.0, -0.005, 0.0))

R = sawyer.quat2rot(q)

v2 = R.dot(v1)

print "move2"

sawyer.move\_to(p + v2, q, speed=0.025)

else:

return

return

if \_\_name\_\_ == '\_\_main\_\_':

n = NeedleFollower()

n.run()