The STANLEy User Manual

Stabilization and Transducer Alignment for Nearby Laser Elastography

Ben Anderson

November 28, 2022

Contents

1 Introduction

The focus of my work for the better part of the last year has been studying how to use a UR3e Cobot for the purposes of focusing and stabilizing a sensor at its focal length away from a target surface. This has involved a thorough examination of the API and programming tools that accompany the Universal Robots ecosystem, as well as creating systems for synchronizing the robotic control system with signal processing and sensor systems.

In doing so a number of systems are being used in ways for which they were not designed. The consequence of this is that my codebase resembles a kludgy pile of half-fixes and temporary-solutions that, technically, have achieved the objective, at the expense of stability and ease of use. In recent months more efforts have been made to improve the readability of the codebase and allow for easier changes later on, but the work is ongoing and the project cannot wait for me to complete this work.

The hope is with this documentation, others in the OCE project can gain enough of an understanding of how the system works to use it in my absence as well as modify or extend the platform I've built.

Due to the kludgy nature of the platform, it may not be suitable to read this guide linearly. The introduction section should hopefully give you an overview of how the components of this section communicate and interface with each other, and a more thorough description of the workings of each module are in the following chapters.

If your only goal is to get started running the robot, strictly speaking you need only read the startup section (section 2), though I recommend also reading subsection 4.1, which covers use of the teach pendant.

If you need to

2 Startup guide

Pending the rest of the summary, the following procedure should be all you need to start running this control system on your machine.

2.1 First installation

These steps only need to be ran once when initially setting up your Cobot, and don't need to be repeated on subsequent use.

1. Install all software dependencies on the computer.
    - Python 3.8+ on target machine
    - Packages required for all of the python code can be installed by running
      `pip install -r requirements.txt`
      from the main directory of this repository
    - MATLAB/LabView/whatever other signal processing code you need for dealing with sensor input
2. Connect the UR3e robot to your machine via an ethernet cable.
3. Assign the ethernet connection to your robot a static IP address with the following criteria:
    - **IPv4**
    - **IP address:** `192.168.0.5` (Corresponds to addresses hard-coded into the running programs)
    - **Subnet prefix length:** `255.255.255.0`

- **Gateway:** `192.168.0.1`
- **Preferred DNS:** `10.1.2.1`
- **Alternate DNS:** `8.8.8.8` (Less critical, any valid DNS will do)

4. Do the same on the robot, instead assigning the following values:
    - **IPv4**
    - **IP address:** `192.168.0.10`
    - **Subnet prefix length:** `255.255.255.0`
    - **Gateway:** `192.168.0.1`
    - Set the DNS settings to any valid DNS addresses.
    - Details on modifying network settings an be found on page 108 of the Polyscope manual
5. Write/save/open a program on the UR3e that does the following:

```
BeforeStart:
    script(before_start.script)
Robot Program:
    script(robot_program.script)
```

    - The corresponding scripts are found in the `\Code_for_the_robot` folder in this repository.
    - For more information on node programming, check out the chapter on programming in the Poylscope manual
6. Make sure the robot has the correct installation selected. This includes making sure the dimensions of the tool match the current setup of the sensor on the end effector of the robot. For more information on configuring the tool, read the section on TCP configuration in the Polyscole manual
    - This will require some knowledge of the shape/orientation of the sensor you are currently working with.
    - When working with sensors that have a focal point offset from the end of the tool, the TCP offset should be defined at the location of the focal point, not the edge of the tool.

This setup will only need to be run when a part of your installation changes. Steps one through three need to be repeated whenever you run the installation on a new computer; step four only needs to be run once for each robot (or whenever the robot goes through a hard-reset). Steps five and six shouldn't *need* to be repeated regularly, but sometimes need to be rerun if the robot isn't shut down correctly. (This is very easy to do).

2.2 Starting up the robot

Once the above configuration steps have be completed, you should be ready to run experiments with the robot after following this procedure:

1. Power on the robot by following the boot-up procedure described in subsubsection 4.1.1 (also power on your computer, obviously)

Warning: The robot will move slightly when it boots up as it unlocks its brakes. Keep sensitive equipment clear from the robot, and do not power off the robot in a position that will put nearby objects at risk when it is turned on.

2. (Optional) Use the teach pendant to manuever the robot into a position convenient for the following test. This is optional as the control loop allows you to toggle freedirve mode, but it may be more convenient to do so here.

3. Make sure the correcet program and installation are selected on the teach pendant; To read more about navigating the program/installation menus, read the section on this subject in the Polyscope Manual

4. Do any setup necessary for your sensor now; the robot can be left idle while you do so.

5. Once your setup is ready, open the `COBOT_Transducer_Control_Code` directory on your PC, and double-click `server.py`

   - A blank terminal window should pop up with a blinking cursor; (You do not need to interact with the terminal window, it is only there for status messages)
   - This activates the socket server that the system uses to communicate between modules.

6. Tap the 'play' button at the bottom of the teach pendant to run the program; The terminal window from the previous step shold show a new connection has been established.

7. Start your signal processing script; make sure your signal processing program follows the appropriate data transmission standard (Described in Figure 2) (TODO elaborate)

8. Finally, run `TransducerHoming.py`. This can be done by double-clicking the file in file explorer, but it is better to run it from an IDE such as Visual Studio; the software is not stable and being able to see an error message when it breaks is very helpful.

   - This should open a terminal window that looks like the following:

```
TCP position in relation to its initial position:
        ([[-1000.00 -1000.00 -1000.00]](mm),[[-57.30 -57.30 -57.30]](deg))
========================
TCP position in base: ([[ 0.00  0.00  0.00]], [[ 0.00  0.00  0.00]]
Current joint position in degrees: ([[ 0.00  0.00  0.00  0.00  0.00  0.00]])
Recent refresh rate: 0.1465371191501617415
------------------------
Latest mag:48
|||||                                                                    |
Freedrive active: False
Press (f) to toggle (f)reedrive mode :)

Press (d)emo to demonstrate the basic pathrouting module
Press (k) to trigger a full scan with hard-coded resolution.
Press (g) to trigger a amplitude max-finding pathrouter.
Press (t) to trigger an alternate amplitude max-finding pathrouter.
Press (q) to quit
```

```
Sun Nov 20 18:23:36 2022
----------------
```

From here you are ready to run your test. Follow the directions on screen to enable/disable freedrive and activate the appropriate pathfinding module. The two menu options I suggest using are '(k) for fullscan' and '(g) for maxfinding'. The fullscan module runs a full scan of a search volume of a hard-coded shape and resolution, and the maxfinding module runs the current-best module for focusing a transducer.

2.3 Common debugging steps

Because the sole author of 90% of this code is such a genius, occasionally this software breaks in ways that may be frustrating. Here are common problems and the best-known steps to address them (short of writing better sofware)

- The robot collides with something during a test and locks itself up
    1. If this happens the experiment is ruined, you will need to restart
    2. Close the `TransducerHoming.py` script, and stop the socket server.
    3. The robot will need to be unlocked through the main menu, after you have done so tap the 'stop' button to cancel the running program (the default behavior is to pause the program, you don't want that)
- `TransducerHoming.py` crashes
    - This is a frequent occurance whenever major changes are made to the program or installation
    - It is worth noting though, that `TransducerHoming.py` is not a dependency for the other components; they will simply sto pmoving. You are able to restart the script without resetting any of the other systems.
    - If you ran the script through an IDE, take note of the error message and where it occured, and alert Ben Anderson to the problem (bpa13@uw.edu) or submit a pull request on github

3 System Overview

The system is designed to keep as much modularity as possible and to isolate responsibilities for particular tasks.

The sensor is a generic device that requires focusing and produces an output that is read by the signal processing package within the computer. The Stabilization and Transducer Alignment for Nearby Laser Elastography system (hencforth referred to as STANLEy) consists of a UR3e cobot from Unviersal Robots, and a script written in using the Universal Robots proprietary scripting language. (Attempts are being made to reverse-engineer some of this languages functions to cut this box out of the control diagram, but the process will be very involved and is being put off in order to meet deadlines).

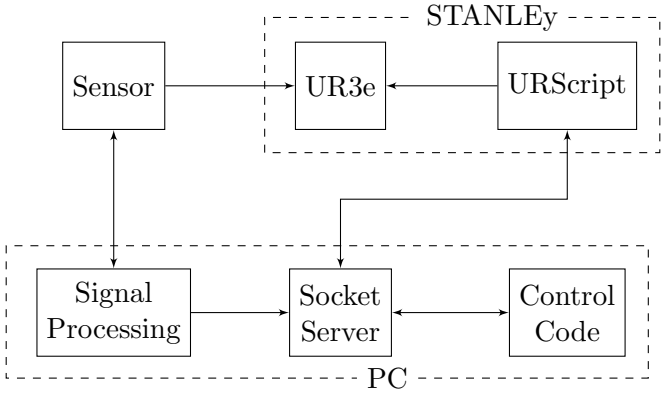


Figure 1: Component overview of the STANLEy system

Physically, this system includes a robotic arm manipulator, a control box, and a teaching pendant. STANLEy interfaces with the sensor through a physical linkage at the end of the robotic manipulator, and with the PC through an ethernet cable.

The remaining control blocks occur within a nearby computer. The signal processing block refers to a program (typically in MATLAB or LabView) which recieves the data from the sensor and processes it for further use.

For the purposes of the control loop that focuses the sensor, the signal processing block must reduce the sensor input, no matter how complex, to a simple magnitude that should be maximized to bring the sensor into alignment. More on this to be added later.

An internal socket server is spun up in a simple python program. This is a program that listens on a hard-coded IP-address and port within the computer for input sent from the signal processing block, the control code, and to the robot through the control box. More on the specific IP addresses in the initial setup section of the startup section. (section 2)

The control code is run from a script named `TransducerHoming.py`, and associated libraries. For now, directly editing this file is the main interface with which we can modify how the homing sequence runs. In future I hope to expand the user interface as well as the command-line options to allow more on-the-fly modification of program parameters, but the duration off tests and the speed at which changes are made have made this an inefficient goal for some time.

Overall the system utilizes three discrete devices and at least three programming languages. I (Ben Anderson) am most directly involved with the programming of the control code and URScript. The top-down overview of each components' functions are pretty simple, but debugging/modifying them will require a rough understanding of the URScript API and a strong understanding of Python.

4 STANLEy and the URScript API

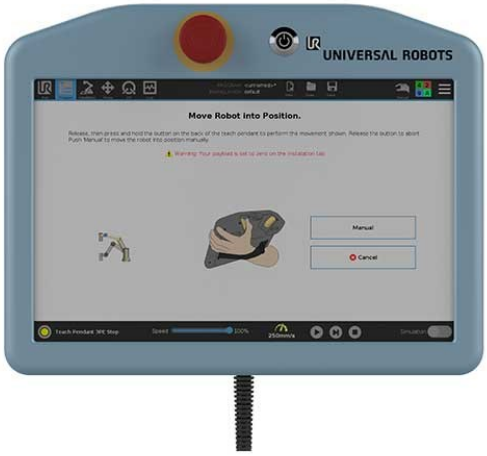
The body of the novelty surrounding this project centers around the use of a UR3e robotic arm to position and hold the OCE sensor.

The UR3e is manufactured by Universal Robots and is primarily designed to automate human labor. Shipped with the robotic arm is a control box and a teach pendant, a hand-held tablet for manually controlling the robot. The control box allows one to write and execute scripts using the proprietary URScript API, a set of functions built into the robot that handle common tasks related to positioning and animating the arm.

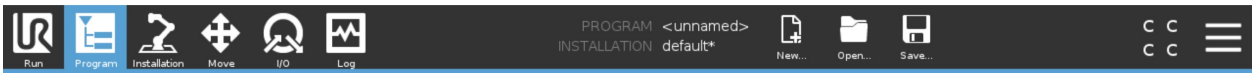
Additionally, the control box has an ethernet port supporting the secondary and Real Time Data Exchange interfaces. Both of these have their uses and can be read about in the remote control interface guide, however they weren't adequate for the method ultimately used.

4.1 The Teach Pendant and the PolyScope interface

The teach pendant is the default accesory for interacting with the internals of the robot. It is shipped with the robot along with the control box.



The pendant, when active, is running an interface known as PolyScope, the GUI used for operation of all UR robots. Along the top of the page, there are six different menus and a file explorer interface that we will refer to later:



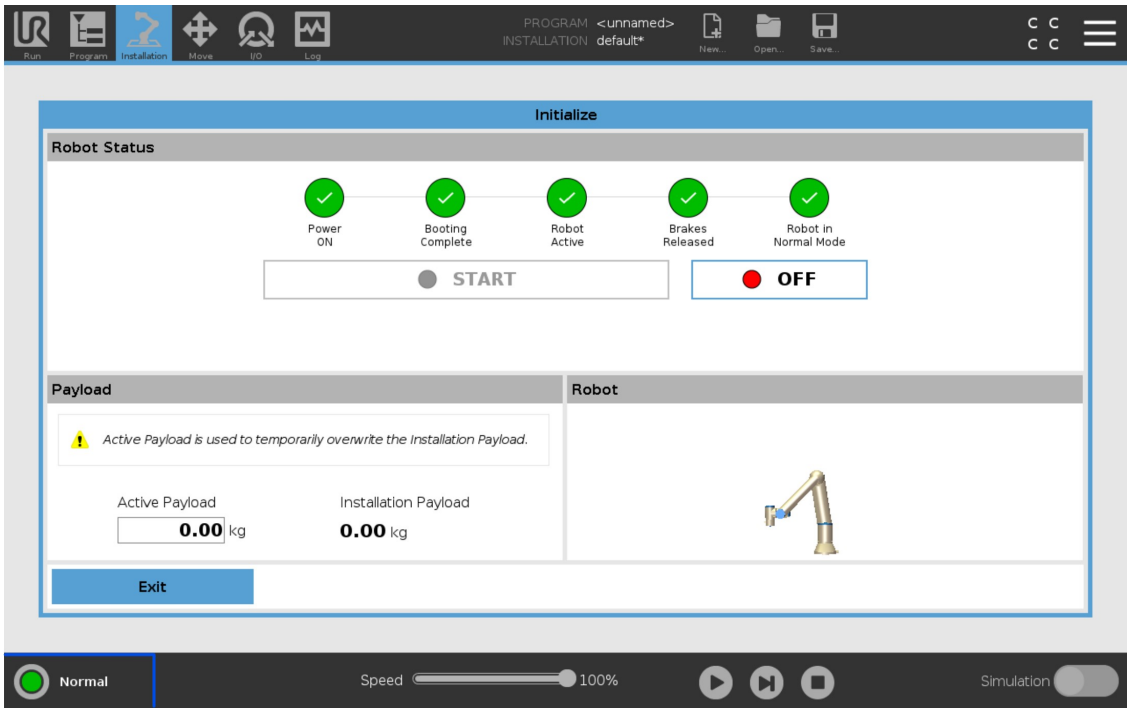
The full manual for PolyScope can be read here, but this guide will cover the elements necessary for operation of STANLEy.

4.1.1 Initialization and setup

In order to use the robot, power on the tool box by pressing the physical power button above the screen and next to the e-stop on the teach pendant. (It can be finicky how long a press is necessary, so if you suspect you're holding the button down too long listen for a click from the control box to indicate it has been powered on. You should release the button immediately after hearing this click).

The power button should light up green and the screen should show a loading screen as the robot takes a minute to boot up.

Upon fully booting up, the robot itself still remains dormant. In the bottom-left of the screen is the robot status indicator, shown here:



The color inside the circle indicates the state of the robot:

- *Red*: The robot is powered off.
- *Blue*: The robot is in freedrive mode.
- *Yellow*: The robot is idle; powered on but not ready for operation.
- *Green*: Powered on and ready for normal operation.

After booting up the control box you enter the Initialization screen (shown in the image above). When powering up the robot from full-shutdown the central button will be active and you will press it twice to enable the robot; first it will say 'ON' and you press it to deliver power to the robot, then it will say 'START' and you will press it to unlock the brakes and enable the robot completely.

Warning: The robot will move slightly as the brakes are unlocked, do not leave the robot in a stat where this motion will put nearby equipment at risk.

4.1.2 Program tab

The program tab is one of the options in the menu tab. Here is where you write/modify programs for controling the robot. Polyscope has a node-based programming langauge similar to scratch, in addition to a powerful scripting language that can be baked into a robot program. When you press the 'play' button at the bottom of the tablet, the program it runs will be in this form (if one is loaded).

Stanely (The UR3e currently used in our lab) has a program saved called `test.urp` that works with our control code. It can be opened by tapping the 'open' button in the top banner and tapping 'Program' from the subsequent dropdown. Our program relies heavily on the scripting language, so the only nodes in our program are a 'BeforeStart' and 'Run Program' node, each containing a script that does everything we need.

> **Note**: If you want to edit either of these programs, *use a USB stick and a computer to transfer/edit the program.* The text input for the teach pendant is a finicky touch screen and I've had trouble getting a keyboard with the right language settings.

4.2 URScript API

URScript is the propreitary scripting language created by Universal Robots and runs natively on all of their machines. The functions are tailored to work for their machines specifically, and I haven't found a way to run URScript outside of the control box. The full URScript API guide can be found here (a copy of this document can also be found in Documentation folder of this git directory).

The URScript API provides one major opportunity and one major problem; On the one hand, it abstracts-away a lot of the low-level handling that's required to move the robot in a controlled manner. At no point in this project did we need to work with inverse kinematics or joint positions or anything of that sort.

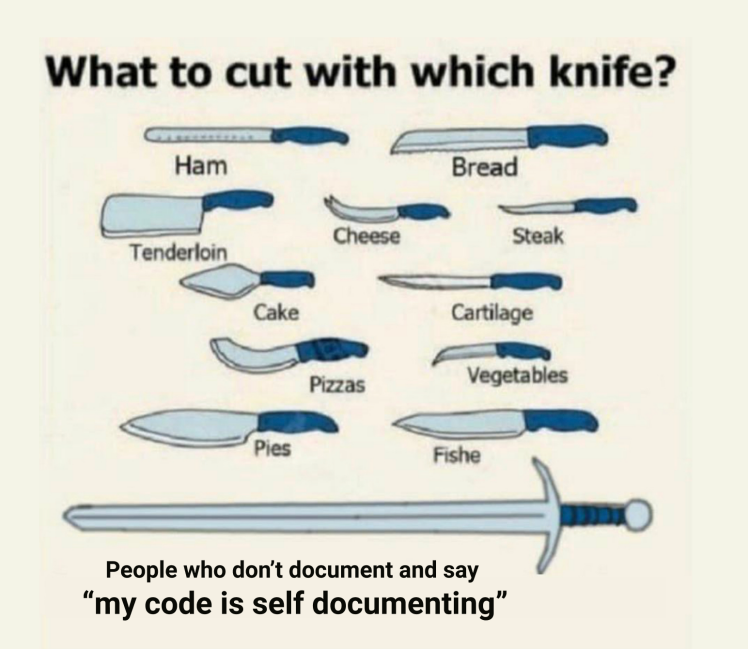
The tradeoff is that the URScript API is extremely confining and doesn't provide a lot of ways to interface with other computers. The UR series of robots were designed to work collaboratively with humans, automating tasks that would normally be done with manual labor. Consequently, most of the scripting language and interfaces are designed with repetitive, simple tasks in mind, and getting the robot to work in an active control setting was a challenge.

Essentially the two ways they design the robot to work is either as a standalone machine, running URScript program nodes that get written once and then repeated forever, or as a mindless drone recieving simple commands from an external computer.

There is an interface in the robot to allow for direct control from an external computer, but the available commands in this mode are limiting.

5 Control code

Working on it



5.1 `TransducerHoming.py`

476 lines of nonsense and yes I wrote the whole thing myself. No I am not fine. More extensive documentation is pending but for now we need only refer to the data formatting standard used to pass data between the sensor and the control code: subsubsection 5.1.1

STANLEy - Stabilization and Transducer Alignment for Nearby Laser Elastography

5.1.1 Interfacing with Sensor output

The control code has been written to move the robot in such a way that it maximizes some value it is getting passed from the outside. It is completely agnostic about where this value comes from, and it is the programmers responsibility to ensure that the value it gets passed is an appropriate function of how 'focused' the sensor you're working with is.

Our goal is to create a fitness function f that produces a single scalar value representing how well focused the sensor is. A fitness function f will result in the robot focusing correctly if:

- The global maximum value of f occurs at the focal point of the sensor
- f has few or no local maxima

Other desireable properties of the fitness function are:

- Function f is concave (e.g. $\nabla^2 f \leq 0$ for everywhere in the domain)

- f has a discrenable magnitude for a wide envelope around the focal point (since eventually the function will be dominated by noise)
- f has a high SNR across as much area as possible
- f is smooth everywhere
  - If f is smooth everywhere except at the focal point $f_{\max}$, that is also acceptable

The robot control code sees this value as some function of the tools position in 6-D space, and will try to maximize that function according to the constraints you give it.

Once that fitness function has been defined, the signal-processing toolkit must send that value to the central socket server as a string in the following format:

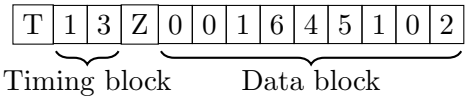


Figure 2: Visual description of the data format in messages between the sensor processing block and the socket server.

The two characters following the T is a timing block that increments by one each time a new value is sent to the server (this is so the code knows when a value it recieves is new). The nine characters following the Z is the data block, and this is where the value of the fitness function must go. Nine digits is an arbitrary choice of precision, but should be sufficient for the kind of resolution we deal with.

> **Note:** The socket server can only handle integers, but high-resolution data can still be provided by multiplying a value by some large number before passing a value in.

For example, my previous setup yielded fitness values in a range between 40 and 700, so to capture full definition I multiplied the value by 1000 before passing it and divided it again on the other side.

6 Testing methodology

The control logic we write for focusing the transducer needs to be resilient to noise and different initial placements of the sensor. Therefore it is critical that each mode of control logic gets tested thoroughly before being relied on in produciton. There are logistical limitations with testing control logic exclusively on the physical system, and if a particular algorithm fails to focus the transducer correctly, it's not always possible to understand why.

To deal with this, control patterns were tested against pre-recorded data so their feasability could be checked quickly, before the simulated results were validated with the real robot.

The `FullScan` module in `Pathfinders.py` defines a pathfinder that visits every point in an n-dimensional search space at a given resolution. Such resolution should be adjsted to reflect the size of the focal area of the sensor.

Shown below is the data in `test_6.json`, which captured over 51749 signal magnitudes. It scanned a search space that spanned $\pm 13°$ in the θ_x, θ_y directions, and $\pm 14mm$ in the z direction, at a resolution of $0.4mm$ along translational axes and $1.0°$ in the rotational axes.

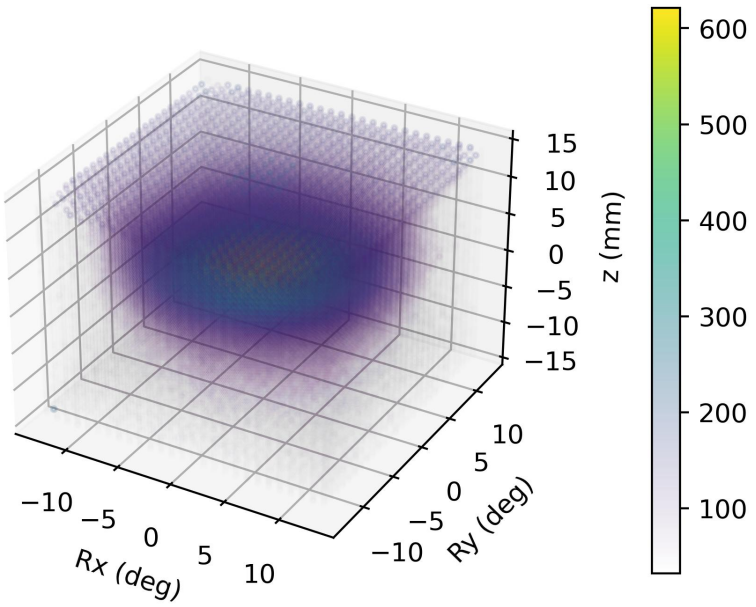


Figure 3: Raw data output from a full-scan (opacity is scaled with data magnitude, so many values are hidden)

There are two helpful things that the data from a full scan gets used for; first of all it allows us to test our pathfinders explicitly on pre-existing data, and secondly it allows us to evaluate how our fitness function plays out in a real scan.

6.1 Sensor fitness function

6.1.1 A foreword in fitness function design

Earlier the fitness function was described as a function of the position of the robot that outputs a scalar representing how well the sensor is focused, and that the robot control code sees this as a function in 6-D space. Now we will get into the process of designing and evalutating effective fitness functions.

Since we are trying to bring a sensor into focus, we assume that the 'focusedness' of the transducer is some function of its position in 6-D space relative to the target. Since, fundamentally, the focusedness of a sensor relates to some physical phenomena, we can imagine a function $\delta(\mathbf{X})$ that represents the true function mapping sensor-target position $\mathbf{X} = \{x, y, z, \theta_x, \theta_y, \theta_z\}$ to sensor focus. This function should relate to the fundamental principles of the sensing method and the quantity being measured.

In the real world our sensor will never be able to take perfect readings because we cannot isolate the quantity being measured from the other elements of the system. Therefore we must imagine that the data collected from our sensor is some mixture of the ground truth function and some stochastic noise.

$$\mathcal{D} = \delta(\mathbf{X}) + \epsilon \qquad \epsilon \sim \mathcal{N}(\mu, \sigma^2)$$

The error term might change depending on the setup, the sample, or how the signal is processed.

When we start focusing the robot, we have no idea where the robot is relative to the target surface, we can only react to input data the sensor provides. So our fitness function f from earlier is some function of the sensor data:

$$f(\mathcal{D})$$

A healthy place to start might be to approximate the ground truth function $\delta(\mathbf{X})$, but that isn't always the case. In the next section I will give an example of a naive fitness function that has inconvenient properties. This function is analogous to a loss function when solving an optimization problem.

6.2 Evaluating the fitness function on real data

The fitness function ultimately exists to help our robot focus effectively; therefore the most direct way of measuring whether the fitness function is effective is to run a focusing routine on the robot and see how close the sensor gets to being perfectly in focus. The problem with this approach is that it's not effective at diagnosing problems; if the robot focuses effectively that's great, but if it doesn't then it's really difficult to reverse-engineer what went wrong.

The more maleable approach instead is to run a high-density scan of a search space that surrounds the focal point of the sensor, and record the data for later analysis. The time cost of running a high-density scan on the robot is significantly higher than doing an individual focusing session, but once the data is recorded, all subsequent tests can be run on the computer with minimal time cost.

Within the `Debugging Scripts` folder of this git directory, there are several python notebooks that are designed to parse data from a scan and test out different homing methods. In particular, `Fullscan_renders.ipynb` was used to produce the images in the flowchart below, which describes my methodology for checking the fitness function on real data.

6.2.1 Running a high-resolution scan

In this example, the robot was equipped with an immersion transducer and brought into focus manually, bouncing a pulse off the bottom of a wet tank. This setup used an extremely naive fitness function which simply returned the maximum magnitude of the return signal after a timing threshold. Figure 4 shows a sample of the raw sensor data with the fitness function overlaid. This is the setup that was used to produce the data in Figure 3.

As mentioned previously the data in Figure 3 spans a search space of $28mm$ in the translational z axis and $26°$ in the rotational θ_x and θ_y axes. The translational x and y axes were omitted because the bottom of the tank the transducer was bouncing off of was flat, so motion in this plane

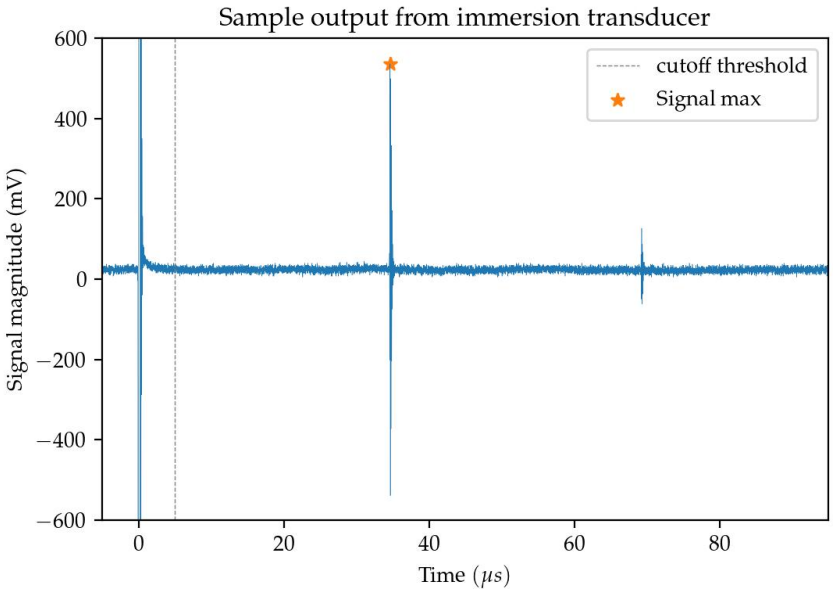


Figure 4: Sample output from the immersion transducer, and the output of a naive fitness function.

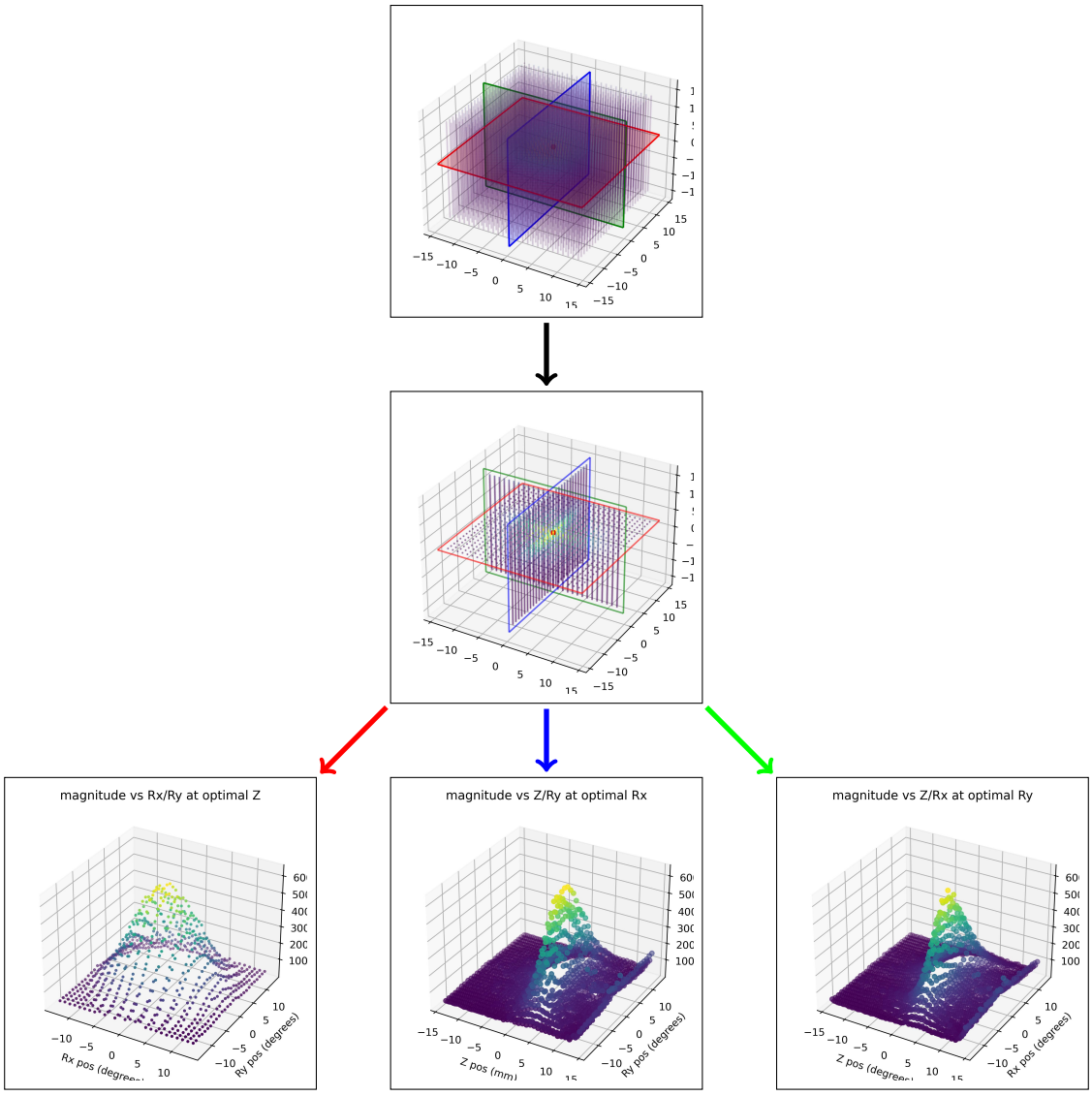
would have no bearing on the output signal strength. We focus the transducer ahead of starting the scan so that we could minimize the size of the search area we needed to look through while still capturing the full shape of the fitness function as it falls off from a fully-focused position to a position where the noise completely blots out the signal.

Higher-resolution data provides us more information about the shape of the fitness function as well as its sensitivity to noise, but the number of points the fullscan needs to visit grows exponentially as the resolution shrinks. The `test_6.json` data has a resolution of 1.0° in the rotational axes and $0.4mm$ in the translational axis; since there are two translational and one rotational axis, the total number of points was equal to $\left(\frac{26}{1}+1\right)\cdot\left(\frac{26}{1}+1\right)\cdot\left(\frac{28}{0.4}+1\right)=51,759$ (Plus-one is added to each range since the motion of the robot includes endpoints)

Since the cost of increased resolution is so high, the resolution and search space volume should be calibrated to capture the minimum number of points that provide you all the information you need about the fitness function. If you know the signal you're measuring falls off within $1.1mm$ of the focal point, don't expand the search space any further than that.

6.2.2 Parsing a high-resolution scan

The full-scan module captures a very large amount of data, but most of it is


magnitude vs Rx/Ry at optimal Z
magnitude vs Z/Ry at optimal Rx
magnitude vs Z/Rx at optimal Ry
Rx pos (degrees)
Ry pos (degrees)
Z pos (mm)
Ry pos (degrees)
Z pos (degrees)
Rx pos (degrees)