NORTHWEST NAZARENE UNIVERSITY

DEFT: Designing a Dexterous Robotic End Effector Capable of Sensing Touch

THESIS
Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF ARTS

Michael Bauman
2023

THESIS
Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF SCIENCE ARTS

Michael Bauman
2023

DEFT: Designing a Dexterous Robotic End Effector Capable of Sensing Touch

Author: 
Michael Bauman

Approved: 
Dale Hamilton, Ph.D., Associate Professor of Computer Science,
Department of Mathematics and Computer Science, Faculty Advisor

Approved: 
Christine Bauman
Second Reader

Approved: 
Barry L. Myers, Ph.D., Chair,
Department of Mathematics & Computer Science

ABSTRACT

DEFT: Designing a Dexterous Robotic End Effector Capable of Sensing Touch
BAUMAN, MICHAEL (Department of Mathematics and Computer Science),
HAMILTON, DR. DALE (Department of Mathematics and Computer Science).

One of the great wonders of the human hand is its ability to sense a variety of aspects of an object. It can sense weight, texture, pressure, hardness, and many other characteristics. Many robotic grippers, up to this point in time, have lacked these qualities. This project, styled DEFT (Dexterous Effector for Fine Touch), seeks to create a five fingered anthropomorphic robotic hand capable of sensing touch. The DEFT system is separated into four subsystems, SOUTHPAW, AR3, SCAM, and TRASH. SOUTHPAW (not an acronym) is the physical implementation of the end effector. It is a 3D printed, five-fingered, tendon actuated robotic hand. The AR3 (Annin Robotics v3) arm serves as a platform for the end effector. SCAM (SOUTHPAW Control and Actuation Module) is responsible for the physical movement of SOUTHPAW. TRASH (Test-driven Robotics Algorithm for Superior Handling) is the software driven control system which is responsible for interpreting requests for movement, forwarding the interpreted requests to SCAM, and responding to sensor feedback from SCAM. The current prototype of DEFT has been shown to be capable of securely holding a 46g payload (form fit) with a diameter of approximately 43mm. The final implementation for DEFT is expected to measure pressure.

ACKNOWLEDGMENTS

I would first like to thank the other members of the DEFT team, Elliott Lochard, Turner Nye, and Noah Smith. This project was for too big for one individual, and without all their hard work the TRASH subsystem wouldn't have a hand or an arm to control.

I also want to thank the faculty and staff of the NNU Engineering and Computer Science departments. In particular, Dr. Dale Hamilton, Dr. Steve Parke, Dr. Duke Bulanon, and April Brown. Without your advice and support this project wouldn't have gotten far.

Also, a big thank you to Micron Technology for funding this project.

And finally, another big thank you to my family and friends for their patience as I spent countless hours buried in the work for this project. I don't know how you put up with me.

Table of Contents

Table of Figures

I. INTRODUCTION

The human hand is an engineering marvel. It is both capable of manipulating small delicate objects and operating under a significant amount load. In short, it is both dexterous and strong. It is a very handy, multipurpose device.

In the world of robotics, there are many different types of devices that perform hand like functions. When speaking in the most generic sense, such devices are often referred to as end effectors. Some examples of end effectors that are currently used in industry will be presented in the background section; however, for now let it be sufficient to state that the majority of such devices are specialized to a specific set of tasks.

A major contributor to the human hand's usefulness as a multipurpose device is the sophistication and resolution of its feedback system. The human hand possesses upwards of 17, 000 touch receptors and nerve endings in the palm alone [1], this allows for incredibly fine motor control and force application.

This project, styled DEFT (Dexterous Effector for Fine Touch), is an attempt to create a robotic end effector that shares some of the characteristics of the human hand; most notably, an end effector that shares a similar level of dexterity and tactility.

This particular document will focus in great detail one of the four subsystems for DEFT. Namely, TRASH (Test-driven Robotics Algorithm for Superior Handling), the software library developed to control the actuation of the end effector. However, to establish proper context, the background section will also touch on the other three subsystems and how they are incorporated. For a more detailed explanation of the other subsystems please refer to the design documentation [2].

II. BACKGROUND

The following section contains relevant background information to this project as a whole. It includes: a short description of some end effectors that are commercially available, a brief description of each of the project's subsystems that will not be discussed in detail, and a description of the hardware platform for TRASH which is useful for later discussions on design choices.

A. Examples of Available End Effectors

Several products for end effectors are currently on the market and can be bought for various purposes. Below are a few items that show the wide variety of devices available for commercial use.

RG2 Series Gripper Systems consist of claw style grippers that are capable of holding 4.4lbs (2kg, hence the name) when carrying a force fit load [3]. The RG2 FT is capable of sensing the force and torque exerted by the claw [4].

Fig. II.A.1. OnRobotics RG2 Series Gripper Systems (Left RG2; Right RG2 FT)

Series GRR Guardian High Capacity Pneumatic Parallel Gripper consists of parallel, pneumatic actuated, plates capable of operation in high load situations, and can survive high impact and shock loads [5].

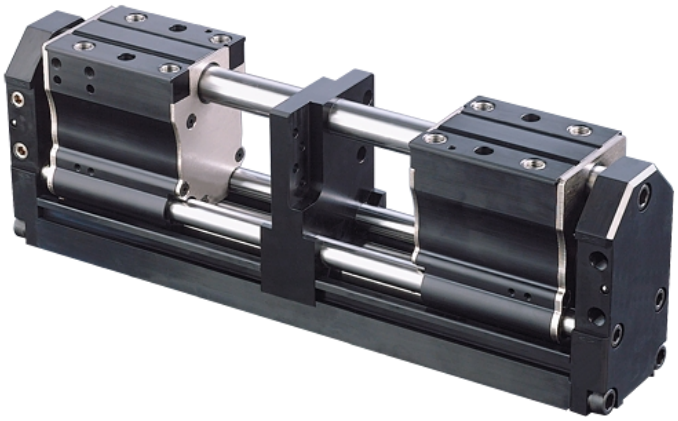

Fig. II.A.2. Series GRR Guardian Parallel Gripper System

This Robot vacuum Gripper, made by Coval, consists of a 3-D printed which uses suction to create a vacuum against the surface of the object to be held [6].

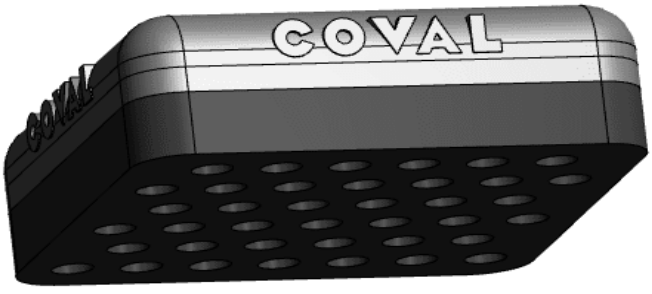

Fig. II.A.3. Coval robot vacuum gripper system

B. SOUTHPAW: The Hand

SOUTHPAW is the name for the physical implementation of the end effector for the DEFT system. The name originates from the hand being designed with a sinistral (left handed) orientation. SOUTHPAW is an anthropomorphic hand with the five conventional digits. The each finger is actuated by a tendon system, with the thumb being an exception and having three tendons to control its movement. The pressure applied by the hand is sensed by measuring the tension in the tendons (more detail in section II.B). An image of the final assembly of SOUTHPAW is shown below in figure II.B.1.

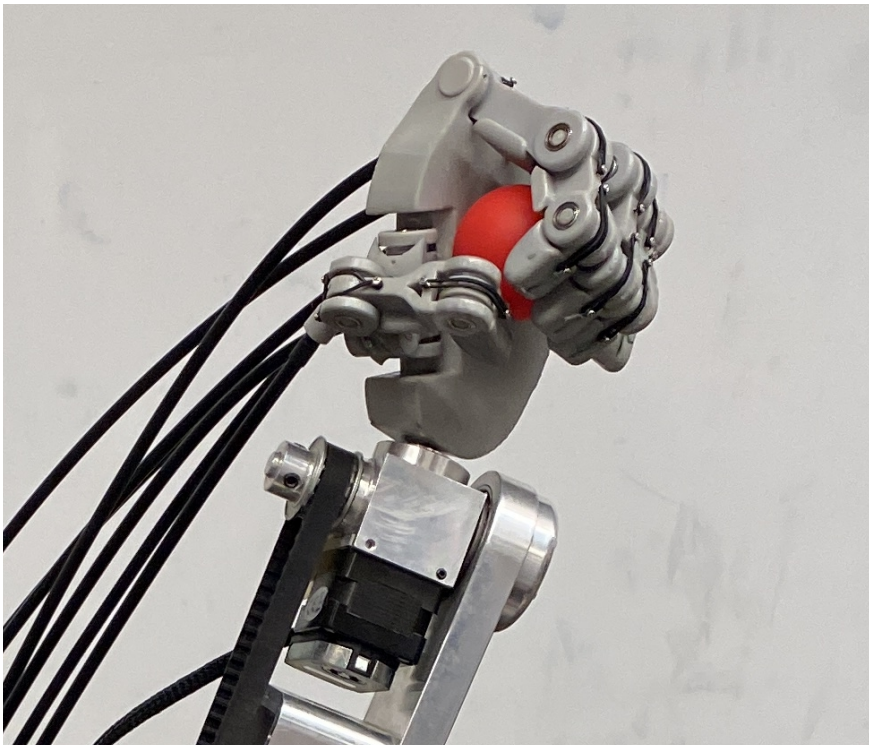

Fig. II.B.1. Final assembly of SOUTHPAW end effector

C. AR3: The Platform

The AR3 (The Anin Robotics v3) arm is a six DoF (Degrees of Freedom) robotic arm which can be purchased and assembled piecewise [7]. The Anin Robotics 3 (AR3) arm serves as a mounting platform for SOUTHPAW. Integrating the SOUTHPAW hand with the AR3 arm improves the system's versatility, by allowing for the hand to be positioned and for supination and pronation (rotation at the wrist) mimicking motion to be achieved.

Fig. II.C.1. 3d Model of the AR3 arm. Be aware that the model does not show any of the driving belts connected.

D. SCAM: The Actuator

SCAM is the SOUTHPAW Control Actuation Module will actuate the SOUTHPAW hand using a Bowden tendon system. The Bowden tendon system is commonly used in bicycle brakes. Each tendon is independently controlled by a stepper motor. These motors move in small discrete steps allowing for precise motion in the hand. As mentioned in section II.B, the pressure applied by the hand is sensed by measuring the tension in the tendons. Tension is measured by using a three roller tension sensor for each tendon (7 in total). An image of a three roller tension sensor is shown in figure II.D.2.

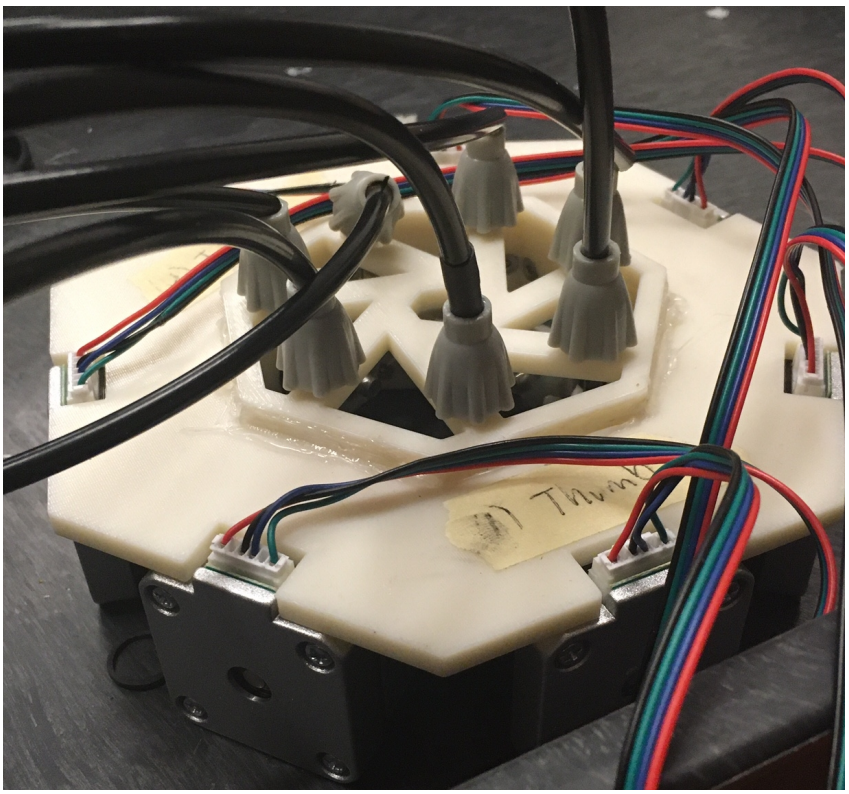

Fig. II.D.1. Final assembly of SCAM (note that one of the tensioners is damaged in this image).

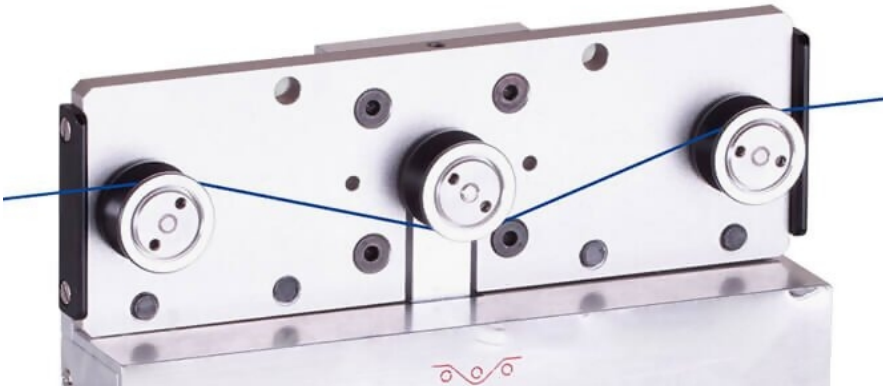

Fig. II.D.2. Example of a three roller tension sensor [8].

E. TRASH Hardware Implementation

The hardware choice for TRASH influences the coding environment for the system. An initial consideration for the system was to use an Arduino microcontroller to manage every aspect of the motor control and sensor feedback. This was feasible; however, such a choice would limit the language for the codebase to be entirely in Arduino code or C. In addition, this would limit the extensibility of the system when trying to interface with an AI controller in the future.

A Jetson Nano was then considered. On its own a Jetson Nano would have permitted development in Python, which for this kind of system is preferable (more on why in the section III.C), and would allow for the system to be extended to permit an AI controller at a later time. With all this said, it was found that a Jetson Nano would not have a sufficient number input channels (pins) to control the necessary number of motors and sensors.

Therefore it was decided to use a Jetson Nano as the main platform for the code with an Arduino acting as the bus for handling communication with all the other devices (motors, sensors, etc...).

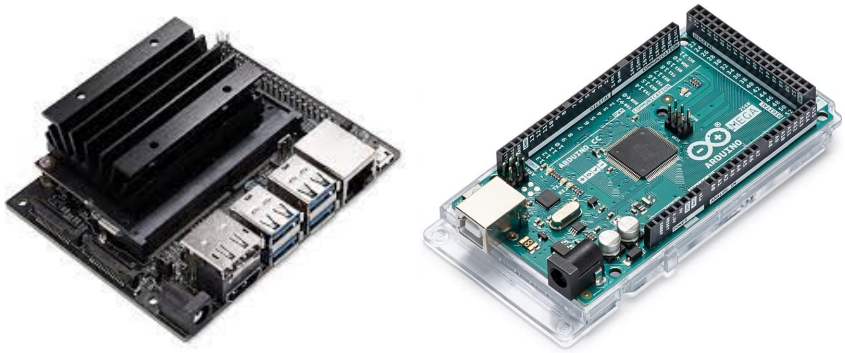

Fig. II.E.1. Jetson Nano (left) and Arduino (right) communication is over USB (not shown)

III. TRASH: THE BRAIN

The following section describes in detail the implementation of the TRASH system and many of the design choices that lead to this implementation.

A. Creating an API: Simplicity is Key

It was decided very early in DEFT's development that it would be best to create an API (Application Programming Interface) for the system. The API has functions for common operations, but is written in such a way that the underlying code handles the complexities of making the function work without the end user needing to know how the function works. In this way control for the system, would take the form of invoking this predefined set of functions and all the hardware interface would be handled behind the

scenes. In the end, this makes performing more advanced movements very simple, and allows an AI actor to interface with the system.

B. Communication Protocol: Say No to I^2C

The initial design for the control system was to use inter-integrated circuit (I^2C or I2C) protocol to communicate between two microcontrollers; however, it was later determined that this would add undue complexity to the design. I^2C is a very simple communication protocol, therefore a specialized communication framework would have been required in order to send commands from one board to the other [9]. In order to implement such a framework, each microcontroller would have required separate programming (and in separate languages). The initial design, the API would have interpreted common operations and converted them into simplified commands that would be transmitted over I^2C to the Arduino. The Arduino would then need code running on it to interpret the I^2C commands and implement them. The Arduino would then execute those commands and report sensor data back to the Jetson Nano via the same I^2C link. As stated earlier, this implementation would have required separate coding for the Jetson Nano and Arduino. This was undesirable due to the complexity of implementation and a larger number of points where failure was likely to occur.

Fortunately, another method was found for managing the Nano-Arduino interface without the need for I^2C. This method takes the form of a pair of Python and Arduino function libraries known as Pymata4 and Firmata Express. Pymata4, allows an operating system (like Ubuntu or Windows) to directly control an Arduino with Python over a USB

connection. The Firmata library is an API of sorts for Arduino. It is part of the default Arduino library. Firmata interprets commands received over a USB link and performs functions based on those commands. Firmata Express is an improved version of the Firmata protocol. Pymata4 does not require Firmata Express, but it benefits from improved performance when it is loaded. A Jetson Nano is designed to run an Ubuntu operating system and has multiple USB ports. This makes the Pymata4-Firmata Express combo an ideal communication protocol for the TRASH control system.

C. Python: The Desirable Choice

Python was the original language choice for creating the TRASH control system. A major contributor to this choice was the skill set of the team: several members had experience working in Python. In addition, Python is platform independent and could be developed and tested simultaneously on devices other than the Jetson Nano itself.

However the final decision was sealed once the Firmata protocol was discovered. While there are libraries for other languages that interface with Firmata, Pymata4 is specific to Python, is well developed, and is compatible with the improved Firmata Express.

D. Test-Driven Development

The code for this system is written using a test driven development framework. Test driven development is a core agile practice, which leans heavily into the iterative nature of agile development. In test driven development the coder creates a series of basic

tests for the control system prior to actually writing any code. The code is then written to meet the test requirements and then more tests are written in order to tighten the constraints on the control systems operation (i.e. the tests become progressively more rigorous as the system improves).

The benefit of such a coding approach is that the programmer can more quickly determine what will and will not work for the given implementation and generates working code quickly. In addition, the testing framework acts as a robust means of ensuring that large scale refactors do not produce uncaught errors in connected pieces of code. This is because the test framework allows tests to be automated, and allows for testing the code in small sections at a time. Because of this, errors due to refactoring are generally caught early (if something breaks, a test will fail).

E. DOD: Data Oriented Design

The internal workings of the API are setup according to the DOD (Data Oriented Design) approach. DOD is an approach to data management that has its origins in video game design [10]. The reason for using it in this project is that it is very efficient when used in realtime applications. In the case of this project, the realtime environment is that of a moving hand. When a command is given to move the hand the desire is for a near instantaneous response, and if another command is given for the system to immediately adapt.

The philosophy behind DOD is that most of the state data in an application should come from where the data is stored, or rather where the data's key is stored. That is to say

that if a piece of data (an object for instance) is has its key stored in a given location then a certain set of operations must be performed on it until the key is moved somewhere else; this why it is called data oriented. In fact, the data storage for such a system is designed to mimic a database (including normalization, composite keys, etc…) [10].

The specific implementation for this project is quite simple but still uses this approach. Data for controlling each motor in the system is stored in a master table, and a primary key is assigned to each motor in the system. There are two tables associated with actions: a ready table and a moving table. If a motor is not moving, its key sits in the ready table. If a command is sent to move it, the motor's key is placed in the moving table along with the relevant commands (how far to move for example). The functions specific to moving continue to run until the command's conditions are met. If a command is sent to stop (this can interrupt a previous command), the key is pulled from the moving table and placed back into the ready table. The beauty in such a system is that all the operations can appear to be occurring in real-time because it can switch between tasks fairly quickly and doesn't add much overhead when new objects are added because each object only adds load based on the table it is in (if an object is in the ready table nothing needs to be done to it.

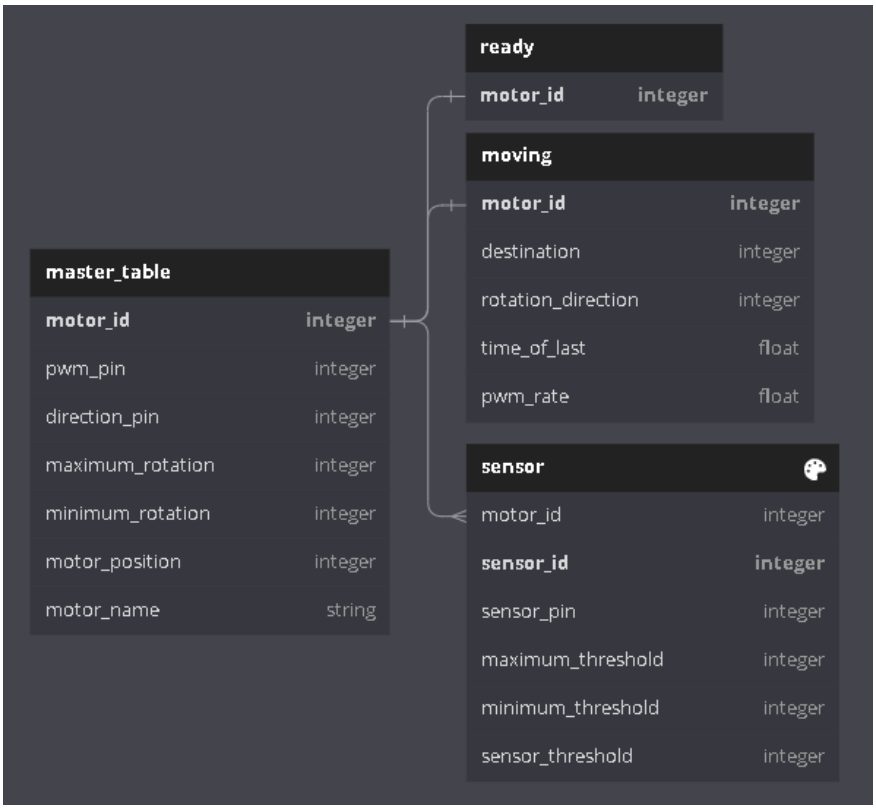

Fig. III.E.1. Database diagram of the tables used in TRASH

IV. DISCUSSION

At times this project was quite difficult but also quite enjoyable. Unsurprisingly most of the issues that arose in development were in the mechanical and electrical systems. The current prototype suffered a failure in the electrical system. A current overdraw caused the stepper motors to overheat and caused mechanical failures elsewhere in the system. The issue is currently being investigated and most indicators seem to point to faulty stepper motor controllers. Replacement of these controllers will

likely fix the issue. In addition, this mode of failure could likely be avoided in the future by using stepper controllers with a digital fault detection system.

Prior to the electrical system failure, the current prototype of DEFT was capable of grasping and manipulating a 46g ball with a diameter of approximately 43mm (a golf ball). DEFT is also capable of sensing pressure through the use of tension sensors; however, full testing and calibration of the tension sensors was not completed due to project time constraints. In other words, metrics for sensor sensitivity has not been evaluated. The hope at this point is that the system developed here will be useful in furthering the development of hand-like end effectors that are touch sensitive.

Finally, a major means of improving the systems control loop would be to add in a machine vision system. This would allow the system to not only verify the how an object is being grasped but also allow for the autonomous location and retrieval of objects. It is hoped that development will be continued on this project to include these systems and an AI controller.

References

[1] National Center for Biotechnology Information, "Anatomy and Physiology of the Hand," in StatPearls, Treasure Island (FL): StatPearls Publishing, Jan. 2022. [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK279362/#:~:text=Our%20hands%20also%20have%20very,nerve%20endings%20in%20the%20palm. [Accessed: Apr. 27, 2023].

[2] M. Bauman, E. Lochard, T. Nye, and N. Smith, "DEFT Design and Development of a Dexterous Robotic End Effector," Engineering Physics Department, Northwest Nazarene University, May 2023. [Online]. Available: https://docs.google.com/document/d/1ImQOq9_SLvjAdKXxjlsvd9L2fSrNfcmGVFGLHOQBaX4/edit. [Accessed: Apr. 27, 2023].

[3] OnRobot, "RG2 Gripper Datasheet v1.4," OnRobot, Apr. 2021. [Online]. Available: https://onrobot.com/sites/default/files/documents/Datasheet_RG2_v1.4_EN.pdf. [Accessed: Apr. 27, 2023].

[4] OnRobot, "RG2-FT Gripper Datasheet v1.2," OnRobot, Nov. 2019. [Online]. Available: https://onrobot.com/sites/default/files/documents/Datasheet_RG2-FT_v1.2_EN.pdf. [Accessed: Apr. 27, 2023].

[5] PHD Inc., "Series GRV2 Gripper," PHD Inc., 2015. [Online]. Available: https://www.phdinc.com/pdf/6441524.pdf. [Accessed: Apr. 27, 2023].

[6] Coval Vacuum Technology Inc., "Coval Vacuum Gripping Systems," Coval Vacuum Technology Inc., Jan. 2022. [Online]. Available:

https://doc.coval.com/g/MVG/doc/mvg_doc_coval_2022_v02-01_us.pdf. [Accessed: Apr. 27, 2023].

[7] Annin Robotics, "Annin Robotics," Annin Robotics, 2022. [Online]. Available: https://www.anninrobotics.com/. [Accessed: Apr. 27, 2023].

[8] ABQ Industrial, "Hans Schmidt MZ1 Narrow Body 3-Roller Tension Sensor," ABQ Industrial, 2021. [Online]. Available: https://www.abqindustrial.net/store/hans-schmidt-mz1-narrow-body-3-roller-tension-sensor-p-2075.html#cl-group-1. [Accessed: Apr. 27, 2023].

[9] K. Hemmanur, "Inter-Integrated Circuit (I2C)," Michigan State University, 2009. [Online]. Available: https://www.egr.msu.edu/classes/ece480/capstone/fall09/group03/AN_hemmanur.pdf. [Accessed: Apr. 27, 2023].

[10] R. Fabian, "Data-Oriented Design book," 2022. [Online]. Available: https://www.dataorienteddesign.com/dodbook/. [Accessed: Apr. 27, 2023].

[11] M. Bauman, "TRASH-alt," GitHub. [Online]. Available: https://github.com/TRASHPythonDev/TRASH-alt. [Accessed: Apr. 27, 2023].

Appendix A: Code

Below is the code for the main module of deft. For all necessary code refer to [11].

```
import motor
from time import perf_counter
from stepper_manager import StepperManager
from sensor_manager import SensorManager
from movement_arduino import BoardManager


class BasicMovement():
    __current_motor_id = 0
    __current_sensor_id = 0

    def __init__(self, board_id = 1) -> None:
        self.__stepper_manager = StepperManager()
        self.__sensor_manager = SensorManager()
        self.__board_manager = BoardManager(board_id)
        pass

    # clears reference to board and clears connection, call before
calling del on BasicMovement instance
    def release_board(self) -> None:
        self.__board_manager.release_board()

    # this function cycles all objects in the moving list and moves them
    # the rate at which this is called determines base motor turning
rate and sensor sample rate
    # running this at a value around or below 0.0001 times per second
    def move(self) -> None:
        moving_list_ids = self.__stepper_manager.get_moving_list_ids() #
get list of all motor ids in moving list
        for motor_id in moving_list_ids: # cycle through all motor ids
in moving list
            motor_data =
self.__stepper_manager.get_master_data_by_id(motor_id) # get motor data
for given id
            moving_data =
self.__stepper_manager.get_moving_data_by_id(motor_id) # get movement
data for given id
            sensor_id_list =
self.__sensor_manager.get_all_associated_for_motor_id(motor_id) # get
sensor id list for given motor id
            movement_complete = False # set movement complete flag for
this motor to false
            for sensor_id in sensor_id_list: # cycle all sensors
                sensor_data =
self.__sensor_manager.get_master_data_by_sensor_id(sensor_id) # for
given sensor id get the sensor data
```

```
                sensor_reading =
self.__board_manager.read_analog_pin(sensor_data["sensor pin"]) # read
analog data for given sensor
                if(sensor_reading >= sensor_data["sensor threshold"] *
(sensor_data["max threshold"] - sensor_data["min threshold"]) / 100): #
compare sensor data to threshold percentage
                    movement_complete = True # if sensor reading exceeds
sensor threshold percentage, we are done moving this motor
            # if movement in range continue
            if(motor_data["max rotation"] > motor_data["motor position"]
and motor_data["min rotation"] <= motor_data["motor position"]):
                pass # this line does nothing other than keep python
happy
            else: # if outside of range then stop
                movement_complete = True # if motor position is outside
range, we are done moving the motor
            # if movement is not done, and we have exceeded minimum rate
and have not reached destination then step
            if(movement_complete == False and perf_counter() -
moving_data["time of last"] > moving_data["rate"] and
moving_data["destination"] > motor_data["motor position"]):
                self.__board_manager.step(motor_data["pwm pin"]) # step
the stepper motor forward one pulse

self.__stepper_manager.set_motor_position_by_id(motor_id,motor_data["mot
or position"] + 1) # increment position counter
            elif (moving_data["destination"] >= motor_data["motor
position"]): # if motor position is at or exceeds destination we are
done moving
                movement_complete = True # if motor position is at
destination range, we are done moving the motor
            if(movement_complete == True): # if movement complete flag
set
                self.__stepper_manager.move_to_ready(motor_id) # move
motor to ready table if done
        pass

    # sets motor to move to a given destination and stop when certain
thresholds are met or when destination reached
    # motor_id: int             - must be valid motor id, if no motors
have been deleted then this is simply an integer < __current_motor_id
    # destination: int          - destination in steps, if this is for a
nema 17 stepper then 400 is a full rotation
    # sensor_thresholds: list   - as a percentage of each sensor's
range, if greater than 100 movement becomes blind, this is stored
between calls
    # rotation_direction: int   - either 0 or 1 no idea which is cw or
ccw, but this can be determined experimentally
    # pwm_rate: float           - this has only been tested for values
>= 0.0001, this is ridiculously fast for a nema 17
    def move_motor_to_point(self, motor_id: int, destination: int,
sensor_thresholds: list, rotation_direction = 0, pwm_rate = 0.05) ->
int:
```

```
        if (self.__stepper_manager.get_if_id_in_use(motor_id) == False):
# if motor does not exist kill method
            return -1 # exit prior to anything, return failure
        if (self.__stepper_manager.get_if_id_in_moving_list(motor_id)):
# if motor already moving kill movement
            self.__stepper_manager.move_to_ready(motor_id) # put motor
in ready state, kill movement
        motor_data =
self.__stepper_manager.get_master_data_by_id(motor_id) # get data for
given motor id
        new_moving_data = motor.create_moving_data(motor_id,
destination, rotation_direction, pwm_rate, 0) # create data to pass to
stepper manager
        direction_pin = motor_data["direction pin"] # get direction pin
out for given motor
        self.__board_manager.set_stepper_direction(direction_pin,
rotation_direction) # set rotation direction and pin based on data
retrieved by id
        self.__stepper_manager.move_to_moving(new_moving_data) # place
moving data in moving table
        motor_sensor_id_list =
self.__sensor_manager.get_all_associated_for_motor_id(motor_id) # create
list of sensors associated with motor
        j = 0 # set increment for sensor threshold list to 0
        for sensor_id in motor_sensor_id_list: # cycle through each
sensor associated with motor
            if j < len(sensor_thresholds): # if we haven't run out of
inputs in the sensor thresholds list then continue

self.__sensor_manager.set_sensor_threshold_by_id(sensor_id,
sensor_thresholds[j])
                j = j + 1 # increment through sensor threshold list
            else: # if we have exceeded the number of threshold inputs
then break out of loop
                break # cancel loop
        return 0

    # sets motor to move to a given destination and stop when
destination reached this method ignores sensors
    # motor_id: int             - must be valid motor id, if no motors
have been deleted then this is simply an integer < __current_motor_id
    # destination: int          - destination in steps, if this is for a
nema 17 stepper then 400 is a full rotation
    # rotation_direction: int   - either 0 or 1 no idea which is cw or
ccw, but this can be determined experimentally
    # pwm_rate: float           - this has only been tested for values
>= 0.0001, this is ridiculously fast for a nema 17
    def move_motor_to_point_blind(self, motor_id: int, destination: int,
rotation_direction = 0, pwm_rate = 0.05) -> int:
        sensor_thresholds =
[200,200,200,200,200,200,200,200,200,200,200,200,200,200,200,200,200,200
,200,200,200,200,200,200,200] # threshold so high cannot reach, enough
for 25 sensors
```

```
        return self.move_motor_to_point(motor_id, destination,
sensor_thresholds, rotation_direction, pwm_rate) # call normal move to
point but with custom thresholds


    # create a new motor to control
    # pwm_pin: int              - arduino digital pin or pwm pin to set
to digital output mode
    # direction_pin: int        - arduino digital pin to set to digital
output mode
    # maximum_rotation: int     - maximum rotation for stepper motor 400
for nema 17 stepper motor
    # minimum_rotation: int     - minimum rotation for stepper motor 0
for nema 17 stepper motor
    # motor_position: int       - current motor position as an integer
defaults to 0
    # motor_name: str           - is just a freindly name as a means of
alternate lookup
    def create_motor_and_add(self, pwm_pin: int, direction_pin: int,
maximum_rotation = 400, minimum_rotation = 0, motor_position = 0,
motor_name = "") -> int:
        if(maximum_rotation < minimum_rotation): # if rotation range is
reversed
            maximum_rotation = 400 # set to defaults if rotation range
is bad
            minimum_rotation = 0 # set to defaults if rotation range is
bad
        new_motor = motor.create_motor(BasicMovement.__current_motor_id,
pwm_pin, direction_pin, maximum_rotation, minimum_rotation,
motor_position, motor_name) # create new motor for assignment to stepper
list
        self.__stepper_manager.add_stepper(new_motor) # add motor to
stepper list
        self.__board_manager.activate_pins_for_output([pwm_pin,
direction_pin]) # activate pins for output
        BasicMovement.__current_motor_id =
BasicMovement.__current_motor_id + 1 # increment motor id number
        return BasicMovement.__current_motor_id - 1 # return the id just
created

    # create new sensor to read
    # motor_id: int             - must be valid motor id, if no motors
have been deleted then this is simply an integer < __current_motor_id
    # sensor_pin: int           - arduino analog pin to set to analog
input mode
    # maximum_threshold: int    - maximum sensor "voltage" reading
generally about 1000 for arduino analog input with supply of 5 volts,
600ish with supply of 3.3 volts
    # minimum_threshold: int    - minimum sensor "voltage" reading
generally about 0 is good, will likely be higher for our tension sensors
    # sensor_threshold: int     - sensor threshold percentage 0 to 100
range, if set higher than 100 will result in sensor being ignored
```

```
    def create_sensor_and_add(self, motor_id: int, sensor_pin: int,
maximum_threshold = 1000, minimum_threshold = 0, sensor_threshold = 100)
-> int:
        new_sensor = motor.create_sensor(motor_id,
BasicMovement.__current_sensor_id, sensor_pin, maximum_threshold,
minimum_threshold, sensor_threshold) # create new sensor for assignment
to sensor list
        self.__sensor_manager.add_sensor(new_sensor) # add sensor to
sensor manager

self.__board_manager.activate_pins_for_analog_input([sensor_pin]) #
activate pin for input
        BasicMovement.__current_sensor_id =
BasicMovement.__current_sensor_id + 1 # increment sensor id number
        return BasicMovement.__current_sensor_id - 1 # return the id
just created

    # returns motor id given the name of the motor
    # motor_id: int             - must be valid motor id, if no motors
have been deleted then this is simply an integer < __current_motor_id
    def get_motor_id_by_name(self, motor_name: str) -> int:
        return
self.__stepper_manager.get_master_data_by_name(motor_name)["motor id"]

    # returns position of given motor
    # motor_id: int             - must be valid motor id, if no motors
have been deleted then this is simply an integer < __current_motor_id
    def get_motor_position(self, motor_id: int) -> int:
        return self.__stepper_manager.get_master_data_by_id(motor_id)
["motor position"]

    # sets position of given motor, useful for nulling on startup
    # motor_id: int             - must be valid motor id
    # motor_position: int       - current motor position to be set
    def set_motor_position(self, motor_id: int, motor_position: int) ->
None:
        self.__stepper_manager.set_motor_position_by_id(motor_id,
motor_position)
        pass

    # returns threshold of given sensor
    # sensor_id: int             - must be valid sensor id, if no
sensors have been deleted then this is simply an integer <
__current_sensor_id
    def get_sensor_threshold(self, sensor_id: int) -> int:
        return
self.__sensor_manager.get_master_data_by_sensor_id(sensor_id)["sensor
threshold"]

    # returns all sensors associated with a motor
    def get_all_sensor_ids_for_motor(self, motor_id: int) -> list:
        return
self.__sensor_manager.get_all_associated_for_motor_id(motor_id)
```

```
    # returns a list of all motor ids that are ready
    def get_all_active_motor_ids(self) -> list:
        return self.__stepper_manager.get_moving_list_ids()

    # returns a list of all motor ids that are moving
    def get_all_ready_motor_ids(self) -> list:
        return self.__stepper_manager.get_ready_list_ids()

    # checks if motor still exists and deletes if it does, checks for
associated sensors and deletes them also
    # motor_id: int             - must be valid motor id, if no motors
have been deleted then this is simply an integer < __current_motor_id
    def delete_motor(self, motor_id: int) -> None:
        if (self.__stepper_manager.get_if_id_in_use(motor_id)):
            self.__stepper_manager.remove_stepper(motor_id)
            sensor_id_list = []
            sensor_id_list =
self.__sensor_manager.get_all_associated_for_motor_id(motor_id)
            for id in sensor_id_list:
                self.delete_sensor(id)
        pass

    # checks if sensor still exists and deletes if it does
    # sensor_id: int             - must be valid sensor id, if no
sensors have been deleted then this is simply an integer <
__current_sensor_id
    def delete_sensor(self, sensor_id: int) -> None:
        if (self.__sensor_manager.get_if_id_in_use(sensor_id)):
            self.__sensor_manager.remove_sensor(sensor_id)
        pass

    @classmethod # when requested will return the id of the next motor
to be instantiated
    def get_next_motor_id(self) -> int:
        return self.__current_motor_id

    @classmethod # when requested will return the id of the last motor
to be instantiated
    def get_last_motor_id(self) -> int:
        return self.__current_motor_id - 1

    @classmethod # when requested will return the id of the next sensor
to be instantiated
    def get_next_sensor_id(self) -> int:
        return self.__current_sensor_id

    @classmethod # when requested will return the id of the next sensor
to be instantiated
    def get_last_sensor_id(self) -> int:
        return self.__current_sensor_id - 1
```