Usan Siriwardana

2025/08/22

This document is for whoever is working on Hongchen’s Auto-Membranes SDL in Jay Werber’s lab in WB366 after me (after Summer 2025)

# Intro

The machine is meant to allow for high-throughput automated manufacturing and testing of different formulations of reverse osmosis membranes. It is not quite done, but getting there. As of writing, I am the last student to have worked on the system, so this document is to help get the next person (hopefully that’s you!) familiar with the system, how it currently works, and what needs to be done.

# Components

The machine is composed primary of 3 robotic parts, which were purchased commercially. An Opentrons OT-2 liquid handling robot (left), a UFactory X7 xArm robotic arm (middle), and a Testresources Newton compression tester (right).

In addition, there is also an Arduino which is used to interface with the compression tester (white box in front of the OT-2), the Arduino also controls a gas solenoid, and there is also an IKA RC2 chiller (under the bench, under the OT-2).

The entire system is housed inside of a ductless fume hood, that uses carbon filters to clean the air that passes through. Because it isn’t ducted, we can’t try to evaporate significant amounts of solvent, instead we use very long NIPS bath time to form the membranes.

# Workflow

The goal of the machine is to make thin polymer membranes. Currently, the main tuneable parameter for using the system as an SDL is the polymer concentration. The Opentrons will have 2 media bottles, one with a stock solution of polymer in solvent (slot 9), and another with pure solvent (slot 6). The Opentrons will draw specific volumes of liquid from each bottle and mix them in one of the wells of the heating block. This process takes a long time, as the solution can be very viscous. The Opentrons will then pick up the newly mixed solution, and deposit it onto the coupon, in 6 droplets, then wait for the droplets to coagulate into a single, roughly linear puddle on the coupon. (this is done instead of dispensing in a line because the Opentrons does not let you dispense while moving the pipette laterally.)

The arm will pick up the knife from cleaning stage, and use it to cast the membrane on the coupon surface. Any excess solution will be pushed over the edge and caught in the coupon stand. Dry nitrogen is blown over the coupon to maintain a low-humidity environment around the coupon, while the arm grabs the nitrogen cap. The cap is placed over the coupon, holding the dry nitrogen, maintaining the low-humidity while the coupon is transferred to the NIPS bath.

When the coupon is transferred to the NIPS bath, the cap is lifted off of the coupon, and water is allowed to react with the solution, forming the membrane. The arm picks up a ring from the ring stand, and places it on top of the coupon in the NIPS bath, to keep the membrane from floating if it detaches from the coupon. The NIPS reaction can take some time, and the properties of the membrane will change with different reaction times, so we wait for some time.

# Software architecture

All of the code for the system is written in python, and runs on the Raspberry Pi from the Opentrons. The easiest way to edit the code is to open the jupyter notebook from the connected laptop (open a browser such as chrome and navigate to <http://169.254.46.48:48888>), and navigate to the folder called “2025-07-02”. Inside, there are 4 things that are important to the system. First there is a folder called “lib”. This folder contains the class definitions for 4 of the separate hardware components of the system. These are stored in “ot2.py”, “arm.py”, “chiller.py” and “arduino.py”. The point of each of these classes is to present a very simple and easy to use interface for the protocol file.

The file “ot2.py” contains the class definition for our opentrons object. The object keeps track of the current state of the tip rack, and the heater rack, so that it knows which tip to pick up next, and which wells in the heater are clean and ready to use. This is done by keeping track of the current “tip index” and “heater well index”, which each tell you how many tips/heater wells have been used. Several parameters that are primarily relevant to the opentrons, such as the concentration of the stock solution, are stored in this file.

The file “arm.py” contains the class definition for the arm object. The space in the fume hood is split into 3 “zones”, called “opentrons”, “middle” and “tester”. Other “items” are defined in this file, which all must be inside of one of these zones. Each of these items has an “item waypoint”, which the arm will move to before and after picking up and putting down each item. Note that an “item” in this context does not represent a real physical object, but rather a position and rotation in which a physical object could be. Just as each item has an item waypoint, each zone has a “zone waypoint”. These are coordinates that the arm will use when moving between zones. These are chosen so that the arm won’t crash into objects in the fume hood when moving between different zones. Several movement commands have a parameter called “pitch”, which is true by default. This parameter controls whether the arm will change its current pitch to match the pitch specified in the coordinates of the destination. Set this parameter to false if you want to make sure that the arm does not tilt as it moves, useful if the arm is carrying several objects in a stack, such as the coupon with the cap on it, or the coupon with a ring on it.

In general, the idea behind the arm.py file was to prefer simple “macros” over a complex system that could, for instance, keep track of objects and their positions at all times. That is not to say that it would be a bad idea, but just that such a level of detail was not necessary at the time, and would only increase software complexity. However, it could be beneficial to add such a system in the future, as it may help with adding useful multithreading.

The file “arduino.py” contains the class definition for the arduino object. The arduino is running a firmware called “firmata”, which allows us to communicate with it over usb serial, effectively turning it into an IO expander. The arduino is used for controlling the nitrogen blower solenoid through a relay, as well as communicating with the compression tester. As the compression tester does not have any kind of API that we can use, we have wired two of the ports of the compression tester to the arduino. One of these ports, is configured as an output for the arduino. The pin is normally pulled high, but the arduino can pull it low when required. The compression tester watches when this pin goes low, and when it does, starts running the compression test. Note that when the arduino object is initialized, this pin may briefly be puled low, accidentally starting a compression test. Either unplug the arduino cables from the compression tester while initializing the arduino and then plug them back in after, or be prepared to stop the compression tester. The second port is configured as an input for the arduino, and is tied to DIO2 in the compression tester software. The compression tester is configured to give a low output on DIO2 by default, switch to a high output while the test is running, and then move back to a low output after the test finishes. The arduino reads this pin to tell first that the compression test has started, and then to tell when it has finished. As of writing, this is not always 100% accurate, and the arduino sometimes thinks the compression test has finished prematurely. A fix has been added, but not extensively tested. Note that the compression tester will not function unless the connected laptop is running the “Newton” software, and shows up as connected.

The file “chiller.py” contains the class definition for the chiller object. The chiller is connected to the Raspberry Pi over usb serial, and we’re using an unofficial IKA python library so we don’t have to decode the serial messages ourselves. The API allows us to set the target temperature, then wait until the measured temperature is within tolerance. This code should eventually be run in a separate thread, so that the machine can continue doing useful work while waiting for the chiller to reach temperature.

There are also 2 .json files in the main directory, called “parameters.json” and “robot.json”. Parameters is used to define the tuneable parameters used for the next run of “protocol.py”. The python script will read the .json file to set parameters such as the weight percent of the polymer in solution, the pullcasting speed, the NIPS bath temperature, etc. The other .json file, Robot, is used to store information about the system, that will change, but needs to be carried between runs of the robot. It currently keeps track of how many coupons are in the clean coupon pile, how many rings are on the stand, how many assemblies of coupons and rings are in the discard pile, and the tip index and heater well index from the opentrons section.

The main point of entry for those working on the machine learning side of the project is “protocol.py”. It’s a python script that uses the aforementioned classes to define how a single run of the protocol should be done. To run the script (not a notebook), open a terminal/ssh session, and navigate to “/var/lib/jupyter/notebooks/2025-07-02/” then run “python3 protocol.py”

## Waypoints (arm)

In the arm.py file, there is a large dictionary which contains a bunch of pairs of coordinates and their corresponding names. Many of these entries have a matching entry, which have the same name but with the word “waypoint” appended to the end. When using the arm class’s pick\_up() or put\_down() methods, the arm will move to the matching waypoint before moving to the object’s position. This is to avoid crashing into objects in the work area. If you need to define new objects, I suggest you use the same system. Normally, the waypoint for an object will have the same coordinates, but have a larger “z” value, meaning the arm will over above the object, before coming down to grab it, and then lift it into the air after.

# Hardware (electronics)

The majority of the electronics were purchased off-the-shelf, and connect to the Raspberry Pi or Windows Laptop over USB or Ethernet. However, some parts required some manual wiring. There is an arduino uno board connected to the Raspberry Pi, running firmata. This arduino is connected to a relay, which in turn is connected to a solenoid valve, that is used to blow nitrogen over the coupon while it is in the opentrons. There are 3 wires that connect between the arduino box and the compression tester, used to start the compression tester, as well as determinine if the test is running or not.

# TODO

As of the time I left, the intermediate stage that is used to transfer between holding the coupon from the top (used when moving the coupon between the pile/opentrons/bath/intermediate stage) and holding it from the side (required when placing the coupon on the compression tester) has broken. The height of the stand needs to be adjusted, and the “ramp” made larger, so that when the robot arm “slides” the coupon off the compression tester and onto the stand, it will prefer to slide up the ramp, rather than breaking the stand.

The new compression tester/arduino integration needs to be tested thoroughly, to make sure that the arduino does not accidentally not see that the test has started, nor should it think that the test has ended early.

The chiller does have a temperature sensor, but it can only monitor the temperature of the coolant fluid, that runs through the copper coils. This means that we cannot currently monitor the actual temperature of the NIPS bath. It may be helpful in the future to place a thermocouple or similar temperature probe inside the NIPS bath, to monitor the temperature directly. It would likely be easiest to read this temperature sensor through the arduino.

The humidity of the environment has a large impact on the membrane structure, and while we do our best to keep the environment directly around the membrane as controlled as possible by flowing dry nitrogen over the coupon during casting, it might be good to monitor the humidity and temperature inside the fume hood. There do exist humidity meters that have an I2C interface, so talk to them using the arduino uno as well.

It is very important that the knife is dry before casting a membrane. The towel in the cleaning stage does a pretty good job at this, but it may become less effective if it gets saturated, which could happen if the humidity in the fume hood is high. It may make sense to use the existing nitrogen blower to dry off the knife, perhaps by having the arm hold the knife in front of the nozzle for some time.

I heard talk about adding a computer vision module to the pipeline, just before the compression tester, where the coupon would be placed in a box with controlled lighting and a camera. A picture of the membrane would be taken and analyzed to determine the best locations for the compression test. This module is not currently implemented.

The ring stand can only hold 3 rings at the moment, which means at most 3 runs can be done before a human needs to restock the rings. A new design should be made, or the current design extended to hold more rings, so that the machine could run for longer between restocking.

The code currently does not throw python exceptions when an error or something unexpected occurs. The code should be checked and modified to do this. Several functions will return 0 on success and 1 on failure. This should be changed to throwing an exception on failure, and exiting the program. We should want the machine to stop what it is doing if an error occurs, not to keep going while things break. Note that this may be somewhat difficult when dealing with the arm, as you need to poll the controller to check if the arm is in an errored state. The protocol file should also be modified to save the current robot parameters in case of such an event, and perhaps try to return the robot arm to home, or put it into teaching mode, so it can be recovered manually.

It may make sense to add more data to robot.json, especially if implementing a more complicated system to keep track of object positions. Theoretically, if we can change the behaviour of the xArm to return to home when initializing, this could allow us to stop the machine at any time, and just recover and continue with the same protocol.

The calculation for mixing the solution and solvent for making the desired weight percent is done using mass balance, which requires the density of the solution and solvent. The parameters currently used may not be the correct ones, so double check them.

# Tips & Tricks

The xArm has a feature called “teaching mode”, which allows you to pose the arm manually. The force sensors in the joins will feel that you are trying to move the arm, and the motors will assist you. This is useful if you crash the arm, as trying to re-instatiate the xArm object to clear the error will have the xArm try to return to home, normally moving in a straight line. But if the xArm runs into an obstacle as it tries to home, it will error and refuse to move. If this happens, you can put the xArm into teaching mode, then move it close to the home position by hand. Then clear the errors, and try to re-instantiate the arm.

The teaching mode is also useful for plotting coordinates, as the xArm’s functions to get the current coordinates still work while in teaching mode. Simply enter teaching mode, move the arm to the desired position, then ask the controller for the arm’s current coordinates.

The compression tester will only work while the newton software is open on the laptop, and is connected to the controller. Sometimes, the software will say that it isn’t connected when it should be. If this happens try the following: Restart the controller (flip the green power switch off, wait a few seconds, turn it back on, wait at least 30 seconds and try to connect again), refresh the newton software, close and re-open (and re-login) to the newton software, restart the windows laptop, try unplugging and re-plugging all the cables in between the compression tester and the laptop.

The blade cleaning stage, NIPS bath, and outer cooling bath all need to be filled with water occasionally, as they loose water by evaporation. For the blade cleaning stage, fill the water bath enough to completely wet the bottom of the blade when it’s placed inside. For the NIPS bath, place the coupon and cap in the bath. The water line should be enough to completely cover the coupon, but not high enough to touch the nitrogen cap. Be careful not to overfill the outer cooling bath, or else the inner NIPS bath may start floating.

If you have restarted the newton compression tester software, it may have reset the zero points. Place a coupon under the pin, and manually jog the pin down until there is only a very small gap between the coupon and pin. Be EXTREMELY careful, as if you over shoot, especially with the fast jog, you can max out the load cell very quickly, which can cause damage. Use the buttons in the newton software to tare the channels, then lift the pin to approximately -8 mm. This is to give the arm lots of space under the pin to move the coupon. Make sure the test profile is set to “Compression Test – Arduino”, which is the test that has been modified to automatically fast jog down to -1 mm, run the compression test, then retract to -8 mm, all while giving the arduino the appropriate signal to show that the test is running.

The password for the laptop is “opensesame”

The password for the account for the Newton compression tester software is “135” (or maybe “153”? I just use muscle memory so I don’t remember but it’s definitely one of those)