Steps:

* Robot repo: <https://github.com/Image-X-Institute/6-DoF-Robotic-Motion-Phantom/tree/main>
* 1D platform repo: https://github.com/Image-X-Institute/Motion-surrogate-platform/tree/main/1D-couch-applications-master/BMET3921/1029updates.
* Review documentation for the robot and 1D platform. Feel free to edit where needed and update as the project progresses.
* Connect the robotic arm with the 1D platform.

1. Run the 1D platform using the Python code. (Done)
2. Convert the Python code in C# and run. (Done)
3. Run the robot software.
4. Disable the send motion function in the robot software and integrate the 1D platform code with the robot code.
5. Enable send motion function in the robot software and integrate the two.
6. Keep an option to run them separately, as well as, parallelly.

* 1D platform:

1. Evaluate the accuracy – Static, dynamic, reproducibility, frequency (using RealSense/OptiTrack).
2. Is there any latency?
3. If accuracy is low, investigate internal feedback mechanism.
4. Do we need to calibrate every time? If so, develop a method for calibration. Each direction requires a separate calibration?
5. If accuracy is low, investigate other motors?
6. Can we replace the bread board with an Arduino board?

* Robot

1. Solve GitHub issues.

* Publication

Software journal, PMB?

Presentation at Image X.

Arrange a code review.

Rationale

The aim of this project is to integrate a 1D vertical motion platform with the existing 6DOF Robot Phantom to establish control of both robots under one software. The purpose of this goal is to find a way to further emulate a patient's behavior that is present during treatment such as breathing, which can make it much harder to track and target the tumor. The 6DOF robot is used to mimic the internal motion of the tumor while the 1D platform is intended to use the data collected from surface/external tracers on the patient to model and replicate the external motion of the tumor. Because of this, we can gather more accurate tumor motion data without the need of extra imaging for the patient which increases dosage.

1D Robot Platform

* + - 1. Install Visual Studio and install the necessary .NET packages for C#.
      2. Install Arduino IDE.
      3. Download and upload the Arduino code.
      4. Download the motion trace files from the github and copy one of the file paths.
      5. Paste the file path into the C# code and run the program.
      6. A console window will open asking you to enter the COM port.
      7. Enter the COM Port connected to the Arduino and the trace will load itself.

6DOF Robot – Setting Up.

1. Remove any already installed plates on the main install platform.
2. Align the moveable screw holes to line up with UR3 base platform.
3. Align the UR3 base platform in the desired position.
4. Get 4x \_\_\_ screws and place them in the screw holes.
5. Use a hex key to tighten all 4 screws.
6. Apply pressure in different directions to ensure the base plate is correctly mounted.
7. Now unpack and place the UR3 6DOF robot into position, lining up the 4 screw holes with the holes in the base plate.
8. Get 4x \_\_\_ screws and place them in each hole, tighten with a hex key.
9. Confirm stability of robot.
10. Connect robot controller unit to power.
11. Connect UR3 Robot to the robot controller unit.
12. Turn on the Robot controller unit and release the emergency stop button.
13. Wait for software to initialize.
14. Once asked, go to the initialization screen and begin initialization.
15. Hold the black button at the top of the tablet, this will release the joint brakes on the robot which will allow for the free movement and adjustment of the robot arm.
16. Physically move the robot to the desired start position,
17. Connect the Robot controller to the host PC via a CAT5 to USB connection.

6DOF Robot – Running the program and loading a motion trace.

1. If this is the first time or new connection to the PC, Go to Control Panel à Network and Internet à View network status and tasks à Click on the ethernet connection à Properties à Internet Protocol Version 4 (TCP/IPv4) à Use the following IP address, and set the IP address to ‘192.168.94.11’ or ’192.168.94.12’ if the former doesn’t work. NOTE if this IP doesn’t work then go into the Phantom Control program à Settings under 6D Motion and the IP address needed to establish connection to the robot is there.
2. Click on the subnet mask field which should automatically set to ‘255.255.255.0’ then click OK.
3. Click OK again and exit out of this window.
4. Now load out the Phantom Control Program and go into the Motion Control tab under 6D Motion.
5. Click on ‘Set Start Position’, the Home and Load Motion Traces button should go white.
6. Click on Home.
7. Click on Load Motion Traces and select the desired trace file. The trace file should load up and you can see the plot on the right-hand side
8. Click on Play Motion, the status bar should start progressing and the plot should start moving/tracing.

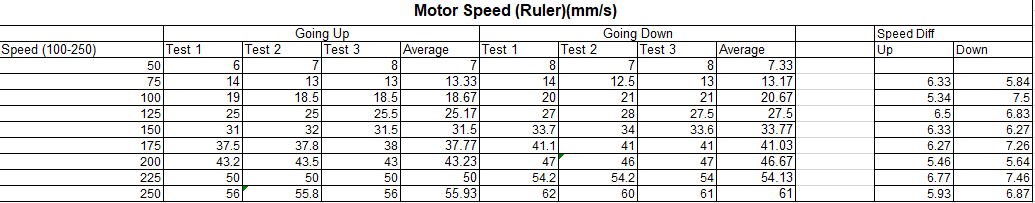
6DOF Robot – Disabling the send motion function

1. Open the csproj file for the 6DOF project.
2. Open the MotionControl (Design) file and view the UI buttons.
3. Double click on the ‘Start Motion’ button which will take you to the function code that will execute when the button is clicked in runtime.
4. Find and comment out the ‘’runMotion()’’ function. This will prevent the software from sending the motion data to the Robot.
5. Build and Run the program.
6. Select ‘Set Start Position’.
7. Select ‘Home’.
8. Select ‘Load Motion Trace’
9. Select the trace you want to run.
10. Select ‘Play Motion’.
11. Now confirm if the software will send the trace to the robot or not. If the trace doesn’t start and the Robot doesn’t move, then you have successfully disabled the send motion function.

# Documentation – How the robot moves

The basis of moving the 1D motion platform is the linear actuator motor, that is driven by a microcontroller; in this case, our build involves the use of Arduino to control the movement of the robot, as well as establishing a serial connection from the host PC to receive data to drive the robot. One of the ways to move the robot involves using a DC Motor Drive, which is a module that connects to both the actuator and Arduino, to control the speed of the actuator. This takes advantage of Arduino’s PWM, which is a technique for getting analog results with digital means. The Arduino’s PWM function can map 0% to 100% to the 8-bit value of 0 to 255; PWM itself applies digital control to create square waves (signals switched on and off) which will simulate voltages between 0 (off) and the full Vcc of the board (5V on the Arduino Uno). [Here](https://docs.arduino.cc/learn/microcontrollers/analog-output/) is the link to the Arduino documentation further explaining how PWM (Pulse Width Modulation) works.

Now we have a technique in simulating analog signals which is important because not only can we move the actuator up and down, but we can now control the speed using an input value of 0 to 255, where 255 is the max speed of the actuator. Now we need to find a way to input a motion trace file into a program which can send this data to move to actuator accurately. There were multiple ways to approach this, however we decided to create our own linear function, mapping the voltage-speed relationship of the motor. This involved using a ruler (Now using RealSense depth camera) to measure how far the actuator moved over a fixed period, with different voltages sent to the Arduino. The group working on this platform last year used a ruler to measure the movement during the static tests and compiled a table to note down the speed in mm/s for each input value from 0 to 255 with increments of 50. By doing this 3 times for each input value, we were able to calculate the average speed in mm/s for each of these inputs and create a linear equation to quantify the voltage-velocity relationship. The table below contains the results.



From these results, we were able to get two equations to model the velocity-voltage relationship, one for going up and one for going down. These equations could then be used in our code from the host PC to calculate the required voltage to be sent to the Arduino, from the given motion trace. Below would be a basic flow chart explaining this.

UP? Use the calculated velocity and input it into the equations to calculate the voltage required to move it up

Return Up bool = false

DOWN? Use the calculated velocity and input it into the equations to calculate the voltage required to move it up

Send voltage value to Arduino to allow for the motor to move

Calculate velocity of each point in time by taking the difference of displacements/time

Motion trace file containing time and displacement

Send Ack

The code was originally written in python but was later converted to C# to make it compatible with the 6DOF Robot software (Phantom Control) which has now been integrated. Below will be the C# functions used to carry out the tasks visualized in the flowchart above.

**This function loads the trace file and plots the input trace**

private void flatButton\_LoadTraces1D\_Click(object sender, EventArgs e)

{

// Clear the current plot in case a new file has been loaded

clearPlot("all");

// Initialise a new Open File Dialog, this will open a window for the user to select a .txt trace file

using (OpenFileDialog ofd = new OpenFileDialog() { Filter = "Text Documents|\*.txt", ValidateNames = true, Multiselect = false })

{

if (ofd.ShowDialog() == DialogResult.OK)

{

// Initiate a integer columnNum this will use the getColNumber function to check how many colum are in the trace file

int columnNum = getColNumber(ofd);

// For 1D motion traces there should only be two columns, Time and Displacement. If columnNum = 2 then continue to read the file

if(columnNum == 2)

{

// Assign the file path name to a string variable \_filePath

\_filePath = ofd.FileName;

// Display the file path in the text box - visual indicator

textBox\_filename.Text = System.IO.Path.GetFileName(ofd.FileName);

// Read the displacement values

List<double> displacementValues = ReadDisplacementValues(\_filePath);

// Shift the displacement values by the minimum value to ensure the new minimum is now 0

List<double> newdisplacementValues = ShiftDisplacementToZero(displacementValues);

// Plot the input trace

drawInputTrace1D(newdisplacementValues);

// Update button availability

if (\_filePath != null)

{

flatButton\_LoadTraces1D.Enabled = true;

flatButton\_PlayStopMotion.Enabled = true;

\_playstopmotionclicked = true;

}

}

// If the no. of columns are less or greater than 2, then the file is not compatible and the user should reselect.

if (columnNum > 2 || columnNum < 2)

{

UpdateStatusBarMessage.ShowStatusMessage("Error: Invaild format, format needed: [t Y]");

Logger.addToLogFile("Error: Invaild format, format needed: [t Y]");

MessageBox.Show("Error: Invaild input file format, format needed: [t Y]");

}

}

}

}

public string FilePath

{

get { return \_filePath; }

}

private string \_filePath;

static List<double> ReadDisplacementValues(string \_filePath)

{

List<double> displacementValues = new List<double>();

try

{

// Read all lines from the file

string[] lines = File.ReadAllLines(\_filePath);

// Process each line

foreach (string line in lines)

{

// Split the line by comma

string[] values = line.Split(' ');

// Ensure there are two values in the line

if (values.Length == 2)

{

// Parse only the displacement value

if (double.TryParse(values[1], out double displacement))

{

// Add the displacement value to the list

displacementValues.Add(displacement);

}

else

{

Console.WriteLine("Error parsing displacement.");

}

}

else

{

Console.WriteLine("Invalid data format in line: " + line);

}

}

}

catch (Exception ex)

{

Console.WriteLine("An error occurred: " + ex.Message);

}

// Return the list of displacement values

return displacementValues;

}

**This next function will convert the displacement values to velocity:**

static List<double> CalculateVelocityfromDisplacement(List<double> displacementValues)

{

List<double> velocityValues = new List<double>();

for (int i = 0; i < displacementValues.Count - 1; i++)

{

velocityValues.Add((displacementValues[i + 1] - displacementValues[i]) / 0.2);

}

return velocityValues;

}

Now this function takes in the velocity values and converts them to the output voltage required to be sent to the Arduino:

private static (double voltageValue, bool isGoingUp) CalculateVoltagePair(double v1)

{

// Initialise List of Voltage Values

double voltageValue = 0;

double speed = v1;

bool isGoingUp = false;

if (speed > 0)

{

voltageValue = (speed + 5.83) / 0.247;

isGoingUp = true;

}

else if (speed < 0)

{

voltageValue = Math.Abs((speed + 6.11) / 0.267);

isGoingUp = false;

}

else // speed == 0

{

voltageValue = 0;

}

return (voltageValue, isGoingUp);

}

We identified a problem however; we realized that the any voltage value less than 50 sent to the Arduino would lead to the actuator not moving, this is because the motor itself has a minimum speed for it to move, and a lot of the velocities calculated between each 200ms time interval from the original trace involved very small movements. We then decided to create a new function to calculate a delay. Currently, the voltage sent to the Arduino would move the motor at the voltage/speed for 200ms at a time, since the input traces had time intervals of 200ms. Now that we were unable to send speeds lower 6mm/s (equivalent to 50 as an input), we had to give the motor a higher voltage, but have it move for less time to achieve the same displacement. The function CalculateDelayValue is responsible for this.

static double CalculateDelayValue(double voltage2, double speed)

{

double delayValue = 200;

double displacement = Math.Abs(speed) \* 0.2;

double Voltage = 60;

if (voltage2 > 50)

{

delayValue = 200;

}

if (voltage2 < 50)

{

if (speed > 0)

{

delayValue = displacement / (Voltage \* 0.247 - 5.83) \* 1000;

}

if (speed == 0)

{

delayValue = 200;

}

if (speed < 0)

{

delayValue = displacement / (Math.Abs(Voltage \* 0.267 - 6.11)) \* 1000;

}

}

if (delayValue > 200)

{

Console.WriteLine("delay:" + delayValue + " displacement:" + displacement+" speed: "+speed);

}

return delayValue;

}

This uses the voltage-velocity relationship to create a new equation for the new delay and returns it as a double.

This last function will send the voltageValue, a bool for Up/Down and the final velocity value to the arduino, checking if a serial connection has been established:

. . .

if (serialPort.IsOpen == true)

{

DateTime now = DateTime.Now;

var result = CalculateVoltagePair(velocity);

var delayResult = CalculateDelayValue(result.voltageValue, velocity);

double newVoltage = (result.voltageValue < 50 && result.voltageValue != 0) ? 60 : result.voltageValue;

string voltageString = $"{newVoltage},{result.isGoingUp},{delayResult}";

serialPort.WriteLine(voltageString);

string data = serialPort.ReadLine();

var timepassed\_iteration = (now - previous);

var x = timepassed\_iteration.TotalMilliseconds;

Console.WriteLine($"Time:{x}Sent Data: {result}, {delayResult}, Received from Arduino: {data}");

if (serialPort != null || serialPort.IsOpen == true)

{

var timepassed = (200 - (DateTime.Now - now).Milliseconds);

var timeshift = (timepassed > 0) ? timepassed : 0;

Thread.Sleep(timeshift);

previous = now;

}

else

{

MessageBox.Show($"Arduino was disconnected! Please reboot the program and establish serial port connection!");

return;

}

. . .

The arduino code will then read the string sent by the arduino and separate each section of the string by the delimiter, in this case it was a ',’. This will allow it to assign the first part of the string to a voltage variable, the second part to a true/false bool and the third part to the delay value. These values will then be used to move the motor in this loop code:

. . .

if (up == true) {

analogWrite(RPWM, 0);

analogWrite(LPWM, voltage);

} else {

analogWrite(RPWM, voltage);

analogWrite(LPWM, 0);

}

delay(delay\_time);

. . .

Serial.println(receivedStr); // Send acknowledgment"