University of Arkansas – CSCE Department
Capstone II – Documentation – Spring 2023

AMBOTS Cooperative Robotic Arm 3D Printing

Stanley Van, Michael Darden, Cassandra Nelson, Alvaro Becares Fernandez

Table of Contents

Abstract

The manufacturing industry currently uses specialized machines and factories to produce products. This limits the flexibility of what can be produced out of one factory. New machines are often required for new products. AMBOTS is a company using existing or developing new technologies to break down manufacturing tasks into smaller and simpler ones. According to the AMBOTS website [1], “AMBOTS is an advanced manufacturing company with a focus on swarm 3D printing and assembly.” These can then be automated and coordinated by robots. However, these robots need a way to communicate with one another. Our goal is to design and implement a communication protocol for different third-party robotic arms to communicate with each other to cooperatively complete a task.

We will solve this problem by breaking it down into manageable steps. First, we will select a group of robotic arms to start with and learn how to operate these arms in ROS, MoveIt, and RViz. After this, we will create a program that can translate G-code files into the robotic arms' native languages. Along with the translation program, another will be needed to allow the separate robots to communicate with each other. Finally, we will use both previous programs to perform a cooperative print using two separate robotic arms. When this goal is achieved, the third-party robots will be able to communicate with each other, perform a task cooperatively, and complete a 3D print of a file they are given.

1.0 Problem

When a new product is designed, there is often a long and expensive production process. If one were to create a prototype of a car and work out all the design flaws, one then must deal with the cost and issues of finding a way to produce the car on a massive scale. The answer to that problem in today’s world is to design a factory around making that car. For several components of the car, specific machines have to be designed and built solely for that one car. This is an expensive and non-reusable solution that has become a massive challenge for almost all mass-produced products today. This also causes issues for some companies in the supply chain. It discriminates against the small guys. Because companies are so specialized on specific things, they have to rely on the services of others who may not want to provide their services as it may seem unprofitable to serve them. As co-founder and CTO of AMBOTS, Dr. Zhou put it, our civilization is built on manufacturing. Any production capability we lose is bad for our civilization. We cannot even reproduce the pyramids. We went to the moon and still have not been back.

Factories are built for the product. When demand for a product decreases, we lose the factory and the capability to produce the product. We cannot reproduce the sophisticated production process we have here on Earth on Mars. These problems lead to the overarching goal of AMBOTS being to create a general-purpose factory. A large obstacle to this goal is stationary and specific machines, and AMBOTS is the pioneer of the answer. In order to rid a factory of its need for these stationary and specific machinery, the production process is broken down to the assignments of specific tasks to different robots. When production changes are needed, the robots can be assigned different tasks and can move around accordingly to their new tasks. This means that there is a need for an open ecosystem software package that can give instructions to these kinds of robots and also support a wide variety of third-party robots. It would be

impractical to create a whole new network of robots for this, so instead, accommodating existing robots in the industry is a more feasible and economical goal.

2.0 Objective

The objective of this project is to perform cooperative 3D printing with Industrial Robotic Arms with ROS. This will be achieved by developing a universal printing interface such that it is possible to control different third-party robots by integrating them into our sponsor's platform and cooperate with other robots for manufacturing. We will do this by designing and implementing a communication protocol such that other third-party ROS-compatible robots can effectively talk with our robots over a local wireless network.

3.0 Background

3.1 Key Concepts

Swarm Manufacturing [2] is a new form of manufacturing developed for future factories. It is the employment of a swarm of different robots to manufacture products cooperatively on an open factory floor.

A 3D printer [3] is a machine where three-dimensional models are constructed by adding material together, typically layer by layer.

The Robot Operating System (ROS) [4] is a set of software libraries and tools that help researchers and developers build and reuse code between robotic applications. ROS 1 has support for real time code and embedded systems. Any code file that utilizes ROS is called a node. Nodes have three ways of communicating. The first way is the publisher subscriber method. The second way is through services. The third way is through actions.

MoveIt [5] is an open-source ROS package. Its basic task is to provide the necessary trajectories for robotic arms. This allows the robotic arms to move to the right locations. There are two main functions which are creating a plan and sending a plan. Known obstacles in the robot's environment can also be added so that they can be avoided.

RViz [6] is a ROS graphical interface. It helps developers visualize and test algorithms and design robots in a digital environment. This is important for not only efficiency but lowering cost and safety risks. Although initially Gazebo was going to be used as the simulation environment, RViz was chosen instead because of its ability to use markers to simulate extrusion.

Degrees of Freedom (DOF) [7] is the number of independent variables that define the possible positions or motions of a mechanical system in space. The number of degrees of freedom is equal to the total number of independent displacements or aspects of motion (translational or rotational).

Linux [8] is an open-source operating system. It comes in many different distributions. The specific distribution we will be using is Ubuntu 20.04. This is to allow the use of ROS 1.

Computer-aided design (CAD) [9] is a design of real-world objects where computers were used in their creation. This type of design lets engineers create precise and quality models and prototypes. There is a system of software that allows designers to simulate, analyze, and optimize their designs.

A G-code [10] file is a file that contains a series of instructions that 3D printers use to create a model in the real world. These instructions tell the printer where to move, how fast to move, and what path to follow.

A slicer [11] is software that takes a 3D object model and converts it to specific instructions for a 3D printer. The output of a slicer is a G-code file. The slicer interprets the 3D models and finds a path for the 3D printer so that it can put down layers of material that will be in the shape of the model.

Python [12] is an interpreted, object-oriented, and high-level programming language. Python files end with the "py" file extension. It is simplistic in its syntax and comes with a standard library of many useful functions and data structures.

The Universal Robots UR10 [13] is the largest robot in the Universal Robots collaborative series. It has a payload up to 10 kg. It is very easy to set up and provides an accuracy of about 0.1mm. It is ROS compatible and can be simulated in both RViz and Gazebo.

The Knova Gen3 Six Degrees of Freedom (6DOF) robotic arm [14] is a 6-axis robotic arm. It is ROS compatible and can be simulated in both RViz and Gazebo.

PyMesh [15] is a code base used for geometric processing. It can be used with both C++ and Python. We will be using it for splitting the object we are printing into what each robotic arm will be printing.

An STL [24] file is a file that describes an unstructured triangulated surface by their normal unit vectors and vertices of the triangles using the Cartesian coordinate system. These files describe the surface geometry of an object only. There are no colors, textures, or other properties that are described in the file. This file format is typically used for CAD software.

3.2 Related Work

The idea of swarm manufacturing is relatively new, and according to Additive Manufacturing [16], AMBOTS achieved the first end-to-end solution for cooperative 3d printing. Because of this there are few other equivalent accomplishments. Instead of looking at similar products, we can look at connected fields like both Additive Manufacturing and Swarm Robotics. We can look at these two concepts because when combined they create swarm 3D printing.

First, we will look at Additive Manufacturing. According to General Electric [17], Additive Manufacturing is the use of CAD software or 3D scanners to deposit material in precise shapes layer by layer. One company that utilizes this concept is 3D Systems [18]. The SLA 750 is one of their most recent developments. It is a "High-speed stereolithography solution for production manufacturing." What this means is it is a large end-to-end manufacturing machine. It uses dual lasers and photopolymer materials. Even with these improvements to a typical 3D printer, there is still the limitation of size. Because it is a single, enclosed machine, the product

produced must still fit inside the machine in the designated area. This differs from AMBOTS as when utilizing mobile robotic arms in swarm printing, size is not limited to inside one specific machine.

Next, we can look at Swarm Robotics. According to Scholarpedia [19], swarm robotics is the design of groups of robotics that operate without any interference of external infrastructure or centralized control. Robot swarms are self-organizing. One company that utilizes this concept is Unbox Robotics [20]. Unbox Robotics uses swarm intelligence to improve the throughput and sorting process. Although this is helpful in the manufacturing industry, it is not manufacturing the products themselves. It is instead just organizing them after the fact.

Finally, we will look at Rosotics [21]. Rosotics is the closest comparison to what AMBOTS is working on. While AMBOTS is working on swarm 3d printing for end-to-end manufacturing of a product, Rosotics is working on Rapid Induction Printing for metal additive manufacturing. According to an article from NASA's startup series [22], Rosotics is "a pioneer of swarm robotics". Although most of their work is hard to find specifics on, from this article we can assume that they use swarm intelligence in their approach. Even knowing this, they are mostly focusing on Rapid Induction Printing, rather than the standardization of communication between third party robotic arms as our project will be focusing on.

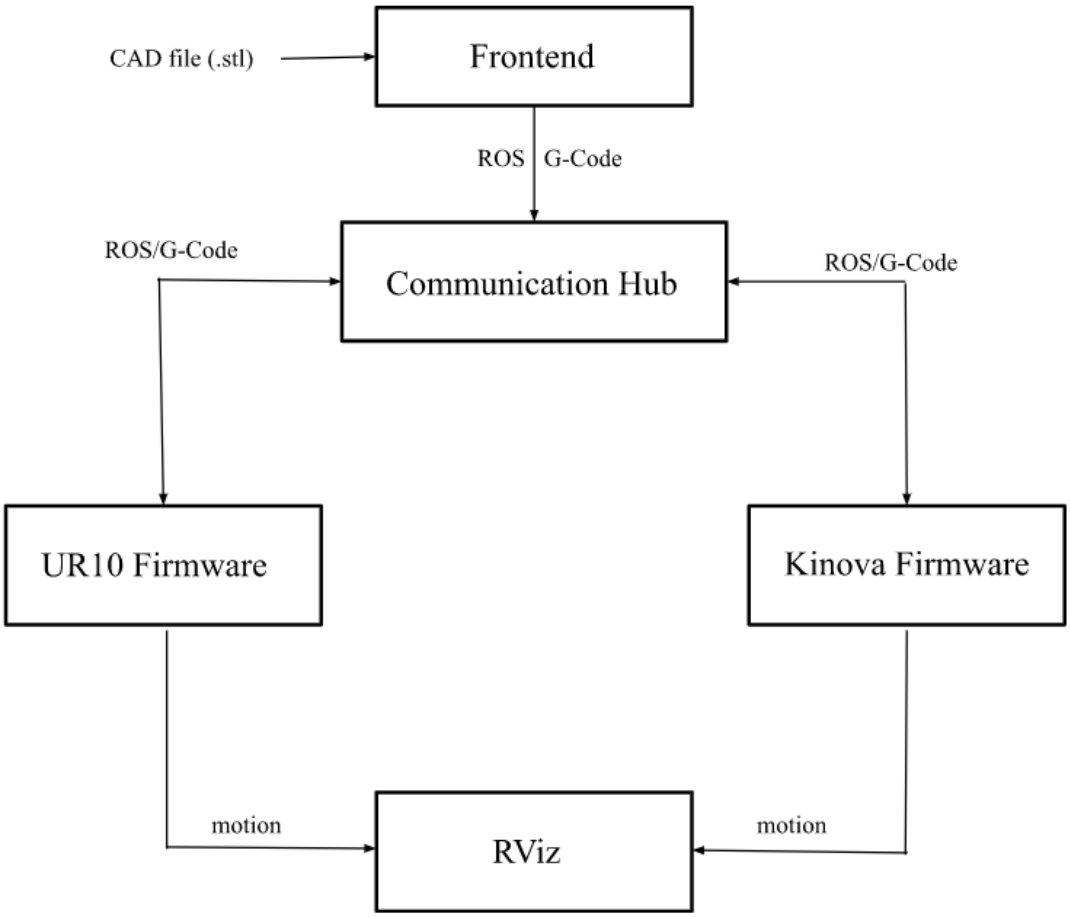
4.0 Approach/Design

4.1 Requirements and/or Use Cases and/or Design Goals

The requirements for this project will include the ability to input a CAD file (STL) to the machine and have two different robots simulate the 3D printing of the object in the CAD file. This code should be general and be able to work on multiple different brands of robotic arm. We are looking for universality. Initially there was a potential to add a slicer so that an object from CAD software can be directly inputted without the prior translation to G-code. We decided to use a third-party slicer to do this and have included it in our software package. We had the stretch goal of designing a slicer ourselves, but quickly realized this was not a reasonable goal to set for ourselves.

The main users of this end project will be mechanical engineers at AMBOTS. It will be controlled via command line interface and will only be able to be run through Ubuntu. Because of this, it will require the user to have some basic knowledge of Linux along with using a command line. This could allow for a wider range of users as they would not have to have as much background knowledge to use the application. The main limitation of this currently is the required system dependencies. Beyond needing a Linux operating system, ROS, MoveIt, the robotic arms you want to use, Pymesh, PrusaSlicer, and RViz all need to be installed and setup. We hope that our documentation (and potential bash files) can ease this process, but it all requires understanding how to download things via a command line.

4.2 Detailed Architecture



The overall structure of our project is shown above. Each module is explained below along with the current implementation. The design is divided up into three parts: front end, communication hub, and firmware. Each of which are meant to be able to be run on different machines. Communication between all three pieces of the software and the simulation environment are done via ROS topics.

4.2.1 Front End:

The front end is run via frontEnd.py. This is where the interaction with the user takes place. It takes STL files, divides the object in the STL file into 2 halves, slices the halves into G-Code, inserts any necessary custom “A” commands, and then sends the G-Code to the firmware for each robot. The user interacts with the front end through the command line. There is a basic command handler to allow the user to easily execute specific commands without memorizing them.

Basic command handler shown below:

```
Type the number of the command to execute the command. Always press the enter key to enter in the command/input:
        1) Start demo (This will select, slice, send, and start a print job to the arms)
        2) Slice an STL file
        3) Send G Code to the arms
        4) Start printing
        5) Quit
```

If number 1 is selected, the user will not be able to choose the file that is printed. Instead, a demo object, which is a cube from the “cube.stl” file, is used. However, if one wants to specify a specific STL file to slice and print, one has to execute commands number 2, 3, and 4. The “cube.stl” file is automatically cut into 2 haves. These halves are then sliced and saved into G-Code files using a slicer script. These G-Code files are sent to the arms. A start command is then sent to the Kinova arm to start printing.

When number 2 is selected, the user will be prompted to type the name of the STL file they want to be sliced. Although it was not required, the front end also has the capability to accept other CAD file formats such as OBJ and PLY. The initial CAD file is subdivided into two separate files for each robotic arm to print. This is done using the Python library, PyMesh. One of the many things PyMesh allows for is Boolean operations on 3D meshes. For our purposes there is a user uploaded file which is imported as a mesh along with an object used to “cut” the mesh. The user’s STL file may not be centered on the possible printing area. In order to ensure the mesh is centered, we iterate through the vertices of the mesh and find the minimum and maximum x values, the minimum and maximum z values, and the minimum y value. We then translate the mesh accordingly. The object used to cut the user’s mesh is just a large rectangular prism that takes up half of the possible printing area. The difference between the user uploaded object and the rectangular prism is then computed. This produces one half of the original object. The intersection between the user uploaded mesh and the rectangular prism is also computed to produce the other half of the original object. These two meshes are then exported as STL files. These STL files are then converted into G-code using PrusaSlicer (an open-source slicer). PrusaSlicer has a CLI that we have used to create a bash file to automate this process.

The front end then injects certain commands at certain locations based on the robot ID in the G-Code file names. For the Kinova arm, we inject a command before all of the G-Code to first move the UR10 arm out of the way. The Kinova arm then starts printing. At the end of the Kinova arm's G-Code, we inject commands to move itself out of the way and tell the UR10 arm to start its print job. For the UR10, the front end injects commands at the end of its G-Code chunk telling it to move out of the way, resume the Kinova arm, and then pause itself.

When the user selects number 3, the two G-code files are then sent to the Communication Hub via the ROS publisher subscriber method. The front end creates a string with the first line being the A5 command and the destination being the robot ID in the G-Code file name that is currently being sent. The G-Code file contents are appended to that string, and the string is sent to the communication hub through the communication hub ROS topic.

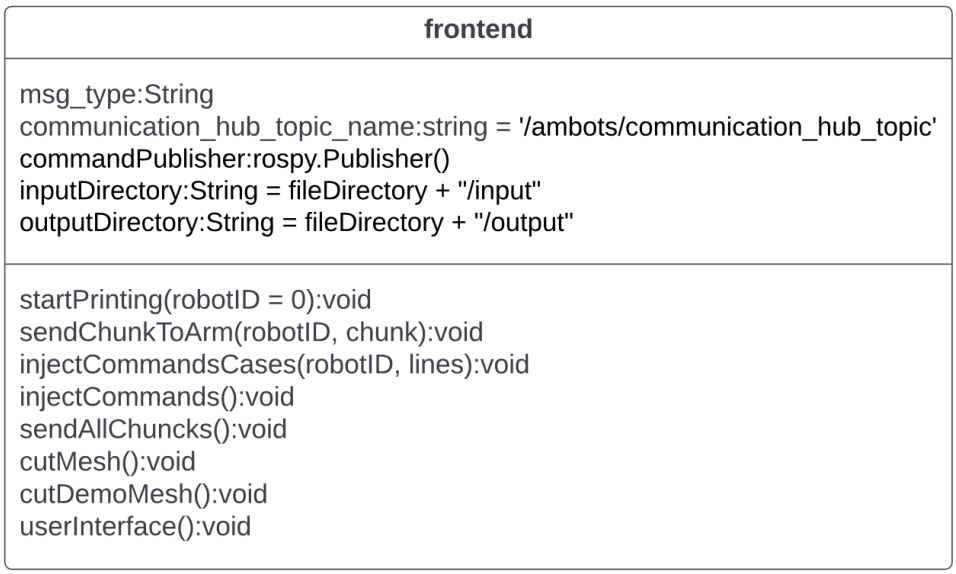
When number 4 is selected, the front end will send the A1 command to tell the Kinova arm to start printing. The number 5 command will exit out of the front end.

The front end is implemented as a class in the “frontEnd.py” file. The constructor has parameters for an input directory, an output directory, and the communication hub topic string. The constructor initializes a ROS node named “frontEnd”. A topic publisher is also initialized,

and the constructor waits for the communication hub to subscribe to the topic so that no messages to the communication will be lost.

The “startPrinting” method sends the A1 command to the specified arm with the default arm being robot 0 which is the Kinova arm. The “sendChunkToArm” sends the A5 command to the specified arm with the G-Code data appended and separated from the A5 command by a newline character. The “injectCommandsCases” method will inject certain commands to certain location depending on the provided robot ID. The “injectCommands” method will go through the output directory and inject commands into the G-Code files. The “sendAllChunks” method will go through the output directory and send each file to the arm specified by the robot ID in the file name. The “cutMesh” method will cut a given STL file into 2 haves. These halves are saved into the output directory. They are then sliced and saved into G-Code files in the output directory. The “cutDemoMesh” method does the same functionality as the “cutMesh” method, but it will always cut and slice the “cube.stl” file only. The “slice” method calls the slicer script with a half STL file as the input file to be sliced and a “config.ini” file as the input file to set the slicing configurations. The output files are written as G-Code files to the output directory.

4.2.2 Front End UML (Unified Modeling Language) Diagram:



4.2.3 Extracting G-Code:

The "extractGCode.py" file contains a collection of functions that open, read, and extract G-code commands. It uses a regular expression to detect lines that start with "G". It also ensures that the lines have no parameters or have parameters with values attached to ensure we capture only valid G-code commands. There is a set of parameters to process the captured G-code commands. The "extract" function takes in a parameter that is a string. The regular expression then finds all of the matches and puts them into an array. The function then loops over the matches. In the loop, we split each match by whitespace. We then create a hash map to assign the parameters from our parameter set values that are present in the matched line that is of the iteration. This hash map is then added to an array that will contain all the hash maps of all the matches. This array is then returned as the extracted G-code commands.

There is an "extractFile" function that takes in a filename as a parameter. This function will open the file, call the "extract" function with the contents of the file as the parameter to the function, and then return what the "extract" function returns.

There are the “extractRaw” and “extractFileRaw” functions that function identically to their “non-raw” versions. These “raw” functions do not put each extracted command into a hash map of parameters. They return an array of the lines of the commands as they appear in the G-Code file or string. Thus, they are labeled as “raw”.

There are default axes conversion functions named "defaultXConversion", "defaultYConversion", and "defaultZConversion". These functions take in a floating-point number and return that number but converted in the space of a robotic arm. These functions are meant to serve as placeholders. The specific implementations of these conversion functions are in the generic Firmware class where values specific to a robotic arm are used. These functions take in the floating-point number, normalize it, and then convert the normalized number to the space of a robotic arm. They are called in the "convertPositions" function.

The "convertPositions" function takes in the G-code command array, an x axis conversion function, a y axis conversion function, and a z axis conversion function as parameters. These parameters are named "arr", "xConvert", "yConvert", and "zConvert" respectively. The function loops through the array and calls the correct conversion function for the axes that are present in the command. The values of the axes are replaced, and the array is then returned.

There is also a function called "getCoordinate". It takes in a single G-code command that was extracted and returns the x, y, and z values as a Python tuple. If a value of an axis is not present, the value of None is given for that axis.

4.2.4 Communication Hub:

The communication hub serves as the director of information for the system. The hub handles sending data back and forth from the robots' firmware. Direct communication between different brands of robotic arms is not currently possible. To solve this issue, any communication between robots will also go through the communication hub. Communication for both the frontend and the firmware utilizes the ROS publisher subscriber method. The topic the communication hub uses to receive messages is “/ambots/communication_hub_topic”.

To send instructions and data to and from the arms, we have created a set of G-Code like commands called the A series commands. Below is a table of what the commands are and their functionality.

A Series Command Table:

Command	Parameters	Description
A1	Rx	Start the specified arm x
A2	Rx	Pause the specified arm x

A3	Rx	Send the arm's current position to the specified arm x
A4	Rx	Clear the chunk data of the specified arm x
A5	Rx\ndata...	Send chunk data of the specified arm x. The first line is the A command. A newline character then separates the G-Code data from the command.
A6	Rx	Resume the specified arm x
A7	Rx Xa Yb Zc	Move the specified arm x to the specified position (a,b,c)

The communication hub is implemented as a class in the “communicationHub.py” file. The constructor has parameters for the communication hub topic string and a Boolean for enabling debug mode. The constructor initializes a ROS node named “communicationHub”. A topic subscriber is also initialized.

The method “startArm” corresponds to the A1 command. Method “resumeArm” corresponds to the A6 command. Method “pauseArm” corresponds to the A2 command. Method “sendStateData” corresponds to the A3 command. Method “clearChunkData” corresponds to the A4 command. Method “sendChunk” corresponds to the A5 command. Method “sendCoordinates” corresponds to the A7 command. The methods “startArm”, “resumeArm”, “pauseArm”, “sendStateData”, “clearChunkData”, “sendChunk”, and “sendCoordinates” are there to allow any future overloading for more custom functionality. Because of how the pause state is implemented in the Firmware class, an explicit resume command is needed.

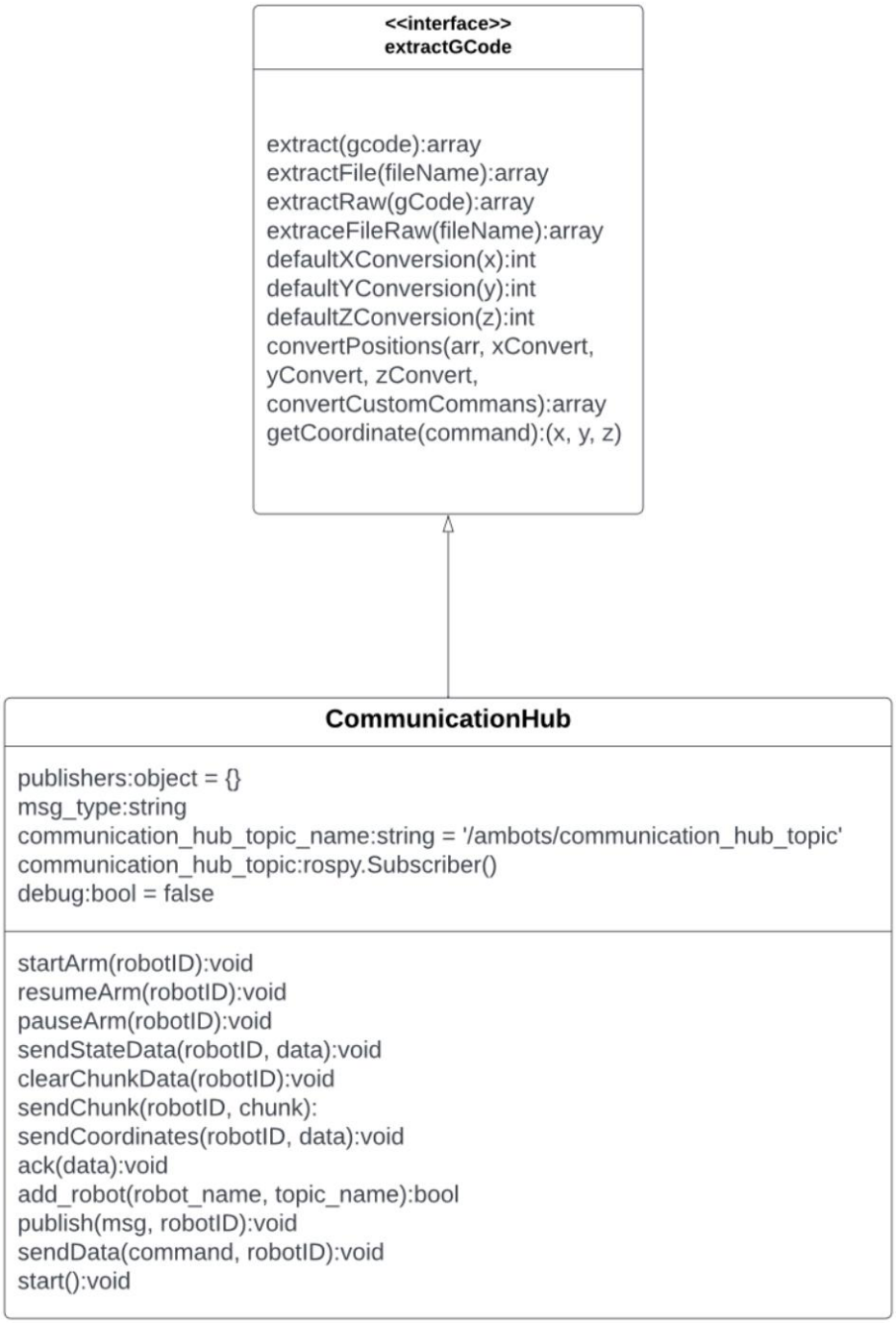
The “add_robot” method, associates a robot ID to a topic so that the communication hub will know where to send a message. It creates a topic publisher to be able to send messages through that topic. This method waits for the corresponding arm to subscribe to the topic so that no messages to the communication will be lost.

The “publish” method sends a message to a specified robot using the ID of the robot. The “sendData” method calls the “publish” method but also prints the command it is sending if debug mode is enabled.

The “start” method will keep the node alive until the user terminates the program. This is done so that the communication hub will keep running and not terminate after initialization.

The main function in the “communicationHub.py” file creates a CommunicationHub instance. The robot ID 0 is then associated to the topic “/ambots/kinova_topic”. The robot ID 1 is associated to the topic “/ambots/ur_topic”. The “start” method is then called to keep the node alive.

4.2.5 Communication Hub UML (Unified Modeling Language) Diagram:



4.2.6 Firmware:

There is a generic firmware class named "firmware" that implements the movement and printing of robotic arms. The constructor for this class take in parameters named "name", "arm_group_name", "tfPrefix", "xGcodeMax", "yGcodeMax", "zGcodeMax", "xGcodeMin", "yGcodeMin", "zGcodeMin", "xArmMax", "xArmMin", "yArmMax", "yArmMin", "zArmMax",

"zArmMin", "xOffset", "yOffset", "zOffset", "filamentDiameter", "maxVelocity", "xConversion", "yConversion", "zConversion", "tolerance", "filamentColor", "debug", “topic”, “communication_hub_topic”, “robotID”, “eef_step”, “jump_threshold”, “filamentXOffset”, “filamentYOffset", “filamentZOffset”, and “attachCollisionObjects". If the conversion functions are not given, an instance of this class will use the defined conversion functions within this class. The ROS MoveIt commander is initialized with the system arguments that are passed to this class. A ROS node is created with the name concatenated with the string "Firmware". The "marker_pub" attribute is assigned as a ROS publisher to publish an array of markers for the simulation of 3D printing. The "marker_array" attribute is assigned to an instance of the ROS MarkerArray class. A counter for markers is created and set to zero for use as the IDs of the markers in the marker array. There is a flag for when the arm is currently printing which is called "isPrinting". A ROS transform buffer and transform listener is instantiated to get poses after they have been transformed by the transform specified in the "bothArmsRvizLaunch.launch" file. The "robot" attribute is assigned to an instance of the ROS RobotCommander to control the arm's description. The "scene" attribute is assigned to the ROS PlanningSceneInterface to plan trajectories. The "arm_group" attribute is assigned to the ROS MoveGroupCommander to execute trajectories and obtain poses of the arm as it is in the simulation. The acceleration of the arm is set to the maximum value. The arm is then made to go to the home position. The coordinates of that position are then saved in the "homePosition" attribute. The end effector link is saved as an attribute with the "tfPrefix" parameter appended to the left side of the end effector link string name. The base link is saved as an attribute with the "tfPrefix" parameter appended to the left side of the base link string name. The debug Boolean will disable the feed rate function and set the velocity of the arm to the maximum amount. This is to speed up the simulation for testing. The “state” attribute saves the arm’s last x, y, and z position so that the next arm can continue where it has left off. The “gCodeChunk” attribute saves the G-Code chunk sent to the arm. The “extruder_name” attribute is the name of the collision box attached to the end of the arm. The “started” attribute is a Boolean that indicates whether the arm has started a print job. The “paused” attribute is a Boolean that indicates whether the arm is in the pause state.

There is a "reach_named_position" method that takes in a string that is the name of a known position. The arm then plans and executes a movement to that position.

The "reach_cartesian_pose" method takes in a pose as a parameter. It first sets the tolerance of the movement. If the arm's "isPrinting" flag is true, a constraint box will be made. This is to force the arm to move in a straight line from one point to another. The pose for the constraint box is also created to make the box lie along the line that connects the current coordinate of the arm to the one that is in the given pose. An orientation constraint is then made to keep the arm in the same orientation as the arm moves along that path. If the debug flag is set, a visual box of where the arm is supposed to move along is created and added to the marker array. The trajectory is then planned using interpolation. The arm then executes the movement. When the movement is complete, any debugging markers are removed if the arm's debug flag is true.

The "getCurrentTransformedPose" method returns the current pose after it has been transformed by the specified transform in the "bothArmsRvizLaunch.launch" file.

The "startExtruderCallback" method is a method that is meant to be overloaded to integrate an extruder. This method is for starting the extrusion of filament. It takes in an extrusion amount as a parameter.

The "stopExtruderCallback" method is a method that is meant to be overloaded to integrate an extruder. This method is for stopping the extrusion of filament.

The “moveAndExtrude” method is a method that starts the extruder, moves the arm, and then stops the extruder.

The “get_planning_frame” method gets the planning frame of the arm with the “tfPrefix” attribute appended to the front of the name of the planning frame.

The "printMarkers" method has the parameters "prevX", "prevY", "prevZ", "nextX, "nextY", and "nextZ". They stand for the previous x, y, z coordinates and the next x, y, and z coordinates respectively. They have a default value of None. If they are None, they are assigned the current value of their respective coordinate. This method will immediately return if the arm is not printing. A start and end points are created using the given coordinates. A cylinder marker with the coordinate of the midpoint of the start and end points is created. The marker is oriented to lie along the line that connects the two points. The colors and alpha value of the filament are then applied to the marker. The marker is then added to the array of markers, and the updated array of markers is published to be reflected in the RViz simulation.

The "goToCoordinate" method has the parameters "homepose", "abosolutePosition", "x", "y", and "z". The "homepose" parameter is set to the instance's "homePosition" attribute if one is not given. The "abosolutePosition" parameter specifies whether or not to treat the coordinates as absolute or relative coordinates. The "x", "y", and "z" parameters are the coordinate the arm will move to. If the coordinates are absolute, the coordinates will be converted to the arm space using the "homepose" parameter. If the coordinates are relative, the coordinates will be added to the arm's current coordinates. The pose is then passed to the "reach_cartesian_pose" method.

The "setFeedRate" method takes in a parameter "f" which is a floating-point number. It assigns the velocity of the arm so that the arm can print at the correct speed. It is normalized and converted into the arm's velocity space. If debug mode is enabled, the feed rate is always set to the maximum value.

There is a method called "getCoordinate". It takes in a single G-code command and returns the x, y, and z values as a Python tuple. If a value of an axis is not present, the value of None is given for that axis.

There are custom x, y, and z conversion methods that are defined. They are called "xConversion", "yConversion", and "zConversion" respectively. The reason for defining these methods is that these methods will be the functions that are passed into the "convertPositions" function to correctly convert the G-code Euclidean coordinates into the arm's coordinate space.

The “buildCoordinateString” method constructs and returns an A series string with the x, y, and z parameters.

The “updateLastState” method updates the last position of any of the arm’s axes.

The “communicationHandler” method checks whether the given command is an A series command. It returns true if it is and false if it is not. It waits for the communication hub to listen to the topic so that no messages are lost. For all of the commands, the arm will check if it is for

this arm and execute the appropriate functions. If it is for another arm, it will forward the command to the communication hub. If the command is A1, it will start a new thread and then call the “executeGCode” method to start printing. For the A2 command, the “paused” attribute is set to true. The A3 command moves the arm to the specified coordinates and calls the “updateLastState” method. If the command is for another arm, the “buildCoordinateString” method is called to build the A series command with wanted parameters to send to the specified arm. The A4 command sets the “gCodeChunk” attribute to the empty string. The A5 command sets the “gCodeChunk” attribute to the value of the “data” parameter that was passed to the “communicationHandler” method. The A6 command sets the “paused” attribute to false. The A7 command moves the arm to the specified coordinates and calls the “updateLastState” method. If the command is for another arm, the “buildCoordinateString” method is called to build the A series command with wanted parameters to send to the specified arm.

The “handleMessage” method is the callback function for when the arm receives a command. It separates the A series command from the data and then calls the “communicationHandler” method.

The Firmware class has an "executeGCode" method. This method first converts the G-code commands in the “gCodeChunk” attribute by calling the “extract” function and then the "convertPositions" function from the "extractGCode.py" file. It then loops through each converted command. The first step it does in each loop iteration is to pass the command to the “communicationHandler” method. If it returns true, then the rest of the movement code is skipped. If it returns false, then the loop will get the x, y, and z values from the command. It then detects the feed rate parameter and sets the instance's feed rate if one is found. If the command matches the criteria of it being an extrusion command, the "isPrinting" attribute is set to true, otherwise it is set to false. The current pose is then recorded. It then swaps the x and y values because the G-code x and y axes are swapped in relation to the space of the arm that will use this generic firmware class. The "goToCoordinate" method is then called with the x, y, and z values. After the arm has moved, the "printMarkers" method is called to place a marker that represents the extrusion. The parameters passed to that method are the Euclidean coordinates of the previous pose that was recorded. After the check for whether the command was an A series command, it will enter the pause state by entering a loop if the “pause” attribute is set to true. The arm will leave the pause state when the “pause” attribute is set to false.

The “start” method will keep the node alive until the user terminates the program. This is done so that the communication hub will keep running and not terminate after initialization.

The “attach_extruder” method creates a box with collision enabled. The box is added to the planning scene. After that, it is attached to the end of the arm so that the arm can simulate movement with an extruder attached.

The “attach_ground” method creates a large box underneath the arm. The box is added to the planning scene, so that the arm can plan movement without moving into the ground.

4.2.7 Kinova Firmware:

The KinovaFirmware class is a subclass of the Firmware class. Thus, it inherits the methods and attributes of the Firmware class. The KinovaFirmware class specifies the name, arm_group_name, tfPrefix, xGcodeMax, yGcodeMax, zGcodeMax, xArmMin, yArmMin,

zArmMin, zOffset, filamentDiameter, maxVelocity, and filamentColor attributes. For the Kinova 6DOF arm, the attributes are set to "kinova" for the name attribute, "arm" for the arm_group_name attribute, "kinova" for the tfPrefix attribute, 891 for the xGcodeMax attribute, 891 for the yGcodeMax attribute, 891 for the zGcodeMax attribute, 0 for the xArmMin attribute, 0 for the yArmMin attribute, 0 for the zArmMin attribute, 0.004 for the filamentDiameter attribute, 500 for the maxVelocity attribute, and a maximum value for the red color and alpha value for the filamentColor attribute. Another attribute is added to the class specifically for the Kinova arm. The degrees_of_freedom attribute is set to the integer value of 6. This is because we are using the Kinova 6DOF arm which has 6 degrees of freedom. The z offset attribute is set to -0.5. The x offset is set to -0.6. The y offset is set to -0.1. These are manually chosen to align the print area with the center of the Kinova and UR10 arms. The robot ID is set to 0. The “eef_step” aprameter is set to 0.05 meters for fine interpolation.

The "name" attribute is set to "kinova" so that the firmware ROS node will spawn with a name that indicates it is the node tied to the Kinova arm. The arm group name for the Kinova arm is named "arm" which is the reason why the attribute "arm_group_name" is set to "arm". The transform prefix in the launch file for the Kinova arm is set to "kinova" which is why the "tfPrefix" attribute is set to "kinova". Because the Kinova 6DOF arm has a maximum reach of 891 mm, the maximum G-code values for all 3 axes are all set to the integer 891. Because the Kinova arm has a full range of movement from a floating-point mapping of 1.0 to -1.0, the minimum Kinova arm coordinate values for all 3 axes are all set to the integer 0. Currently, we are experimenting with normal 3D printing filament in the simulation. Thus, we are using 4 mm or a float value of 0.004 for the "filamentDiameter" attribute. Because the Kinova 6DOF arm has a maximum speed of 500 mm/s, the "maxVelocity" attribute is set to the integer value 500. We are distinguishing both of the robotic arms' filament by their color. Thus, the Kinova arm is assigned the color red for the "filamentColor" attribute in the simulation.

The KinovaFirmware class only has a constructor which has the parameters "degrees_of_freedom" and "debug". The "degrees_of_freedom" parameter has a default value of 6, and the "debug" parameter has a default value of false. For the development and simulation of the arms, we specify the "debug" parameter as true.

The kinovaFirmware Python file has a main function which instantiates a KinovaFirmware object with debug mode on. It then reads a file named "cube_kinova.gcode" in the same directory as the kinovaFirmware Python file and executes the G-code commands within that file.

4.2.8 UR10 Firmware:

The URFirmware class is a subclass of the Firmware class. Thus, it inherits the methods and attributes of the Firmware class. The URFirmware class specifies the name, arm_group_name, tfPrefix, xGcodeMax, yGcodeMax, zGcodeMax, xGcodeMin, yGcodeMin, xArmMin, yArmMin, zArmMin, xArmMin, yArmMin, zArmMin, zOffset, filamentDiameter, maxVelocity, and filamentColor attributes. For the UR10 arm, the attributes are set to "ur" for the name attribute, "manipulator" for the arm_group_name attribute, "ur" for the tfPrefix attribute, 0 for the xGcodeMax attribute, 0 for the yGcodeMax attribute, 1200 for the zGcodeMax attribute, 1200 for the xGcodeMin attribute, 1200 for the yGcodeMin attribute, 2 for the xArmMin attribute, 1 for the yArmMin attribute, -5 for the zArmMin attribute, 6 for the

xArmMin attribute 6, 5 for the yArmMin attribute, -1 for the zArmMin attribute, 0.004 for the filamentDiameter attribute, 1000 for the maxVelocity attribute, and a maximum value for the green color and alpha value for the filamentColor attribute. The z offset attribute is set to 0.2475. The x offset is set to -3.415. The y offset is set to -0.55. These are manually chosen to align the print area with the Kinova print area. The robot ID is set to 1. The “eef_step” aprameter is set to 1 meter for interpolation that will not cause the arm to fail at planning.

The "name" attribute is set to "ur" so that the firmware ROS node will spawn with a name that indicates it is the node tied to the UR10 arm. The arm group name for the UR10 arm is named "manipulator" which is the reason why the attribute "arm_group_name" is set to "manipulator". The transform prefix in the launch file for the UR10 arm is set to "ur" which is why the "tfPrefix" attribute is set to "ur".

The UR10 arm has a maximum reach of 1300 mm. Because of our current and limited slicer settings, the maximum G-code values for all 3 axes are 1200, which is the closest value our slicer can handle without complete custom settings. This limits the range of the G-code value ranges from 0 to 1200. The UR10 arm's position in the simulation environment causes some of the axes of the G-code with respect to the arm to be flipped. This is the cause for setting the attributes xGcodeMax and yGcodeMax to zero instead of 1200 and for setting the attributes xGcodeMin and yGcodeMin to be set to 1200 instead of zero. The z axis for the G-code is unaffected, and thus, the zGcodeMax attribute is set to 1200. The UR10 arm has a range of movement on the x axis from a floating-point mapping of 0.0 to 12.0. We only use a limited section of this which is why the xArmMax attribute is set to 2, and the xArmMin attribute is set to 6. The UR10 arm has a range of movement on the y axis from a floating-point mapping of -6 to 6. We only use a limited section of this which is why the yArmMax attribute is set to 1, and the yArmMin attribute is set to 5. The UR10 arm has a range of movement on the z axis from a floating-point mapping of -6 to 6. We only use a limited section of this which is why the zArmMax attribute is set to -5, and the yArmMin attribute is set to -1. The z axis in the UR10 arm space is flipped which is the reason for the negative signs. Currently, we are experimenting with normal 3D printing filament in the simulation. Thus, we are using 4 mm or a float value of 0.004 for the "filamentDiameter" attribute. Because the UR10 arm has a maximum speed of 1000 mm/s, the "maxVelocity" attribute is set to the integer value 1000. We are distinguishing both of the robotic arms' filament by their color. Thus, the UR10 arm is assigned the color green for the "filamentColor" attribute in the simulation.

The URFirmware class only has a constructor which has the parameter "debug". The "debug" parameter has a default value of false. For the development and simulation of the arms, we specify the "debug" parameter as true.

The urFirmware Python file has a main function which instantiates a URFirmware object with debug mode on. It then reads a file named "cube_ur.gcode" in the same directory as the urFirmware Python file and executes the G-code commands within that file.

4.2.9 Kinova Firmware UML (Unified Modeling Language) Diagram:

4.2.10 UR Firmware UML (Unified Modeling Language) Diagram:

4.2.11 RViz:

For simulation in RViz, we have launch files to spawn the arms and execute arm movements. To get both of our chosen arms in the same RViz environment, we have a launch file named “bothArmsRvizLaunch.launch”. In it there is a namespace called "ambots" that separates our package from all of the other packages on the system our software will run on. We declare the arguments “use_gui”, “arm”, “dof”, “vision”, and “sim” for use with the Kinova arm's namespace section.

There are two sub-namespaces. One is called "kinova" for the Kinova 6DOF arm, and the other is called "ur" for the UR10 arm. Inside each of the sub-namespaces, we set their "tf_prefix" parameter to their respective names which are "kinova" and "ur". There is a static transform publisher for both of them. For the Kinova arm, the transform does nothing. For the UR10 arm, the transform moves it along the x axis for 1.5 meters and rotates it using a quaternion so that it is facing backwards in the direction it was moved along the x axis. This causes the UR10 to be placed in the correct location facing the correct direction, but it will be upside down. The quaternion will also rotate the arm around the x axis so that it is facing with the right side up. Then each sub-namespace finds the model file, which is an xacro file, for each arm and loads it as the "robot_description" parameter. The joint state publisher nodes for each arm are loaded as well as the robot state publisher nodes. These nodes handle the state and pose of the arms. At the end of the sub-namespaces is the loading of the "move_group.launch" file. This is what lets the firmware for each arm control the simulation arms. Outside of the sub-namespaces and at the end of the launch file is the RViz node that loads the RViz simulation environment window. The "kinematics.yaml" files for each arm are called to ensure the arms will have movement in the environment.

For launching the firmware of the Kinova arm, there is the "kinovaFirmware.launch" file. This file sets the namespace for the firmware as "ambots/kinova". The reason for the inclusion of "ambots" is that there is a global namespace of "ambots" in the “bothArmsRvizLaunch.launch” file. The node "kinovaFirmware" is then spawned which executes the "kinovaFirmware.py" file. The namespace of this launch file is also passed to the "kinovaFirmware.py" file.

For launching the firmware of the UR10 arm, there is the "urFirmware.launch" file. This file sets the namespace for the firmware as "ambots/ur". The reason for the inclusion of "ambots" is that there is a global namespace of "ambots" in the “bothArmsRvizLaunch.launch” file. The node "urFirmware" is then spawned which executes the "urFirmware.py" file. The namespace of this launch file is also passed to the "urFirmware.py" file.

The currently included G-code files in our repository have zero infill. This is to speed up the testing of the arm movements and the simulation of printing.

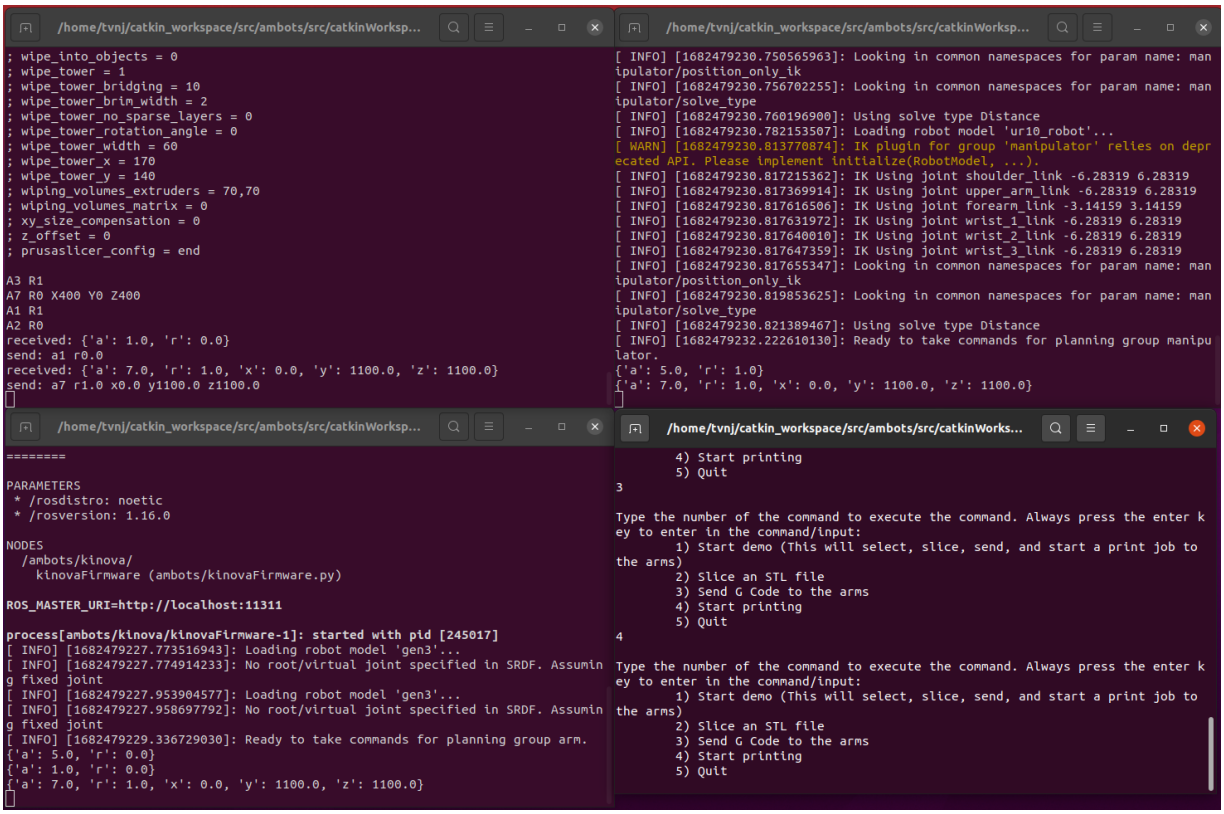
To visualize the arms in RViz. We used the RobotModel display type, which applies the transforms found in the “bothArmsRvizLaunch.launch” file, to represent the arms in the simulation. There are also the PlanningScene display types to display the boxes that represent the extruders attached to the arms. Because the Kinova arm has no transform applied, we left the PlanningScene display for it on to show the extruder visualization box. Since the PlanningScene display type does not apply the transform we used to set the arms at a distance apart, we left the PlanningScene display for the UR10 arm off since it would be distracting to see a detached extruder visualization box moving about.

4.2.12 Testing:

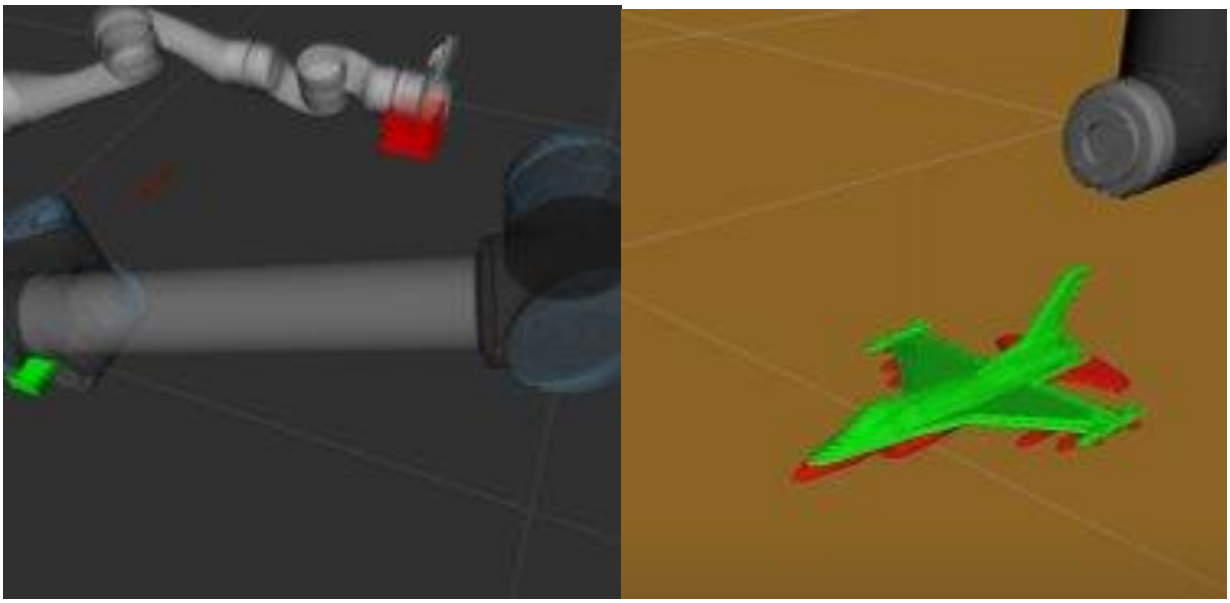
To test and debug the software, a debug mode has been implemented for the firmware and communication hub classes. For the firmware, the debug mode will set the velocity of the arm inheriting the class to the maximum value. Any A series commands the arm encounters will be printed on the terminal. For the communication hub, any A series commands the arm encounters will be printed on the terminal as well.

In the “tests” folder of the “src” folder of the repository, we have code, scripts, and launch files to test our various pieces of software, RViz settings, arm movement, and slicing. For integration testing, we used the debug mode of the firmware and communication hub classes to ensure the correct functionality between our pieces of software.

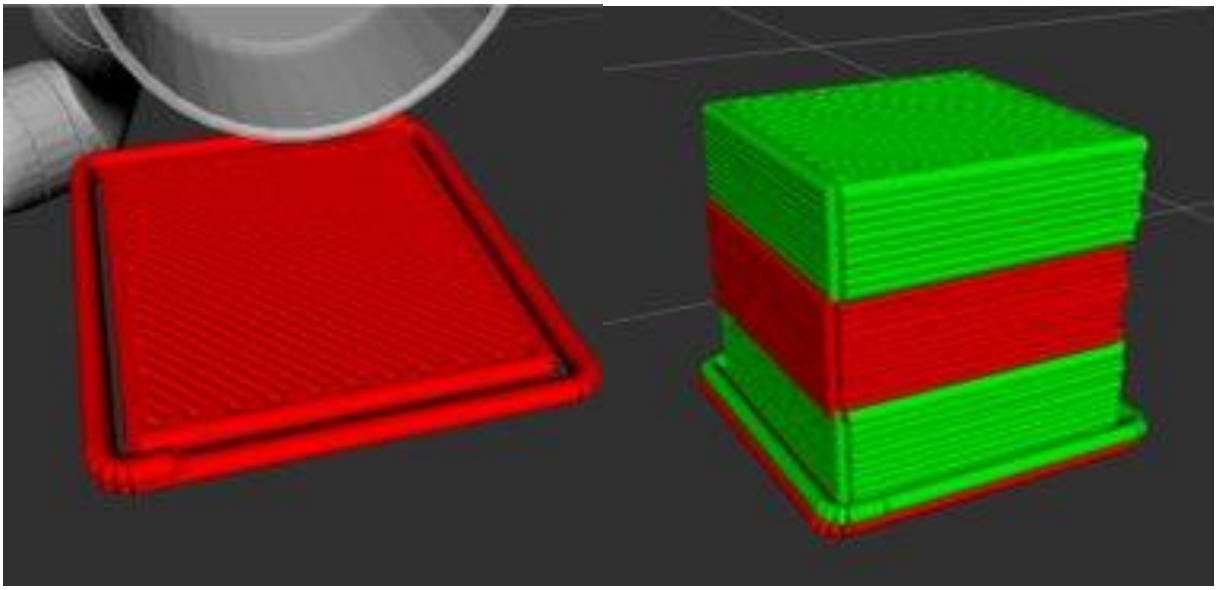
Testing of Communication in the Whole System Using Debug Mode:



4.2.13 Final Results:



The image above on the left shows both arms printing in the same RViz instance. The image above and on the right shows the finished print of a jet.



The image above on the left shows the beginning of printing a cube. The image above on the right shows the finished print of a cube.

4.2.14 Lessons Learned:

One of the main lessons learned so far is to ask specific questions from the sponsor as soon as possible. It is important to get a clear understanding of the minimum end goal, reasonable end goal, and stretch end goal. It is also important to know if they already have a structure in mind, or even specific packages that they already use that would be good to use in the project. Overall being overly communicative is imperative to the success of the project.

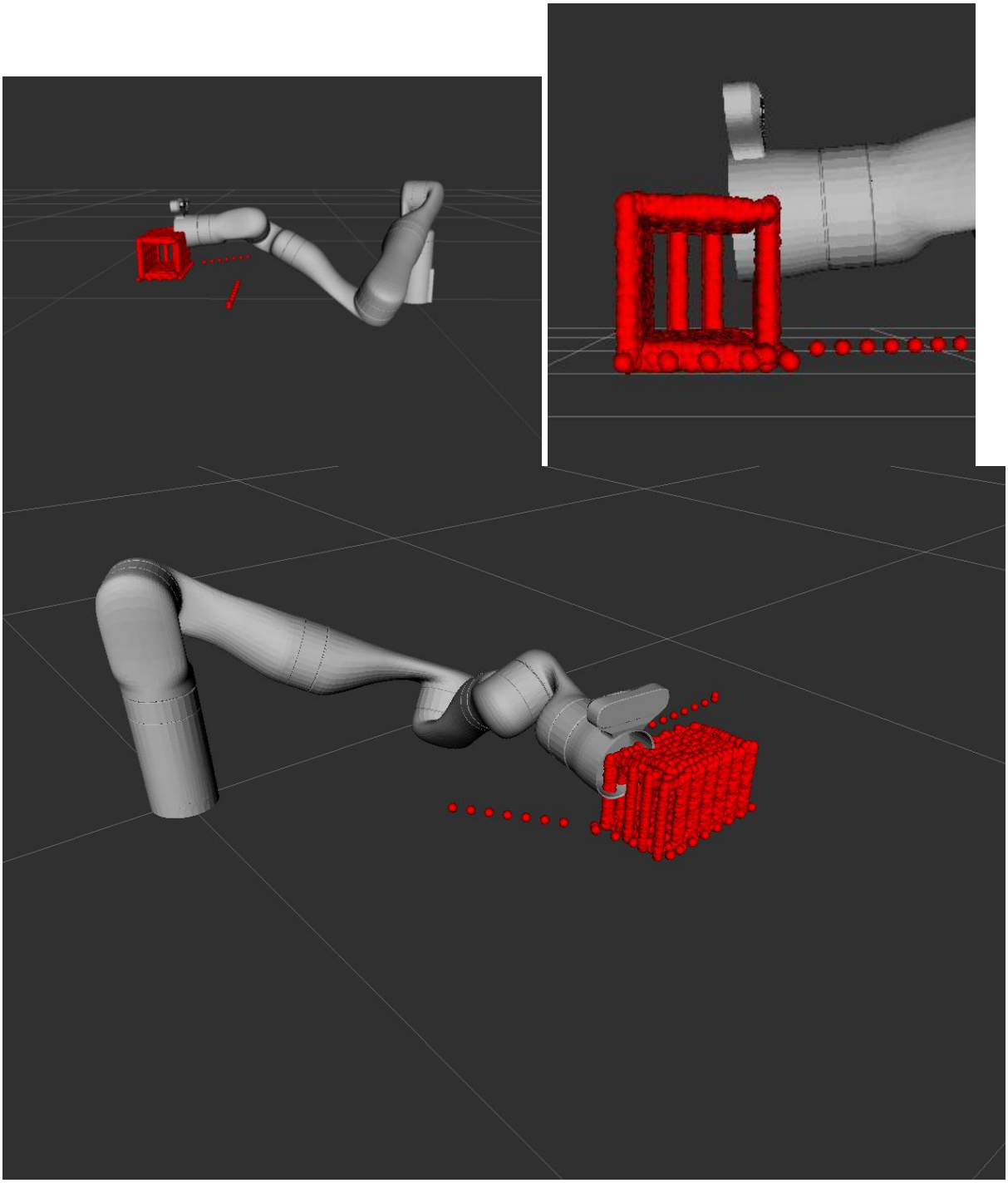
Another lesson learned was to split up tasks extremely small. This was encouraged throughout the last semester, but we could have simplified the tasks even more. This would help with a more even distribution of tasks among team members. Having team check-ins at least two times every week would also help with accountability and to solve any problems members may be stuck on.

The python interface of ROS does not have all of the functionality that the C++ interface has. The C++ interface allows the use of different network protocols and more functionality than the python interface. This has not limited us in this project as we did not need that extra functionality.

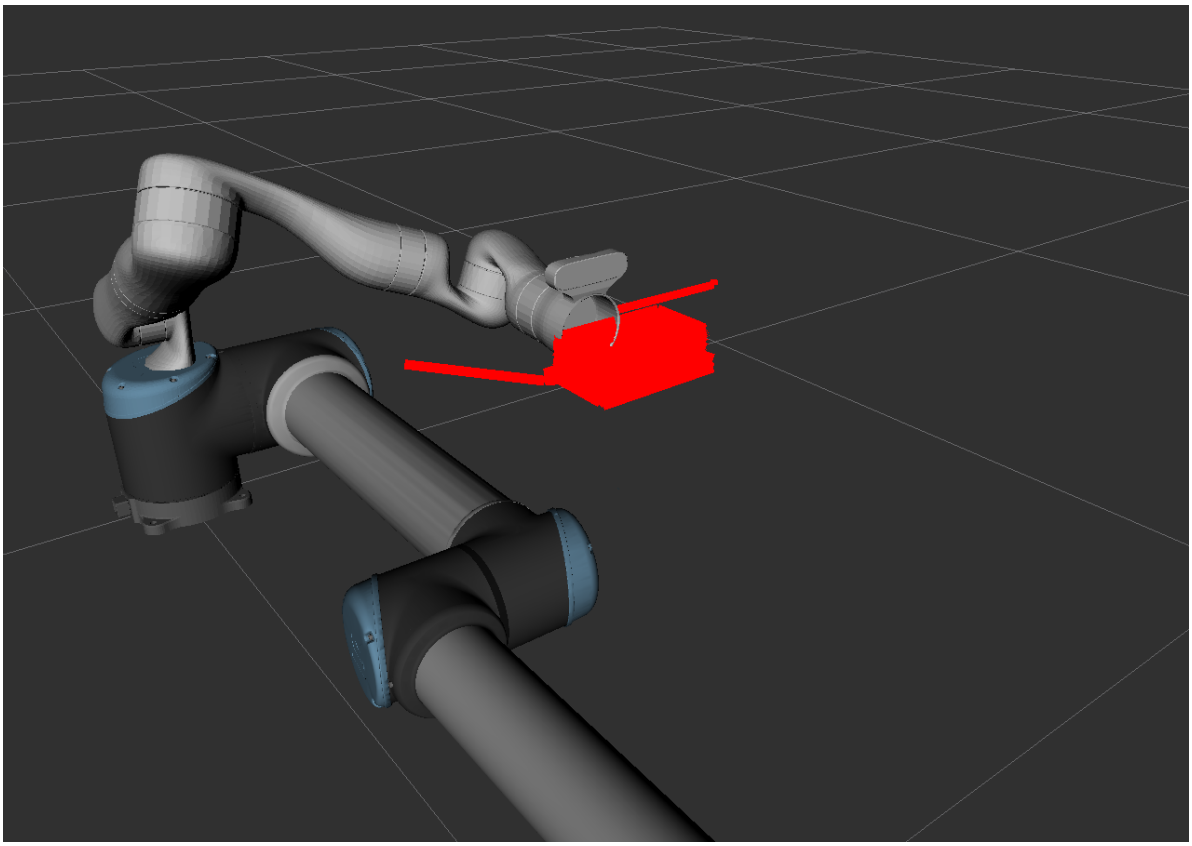
RViz has limitations on what it can do with simulation. To visualize the arms with attached objects, which are the extruder representations, we needed to use the PlanningScene display type. This, however, does not apply the transform we used to set the arms at a distance apart. Thus, we used the RobotModel display type which does apply the transform to represent the arms in the simulation.

4.2.14.1 Trying Out Different Markers:

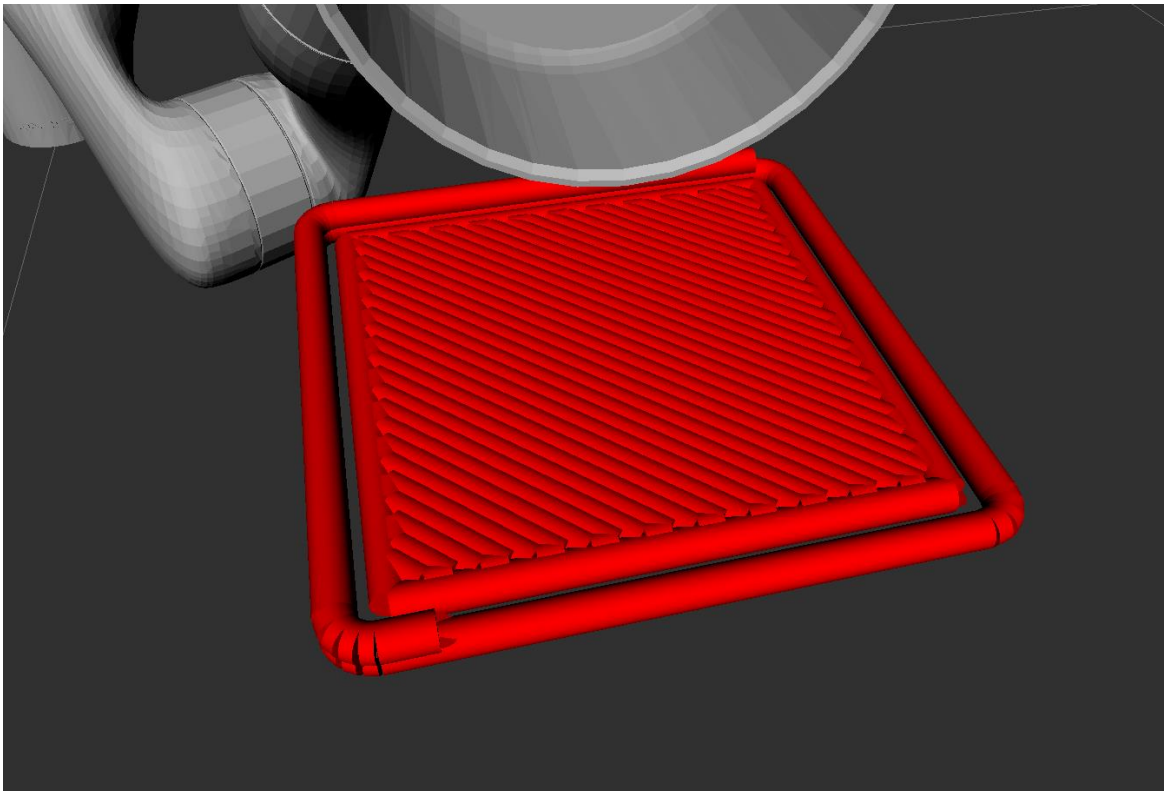
We first used sphere markers with interpolation. This resulted in poor performance in RViz and poor quality on the print visualization.



We then used line markers. This improved performance, but the line markers were unfit for visualization as features could not be seen from the lack of shading.

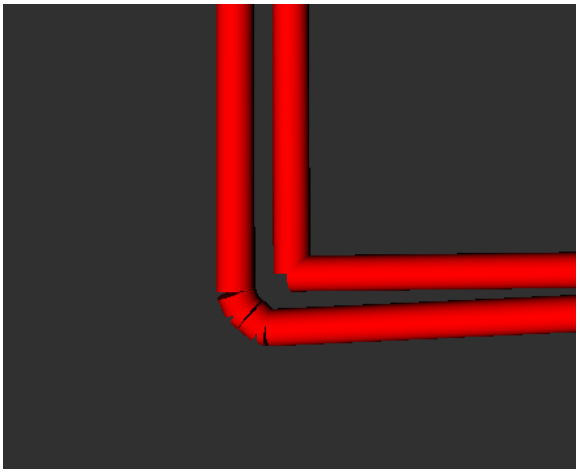


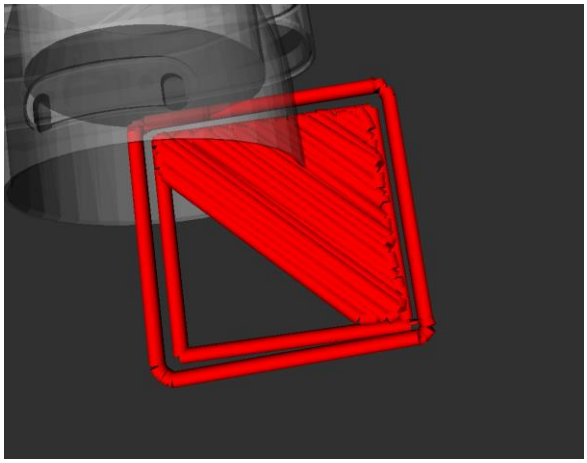
Cylinder markers were decided upon as it provided decent performance and good visualization of filament.

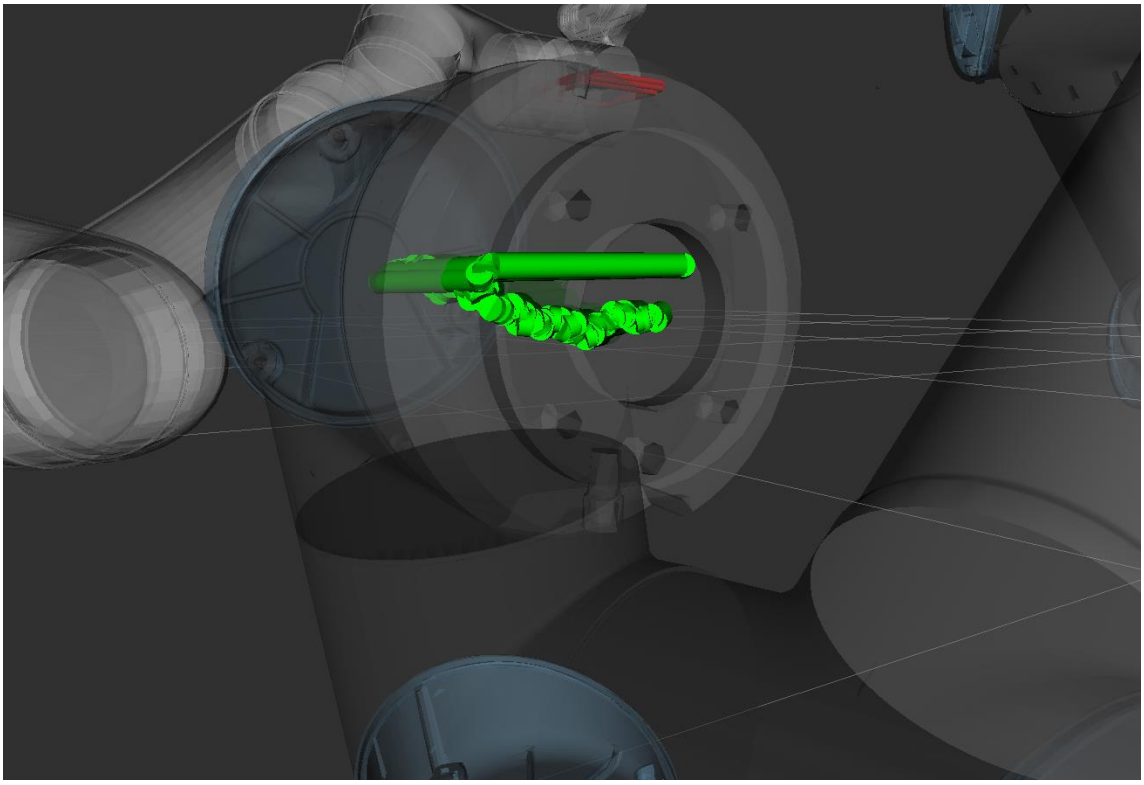


4.2.14.2 Tolerance:

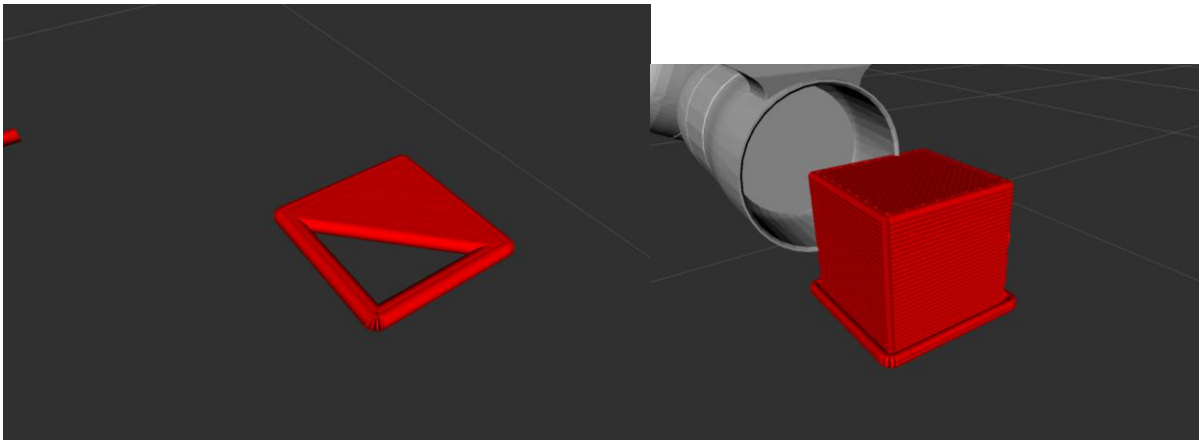
Initially, we have a high tolerance. This resulted in inaccurate and poor print quality.







Having a low tolerance achieved better results for print quality and accuracy.



4.2.15 Potential Impact & Future Work:

This project's potential impact is for AMBOTS to use multiple different robotic arms from different brands in their research and products. This allows for collaborative printing without having to develop a whole network of new robotic arms, which is impractical and inefficient. Being able to use 3rd part robotic arms allows for flexibility in the development of new ideas in additive manufacturing.

There is a lot of potential for future work on this project. Looking at the first module (the front end) we currently are just splitting the CAD file in half for the two robotic arms we are

using. Although this works for a proof of concept, you ideally want to be cutting the object based on what is the most efficient for the robotic arms. You also want to be able to use any number of robotic arms depending on what you have available. Our sponsor, Dr. Zhou, mentioned that AMBOTS is working on an algorithm that will decide which robotic arm prints what. This isn't just important for our front end but extends into the communication hub. Our communication is just pause and resume commands not necessary to print our object. If the two arms are printing more intricate pieces, the communication between robotic arms will be much more pertinent.

It would also be much more ideal to use Gazebo and M3DP-Sim instead of RViz. Gazebo is a simulation environment that has a wider range of capabilities. M3DP-Sim is a simulation tool for 3D printing viscoelastic fluids. Collision detection should also be added as well as automatically computing alignment offsets instead of manually computing them.

This project currently uses two robotic arms. Although this is good as they are different brands and it is important to see how using arms from different brands impacts the set-up and communication of the arms, ideally an undetermined number of arms will be used. Currently the firmware model is implemented to make the setup of additional robotic arms as easy as possible. By creating a firmware class, a new arm can be implemented relatively easily. A launch file will also have to be created.

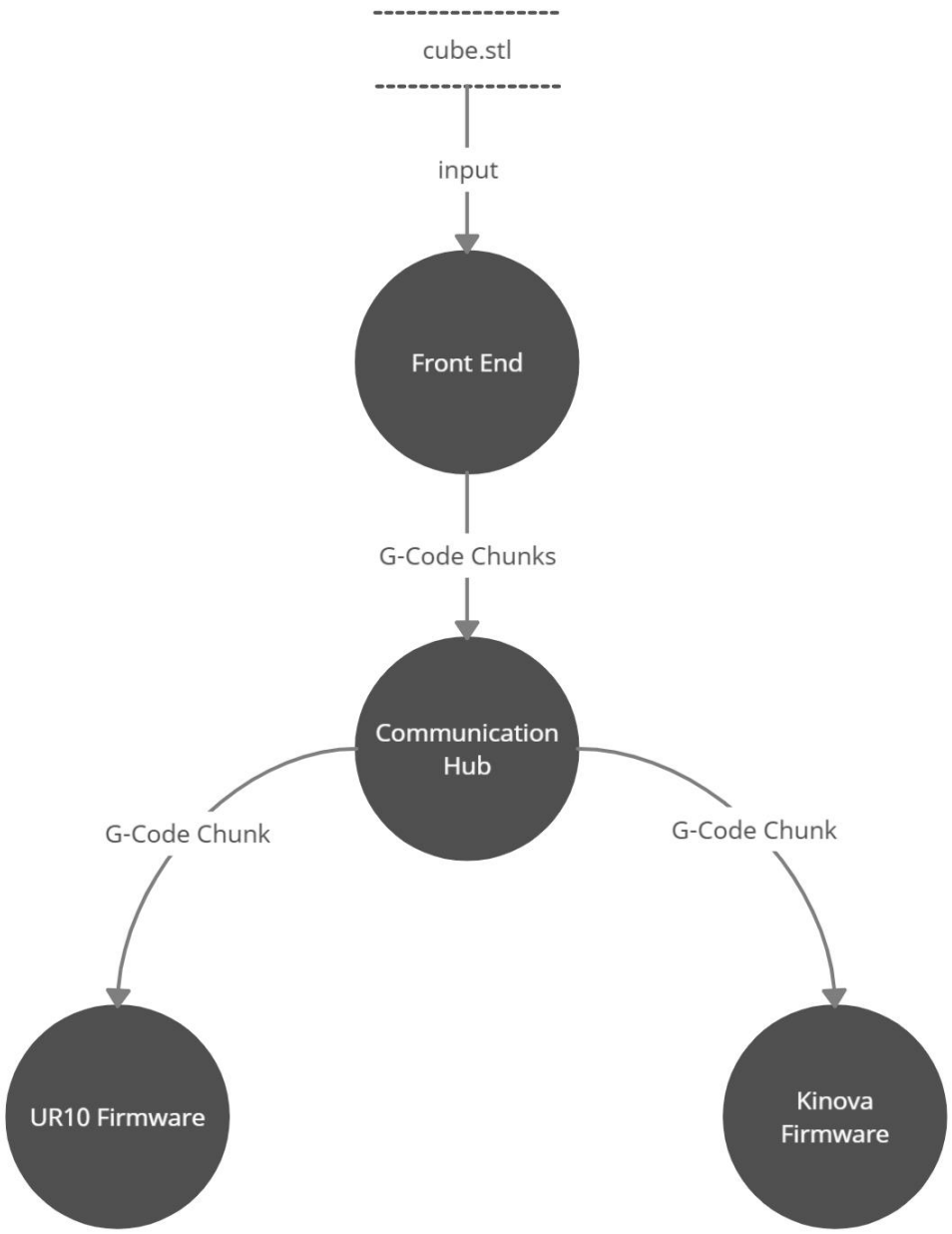
Currently we have developed a command line interface which adds a level of complexity that inhibits non-technical workers from using the system. Adding a graphical user interface would provide a more visual and user-friendly interaction with our system, increasing its usability.

Another future development that could be added is creating a custom slicer. This project utilizes PrusaSlicer which is an open-source slicer. Our sponsor, Dr. Zhou, mentioned how they would like to have their own slicer and not rely on PrusaSlicer.

4.3 Data Flow

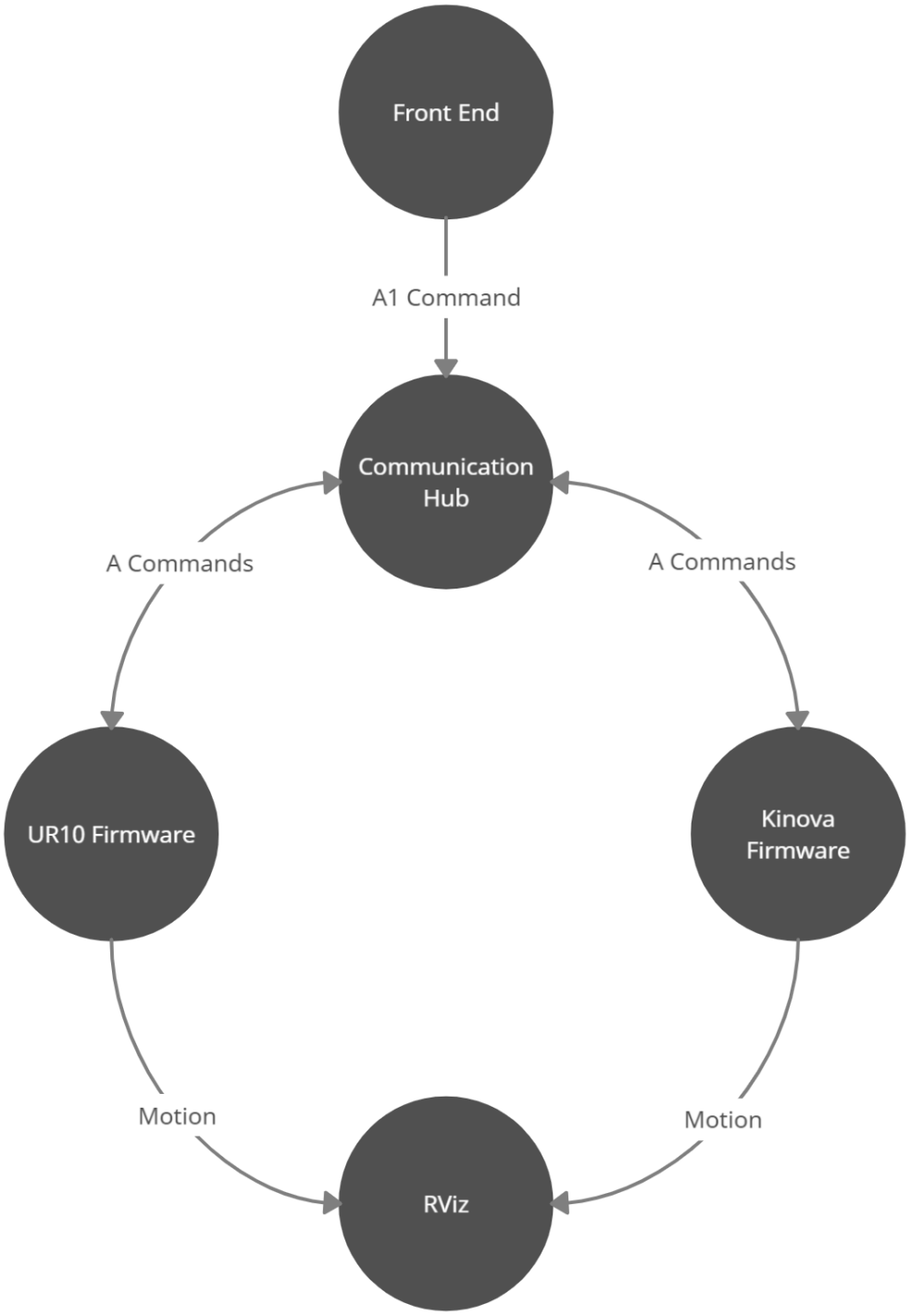
For slicing and distributing G-Code, the flow of data begins with the front end. The front end takes in an input file. G-Code chunks then flow from the front end to the communication hub. From the communication hub, the G-Code chunks are sent to the correct arms.

4.3.1 Flow of Data During Slicing and Distribution:



For printing, the A1 command is sent from the front end to the communication hub. The communication hub routes this command to the correct arm. The specified arm will start printing and may send A series commands to the other arm. The other arm may send A series commands as well. From both of the arms, the planned trajectories are sent to the RViz simulation.

4.3.2 Flow of Data During Printing:



4.4 Tasks

1. Research ROS1/ROS2, MoveIt, and Gazebo/RViz. When researching, a decision should be made on which version of ROS will work best with this specific project.

2. Research Robotic arm options and select two arms of different brands. The robotic arm should support the utilization of RViz and should preferably have vast documentation to make the learning process smooth.

3. Document set up of ROS, MoveIt, the robotic arms, and the Ubuntu environment. The version of Ubuntu used will depend on the version of ROS. This setup will take some time as each robotic arm has a different launch process. Each member of the team should have both robotic arms simulated on their machine once this step is done. The arms will be in separate simulation environments at this point. The documentation will be organized in GitLab for future usage.

4. Research G-code. An object that is to be printed will be provided via an STL file. This STL file will be imported into a slicer and a G-code file will be exported.

5. Create a program to translate from G-code file to robotic arms' native language. The G-code file will be parsed, and the coordinates will be extracted. These coordinates will then be translated into the robotics arms' native language for each robotic arm respectively.

6. Research how each of the chosen robotic arms receive commands. In order to successfully implement communication and printing, the chosen robotic arms need to be better understood. Each brand of robotic arm has its own native language. We will research how the individual arms receive movement commands and how they utilize MoveIt to calculate the trajectories necessary to end at given positions.

7. Simulate basic movement of robotic arm choices in RViz. This will include looking into the launch file (typically launch.py) of each arm. To verify that we have set up the robotic arms correctly we will simulate them in RViz and have them move in both a line and a square.

8. Download and use slicer to create G-code file. We will use PrusaSlicer [24], which is an open-source slicer that can be run via command line. A batch file that takes in an STL file will then be created for easier use.

9. Simulate the G-code movement of robotic arm choices in RViz using basic G-code files. This is the process to make sure the program created in step 5 works. The program can then be debugged and improved upon.

10. Run both robotic arms in the same environment. Up until this point the robotic arms will be run in separate environments. Although on the same machine they will not be run in the same instance. This will be a process of figuring out how to have the robots running simultaneously and will allow for communication to happen in future steps.

11. Create a custom package to launch and run programs from. This should be compatible with all of the previously worked on steps.

12. Simulate printing using a marker in RViz. While the robotic arm is moving using the coordinates from the G-code file, the marker should begin simulating extruding. This is to prepare for the collaborative printing process. This should be done with both robotic arms in the same environment.

13. Create a generic firmware class for each arm to inherit from. This will most likely just be a restructuring of the code we have already developed but will allow for the ease of adding additional arms in the future.

14. Research and install PyMesh. Create installation instructions. PyMesh will be used to cut a STL file into separate chunks for each robotic arm to print.

15. Add custom g-code commands for the communication hub to use. These commands will tell the robotic arms to take certain actions.

16. Implement communication hub using an ack system.

17. Use Pymesh to cut the original STL file in half. This can also have the capability for other types of files to be printed but is not required to.

18. Find and set up a concrete or material extruder for both robotic arms. If a concrete extruder cannot be found, some kind of extruder or object to simulate an extruder should be added.

19. Connect all of the modules together (Front end, Communication Hub, Firmware, and RViz). There should be some kind of user interface allowing the user to choose an STL file to print.

20. Create Demo, Unit Test and fix any bugs that are found during testing.

4.5 Schedule

Tasks	Dates
1. Reasearch ROS, MoveIt, and RViz	11/07 - 11/14
2. Research and Select Two Robotic Arms	11/07 - 11/14
3. Document Setup of ROS, MoveIt, Robotic Arms, and Ubuntu	11/14 - 12/05
4. Research G-Code	12/05 - 12/12
5. Create G-Code Extractor	12/05 - 12/12
6. Research Robotic Arms Movement Commands	12/12 - 01/02
7. Simulate Basic Robotic Arm Movement	12/23 - 01/02
8. Create Slicer Script	01/02 - 01/09
9. Simulate G-Code Movement of Robotic Arms	01/02 - 01/16
10. Run Both Arms in the Same Environment	01/16 - 01/30
11. Create Custom Package to Launch/Run Programs	01/30 - 02/06
12. Simulate Printing Using RViz Markers	02/06 - 02/13
13. Create Generic Firmware Class	02/06 - 02/13
14. Research & Install PyMesh	02/13 - 02/20
15. Add Custom G-Code commands	02/20 - 02/27
16. Implement Communication Hub	02/27 - 03/13
17. Cut STL file in half	03/13 - 03/27

18. Add Extruder	03/27 - 04/03
19. Connect the Modules & User Interface	04/03 - 04/10
20. Demo, Unit Test, Bug Fixing	04/10 - 04/24

5.0 Key Personnel

Stanley Van – Van is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed the relevant courses Software Engineering, Database Management Systems, Programming Paradigms, Operating Systems, and Computer Networks. This student will be responsible for making the G-code file reader for the ROS nodes and piping the G-code files to the robots along with any other tasks that need extra assistance. His contact information is: stanleyc.van@gmail.com

Cassandra Nelson– Nelson is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. She has completed relevant courses up to and including Database Management, Programming Paradigms, and Software Engineering. She held a C-Unix Programmer Analyst Intern position, and currently works as a Computer Support Assistant at the CORD on the University of Arkansas campus. Nelson will be responsible for the Frontend module, documentation, and any other tasks that need extra assistance. Her contact information is: cass.nels@outlook.com

Michael Darden– Darden is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed relevant courses such as Programming Foundations one and two, Programming Paradigms, Computer Organization, Software Engineering, Computer Networks, and Discrete Mathematics. He has interned at Clear C2 for the summers of 2020, 2021, and 2022. Clear C2 is a web development company that creates costumer recourse management tools for other businesses. This student will be responsible for setting up the environments and simulations of the robots. This includes development in ROS to adapt the environments to support the different types of robots. His contact information is: mjondarden@gmail.com

Alvaro Becares Fernandez– Becares is a senior Telematics Engineering major in the Telematics Engineering Department at the University Carlos III of Madrid, currently an exchange student at the University of Arkansas. He has completed relevant courses such as Artificial Intelligence, and Big Data Analytics and Management. Becares will be responsible for compiling research done along with any other tasks that need extra assistance. His contact information is: varobecares@gmail.com

Dr. Wenchao Zhou– Dr. Zhou is an associate professor in the Mechanical Engineering department at the University of Arkansas. He is the co-founder and Chief Technology Officer of AMBOTS. After receiving his PhD in Mechanical Engineering from the Georgia Institute of Technology, he has participated in research that led to the publication on swarm manufacturing cited previously in this proposal [2] along with others not mentioned in this report.

Zachary Hyden – Hyden received a bachelor's degree in mechanical engineering from the University of Arkansas in 2019. He is now the Chief Mechanical Engineer at AMBOTS. During

his undergraduate education he was an Additive Manufacturing Researcher in the AM^3 lab at the University of Arkansas.

6.0 Facilities and Equipment

Because this project will be developed using simulation software, there will not be any facility usage required. The funding required to acquire the robotic arms is estimated to not arrive within the time period of the development and completion of the project. This is another reason that facility usage is not required. As for equipment, each member will be using a computer with Ubuntu installed and running on a virtual machine or natively as the operating system. Although there is a potential for this project to move to the physical lab, initially everything will be developed through simulation. It is then up to the sponsor whether to actualize our solution in their lab in the future.

7.0 References

[1] AMBOTS, https://www.ambots.net/

[2] Poudel, L., Marques, L. G., Williams, R. A., Hyden, Z., Guerra, P., Fowler, O.L., Sha, Z., and Zhou, W. (February 16, 2022). "Toward Swarm Manufacturing: Architecting a Cooperative 3D Printing System." ASME. J. Manuf. Sci. Eng. August 2022; 144(8): 081004. https://doi.org/10.1115/1.4053681

[3] 3D printing, https://en.wikipedia.org/wiki/3D_printing

[4] ROS, https://www.ros.org/

[5] MoveIt, https://moveit.ros.org/

[6] RViz, https://www.stereolabs.com/docs/ros/rviz/

[7] Investopedia, https://www.investopedia.com/terms/d/degrees-of-freedom.asp

[8] Linux, https://www.linux.com/what-is-linux/

[9] TechTarget, https://www.techtarget.com/whatis/definition/CAD-computer-aided-design

[10] G-code, https://en.wikipedia.org/wiki/G-code

[11] Slicer (3D printing), https://en.wikipedia.org/wiki/Slicer_(3D_printing)

[12] What is Python? Executive Summary, https://www.python.org/doc/essays/blurb/

[13] UR10, https://wiredworkers.io/product/ur10/

[14] Kinova Robotics, https://www.kinovarobotics.com/product/gen3-robots

[15] PyMesh, https://github.com/PyMesh/PyMesh

[16] Additive Manufacturing, https://www.additivemanufacturing.media/articles/robots-assemble-a-new-path-to-autonomous-mobile-3d-printing

[17] General Electric, https://www.ge.com/additive/additive-manufacturing

[18] 3D Systems, https://www.3dsystems.com/3d-printers/sla-750

[19] Scholarpedia, http://www.scholarpedia.org/article/Swarm_robotics

[20] Unbox Robotics, https://unboxrobotics.com/

[21] Rosotics, https://www.rosotics.com/

[22] NASA, https://technology.nasa.gov/virtual-event/startup-nasa-series-rosotics-inc

[23] PrusaSlicer, https://www.prusa3d.com/page/prusaslicer_424/

[24] STL (file format), https://en.wikipedia.org/wiki/STL_(file_format)