# A System to Explore Using a Collaborative Robot in Improvisational Dance Practice

Nick Weldin[1]

August 2019


[1]Supervisor Franco Raimondi

**Abstract**

This project built a system to explore the use of a UR10 collaborative robot arm in the context of improvisational dance. A LaunchPad Pro midi interface was used with it to enable multiple positions, and sequences of positions to be quickly and simply recorded from physical manipulation of the robot arm. An end effector was built to make the physical manipulation easier, and enable the system to be used without needing the robot pendant or computer screen. The system was tested with a range of users and improvements made based on feedback and observations. Recommendations for further development of the system are also made.

# Contents

# Chapter 1

# Introduction

## 1.1 Problem Definition

Traditional industrial robot arms use programming environments optimised for the tasks that they perform. These involve operating the robot primarily from a distance, using indirect methods like pendants and code. This is often aimed at one specific task that will then be repeated continuously until the robot is reprogrammed. Collaborative robot arms are emerging as a new style of industrial robot that enables faster programming than traditional arms, as well as sharing the same physical space as a person, but the tools still assume similar use cases and the programming tools still require multiple confirmations to initiate simple actions.

Dance is often devised in an improvisational context, trying out lots of different things quickly in a fluid and exploratory way that does not fit well with the current tools for programming robots. It is difficult to use these programming methods, even with a collaborative robots to work in the fast, spontaneous way that a dance rehearsal would ideally proceed.

## 1.2 Research Question

How can the new possibilities that collaborative industrial robot arms offer be used to enable dancers to work in an improvisational, exploratory way with a robotic arm? This questions will be explored by proposing a system that would support this, building it and testing it with people.

## 1.3 Structure of the report

Chapter 2 looks at the background to this question, considers how robot arms are programmed, some digital systems that have been used in improvisation, Chapter 3 considers the technologies that can be used to implement a system. Chapters 4 and 5 describe the proposed system, how it was implemented and tested, and Chapter 6 considers looks at future work may come from this investigation.

# Chapter 2

# Background

## 2.1 Cobots

Collaborative robots (cobots) have been developed over the last twenty years, with the first commercial cobot being sold in 2008 [47]. A good overview of what a collaboration between robots and people involves is provided by Billard in [16]. The key feature of a collaborative robot arm is that it can be used around people, unlike traditional industrial robot arms which need to operate in industrial environments that ensure people are not present while its operating, either by excluding them through safety cages, or detecting their presence if they get too close and stopping. To enable people to be able to work around cobots their maximum speed is slower, and they will only lift fairly low weights to ensure that any collision has low momentum. In addition the robot will detect collisions at a fairly low threshold (either through sensors on its exterior, or through monitoring joint torques) so it can stop before causing serious injury. The joints often have physical compliance built into them as well, so that they have flexibility or give. In 2016 the International Organisation of Standards published a standard for collaborative robots [37], which provide guidance to ensure that installation of the arm does not remove compliance because of the way the arm is used. In addition to the physical characteristics the mechanism for programming them will usually involve the ability to use elements of programming by demonstration (PbD) where a human operator physically manipulates the arm to move it to waypoints, or record movements. The aim is for the robot to be able to be programmed by the people who work with it, rather than robotocists. These abilities offer new opportunities for robots to be used in a dance context.

## 2.2 Some Examples of Robot Arms in Dance

Traditional industrial robots have been used in many dance performances, but because of their non-complaint nature this usually involves the entire performance being pre-programmed and the dancers needing to fit around the robot with their movements. With large robots the robot may not share space with the dancers very often, if at all, as described by Hyuang Yi [73]. The programming can take significant time, and is often done away from the robots and dancers. Hyuang Yi can spend ten hours programming one minute of movement [62]. Madeline Gannon has been exploring sharing space with a large industrial robot and tracking people to avoid collision [29], but when it came to an actual installation this still involves excluding people from the robots environment [30]. Cobots were used for the Slave/Master performance and installation at the Victoria and Albert museum for London Design Week [63]. The performers were able to interact with these robots more easily [75], but the programming interface they used was Powermill CAD/CAM software [12].

## 2.3 Programming Arms to Perform Tasks

How a task is performed, without traditional programming has been an active area of research for many years [24]. A good overview of work in this field in relation to robotics is provided in [15, 1]. The approaches can be divided into Programming by Demonstration (PbD), and Learning by Demonstration (LbD). In PbD the robot is physically manipulated to perform the task and in LbD the robot "observes" a task in some way and works out how it would perform the action. As cobots tend to work in less constrained environments a key issue they face is generalisation of a task - how to perform it in slightly different conditions. A key element in this is trying to tease out what the purpose of the task is, and how can you judge what parts of what you have learnt need to be preserved, and which can be altered while still successfully performing the task. In addition to this in LbD there is an additional mapping exercise between the physical capabilities of the demonstrator, and the robot that is trying to learn the task. As you move to more autonomous systems it is additionally complicated by the need to work out when and what to observe.

In many ways this is mirrored in the interpretation of dance. While the choreographer may have certain ideas in their head when they are devising a dance, that needs to be transformed into physical movements by dancer/s, and the audiences perception of these may be very different. The intentions and ideas may be difficult to describe in words, and the process of choreography tend to be very personal [17]. Dance is an expressive medium, and the emotional content of what

is intended and produced is of great importance. The devising, recording and transmission of dances has been an evolving practice too. Some choreographers use notation systems like Laban (although none are universal) to physically transcribe pose, movement and emotional intent, and these have been used with robots to interpret and generate robotic dance moves [56]. As technology has developed some dance practitioners have embraced it as a way of documenting performances for preservation and analysis. Using motion capture suits and rigs huge amounts of data can be generated, but the difficulty is like in LbD how we do move from data to information, and then to higher level knowledge. While much research is still being carried out in many fields to address this question Gelder [25] is clear that we are a long way from being able to read the emotional intent in dance from observation with computers. Something that may be of use is the work that the WhoLoDance project has done. They have been looking ways of digitally documenting and classify dance across a wide range of genres. This has been a difficult process because of the great diversity of dance styles and methods, but they have produced a series of key metrics that [74] could be amenable to some automatic generation if you have skeleton tracking data for dancers.

Improvisation in contemporary dance is a key part in the devising of work. Often the dancers are trying to work out the inverse kinematics of their bodies - how do I connect this movement to that movement without dislocating my shoulder when I try to go faster, through experience and trial and error. It is a process of discovery and experimentation. The aim of this project would be to devise methods that would enable you to work quickly, kinesthetically and experimentally with the robot as part of this process in rehearsal, and potentially performance too.

Programming a robotic arm in an industrial context usually involves specifying a number of positions for the arm to move to. These are referred to as waypoints and you are usually primarily concerned about the pose of the end of the robot arm (its position and orientation in space), and is referred to as position control. At each of these waypoints the robot performs some action, using a tool (end effector) attached to the end of the arm, for example a gripper can pick some thing up, or put it down. The robots end effectors position can be specified by the angle of all it joints. From this you could calculate where the end effector is in relation to the base. This calculation is called forward kinematics. In order for the robot to move from one position to another it needs to calculate how to move its joints to get to the new position. The simplest way for it to do this is to simply move its joints to the new positions (assuming it will not hit itself as it does that - it may need to move joints at different speeds to ensure it is not in self collision). This is fine if you don't care about the route that it takes, and is referred to as moving in joint space. It is the most 'natural' movement for the robot to perform, and the

simplest for the controller to calculate. The controller will plan the movement of the joints to minimise sudden changes in velocity or acceleration, and at speeds to enable all joints to reach their final position at the same time a simple approach to performing this kind of control is described by Aleksander Zivanovic in [45]. If you need the end effector to move in a straight line, or otherwise defined path then you would move in Cartesian space - now the robot needs to work out how to move its joints to get the end effector to move between the waypoints while maintaining a linear path, or possibly a series of curves. All industrial robots ship with a controller unit that is able to perform these calculations and generate data to control the robot. It will be optimised for the robot that it is supplied with, and each companies product will work slightly differently. They will also be able to perform inverse kinematics - given a pose in space, what angle should the joints be set to to put the end effector in that position. These are much more complicated calculations than moving in joint space, and may have zero or more answers (the robot may not be able to physically move to the position you have requested, or follow the path specified). To have fine grained control of the robot you break the movement down into smaller and smaller parts by inserting more waypoints. This is why it takes Hyuang Yi so long to programme his performances.

In terms of dance you may be interested in poses that people adopt, but you also interested in the quality of the movement that they make, artists have been aware of these issues for a long time [9]. Instead of using positional control and and relying on the controller to work out how to move the joints, you can instead determine the velocities that you want joints to move with, and vary these over time. This opens up the ability to control the quality of the robot arms movement, not just where it is moving to, but is a more complicated process than either of the previous methods .

## 2.4 Other Digital Systems that Support Improvisation

There are some interesting examples of technical and programming systems that aim to support or encourage live performance in different artistic fields that may be useful. The music software Live [2] changed the way that people thought about performing and creating electronic music in a live context. It took the idea of loops of sound and produced a grid interface slots You can quickly and easily record audio into a slot, or drop some pre-recorded material on to it, and then play it back by selecting that slot. You could very easily make the sound loop and manipulate it to synchronise with other loops. It enabled experimentation and improvisation by providing tools for manipulating the sound in each slot while

everything else continued - previously most music software had a much clearer differentiation between editing and play modes. Isadora [20] is a graphical programming environment for producing and controlling multimedia content in a live context. It was originally developed to support the live manipulation of digital content for dance performance and like the Live program, editing and playing are not separate, distinct processes. There is not a program/run cycle, it is constantly running and as soon as you change any connections between the nodes the system reflects that change with what is produced. The aim was to enable the software to be used in the rehearsal process with the dancers, producing content that could be controlled and driven by them onstage, rather than being pre-recorded and set in time. Mark Coniglio, the programmer who develops Isadora, felt that this led to stilted performance by the dancers as they tried to keep in sync with the projections, rather than the projections responding to them. These pieces of software will not be used to control the robot, but will guide some of the choices for how to structure the system.

Wekinator [27] is a piece of software that is aimed at musicians. This tool enables you to use data streams, usually from sensors of some description, and builds machine learning models from them as you perform. You can then use these models to produce open sound control messages that can interface with other software to drive a wide range of multimedia systems. It is good for quickly trying out different ways of interpolating between multi-dimensional data sets, like skeleton tracking data, or other sensor data, and could have a role in building tools to recognise some of the characteristics that the WhoLODance project have described in a live context. It can also be used for live interactive learning of gesture recognition. Other tools for exploring this could be gesture recognition toolkit [32], or the recently released Runway system[55].

## 2.5 Responding to Dancers Movement

To enable the system to respond to the movement of the dancers around the robot there are a number of other sensor technologies that may be appropriate. Full body tracking systems can be used, but they involve the setup and calibration of many cameras, along with the infrastructure to position them around the space. RGBD cameras will give depth data, but tend to be fairly short range (<10 meters) and have a narrow field of view. LIDAR units provide longer range, but only give you slices through the world around you, and multiple channel units are very expensive. More traditional RGB cameras can provide a much longer range, and may cover a whole room, but need a lot of processing to produce information about people in the image. OpenPose [18, 57, 64] is currently the best open source

implementation that has been widely used in many fields. Advantages of this route are that it supports an arbitrary number of skeletons (so don't point the camera at the audience), can also detect facial features and hand/fingers if required, and with multiple cameras can generate 3D skeletons. It can also cover larger areas, the limit being the field of view of the camera, and the people being large enough in the image to be recognised The disadvantages are it needs a computer with multiple powerful graphic cards to accelerate the process for it to work in realtime (30 frames per second or faster).

An advanced concept that could be explored with the this kind of data would be altering the quality of movement of the arm. One technique that could be considered for this would be style transfer machine learning processes. These have had a very high profile recently in terms of producing images in the style of famous painters, by taking style data from one image, and applying it to the content of another image. They have also been used with time based medium - to video [54], character animation [34], and motion data [33, 14]. The aim would be to use performers motion data to influence the style of movement of the robot arm. There are a number of issues to resolve for this. One is the correspondence problem - how do you map the data about the movement of people to the movement of the robot. There is not a one to one correspondence between the physical characteristics of the dancers (even if you just consider one of their arms). While this is due in part to robots being designed to do mechanical tasks that may not need to replicate human form, there is also a fundamental engineering issue with replicating human like movement. Human joints have more than one degree of freedom (the shoulder and wrist especially), while all current joints for robots work in one degree of freedom, and put multiple joints close together to try and emulate more sophisticated joints that are available in animals. The difficulty in producing mechanical joints that have multiple degrees of freedom in robots is primarily how to drive them with actuators. The other issue in terms of this project, is that the models take significant time to train, and so would not lend themselves immediately to the short time scale interaction that we are looking to create. An interesting project that is looking at correspondence from the other direction (robots to humans) is described by Gemeinbock and Saunders [31].

# Chapter 3

# Technologies

## 3.1 Robot Operating System

Robot arms have specialist controllers that consist of sophisticated realtime control systems for monitoring and controlling the motors in the joints. Traditional industrial arms will usually have a self-contained system for controlling them from a pendant, that may also interface to more sophisticated propriatary control software that can be used offline to plan more sophisticated tasks, and then pass the planned operation back to the controller. There is often an API of some description that can be used to programme the arm using any programming language, but the mechanism is different for different companies, or may not exist at all.

Robot Operating System (ROS)[46] is an open source project that has had a profound effect in the world of robotics research and industry over the last ten years. Despite its name it is not an operating system, but provides the kind of functionality that an operating system provides for computers, but for robots. It provides robot agnostic ways of passing messages between processes, hardware abstraction, methods for describing robots that then leverage reference implementations of common functionality and tools to help monitor, and troubleshoot robotic development. It is often referred to as "plumbing" for robots. It was originally developed in a startup called Willow Garage in 2007 with the aim of producing commercial robots that could perform useful tasks. The core elements are developed in the spirit, and using many of the processes of the Linux operating system - it is a common resource that is of benefit to many groups and partners, both academic and commercial, so everyone benefits from its joint development. Following the closure of Willow Garage in 2013 the open source robotics foundation (OSRF) was formed by some of the former employees of willow Garage to maintain and

develop the core functionality of ROS. They renamed themselves Open Robotics in 2017, and continue to support and develop the core components of the ROS infrastructure.

A ROS system consists of a computational graph made up of nodes. These are programmes which perform tasks - they can be written in any programming language that has ROS bindings (or you can write your own bindings if you need to use a language that is not currently supported). Most nodes are written in Python or C++. Each node performs a particular task and communicates with other nodes through publishing and/or subscribing to topics. Topics are unidirectional, typed data connections designed for streaming data. Nodes can be launched and stopped without stopping the rest of the system, enabling incremental development and troubleshooting to take place.

Before nodes are launched roscore needs to be running - as nodes launch they advertise their existence to roscore, and it provides details of other nodes that are subscribing or publishing to topics that the node uses, and they then directly communicate with each other. Roscore maintains a parameter server that holds global data for the system - this usually includes a machine readable description of the robot, and other parameters associated with it so that many calculations can be automatically generated from them, including transform frames between defined frames as the robot moves.

On top of the basic functionality that topics provide ROS has services to provide call and response functionality. It does this by combining two topics in a defined way. Services are designed for requests that complete quickly (asking for the result of a simple calculation, or sensor reading for example). For longer tasks that may require pre-empting there is a more complicated construct called an action that is built on top of topics. These are used to instigate more complex actions, like navigating a robot to a defined location.

From these basic building blocks complex robotic systems can be built. This distributed system makes it relatively easy to share processing tasks across multiple computers if needed, to increase the computational power available, or to move processes away from mobile battery powered robots, to tethered mains powered computers. The open source nature of ROS has encouraged people to share their code, and when publishing academic research make code available so that people can validate and build on the results. Because of this it has become the dominant system in academic research and is being widely used commercially in robotic startups and industrial research. Examples of commercial organisations involved in ROS development include NASA, Bosch, Google, Amazon, Ubuntu, Microsoft.

ROS industrial [36] is a consortium that has been working to implement bridges between the APIs that industrial robot arms have with the ROS ecosystem, to enable them to be used with the sophisticated algorithms and planning libraries that ROS makes available.

While ROS is widely used it is not suitable for all situations. It is not a real time, deterministic system, which can be important in robotic systems. There is overhead involved in publishing and subscribing between nodes. It was originally design with a single robot in mind, with high speed networking available between components in the system. Roscore is a single point of failure in the system, and there are few guidelines around good practice for building more complex systems, including working with multiple robots.

To address these issues ROS2 has been being developed in parallel to ROS since 2015 [44] ROS2 deliberately starts from scratch again with a new architecture. Roscore is removed and nodes are true peer to peer processes. Deterministic behaviour is possible, and many of the infrastructure components use tools developed by other groups to enable the core developers to concentrate more on the robot specific elements rather than the infrastructure.

## 3.2 Robot Arms

During the project there was access to two different collaborative robot arms to work with, Rethink Robotics Sawyer and Universal Robots UR10.

### 3.2.1 Sawyer

The Sawyer robot is a robot arm with 7 degrees of freedom[50]. It is a conformal robot and was manufactured by Rethink Robotics between 2015 and 2018. It is the successor to their Baxter robot with changes specifically aimed at deploying it in a light manufacturing context[51]. Rethink produced their own visual programming environment for Sawyer, called Intera [48] that enables you to program the robot to perform a range of activities from the robot arm using just the small screen built into it and the buttons on the arm. This interface was built on top of the Robot Operating System that runs underneath, using nodejs, with the generic nodejs code being contributed back to the community in the rosnodejs package [58] . It is possible to connect to Intera over a network connection, and open a web browser with a visualisation of the robot and a programming interface that is shared with the on board programming system. Some of the more sophisticated options with Intera can only be used from the web interface, but the primary mode of initial operation is clear the physical programming interface (you have to touch,

and physically interact directly with the robot arm to use it). While Baxter was only available as a manufacturing robot (booting only to Intera software), or a research robot (booting only to ROS), Sawyer introduced the option of booting to either. Using ROS it is possible to develop your own interfaces and run your own kinematic solvers to control the robot if you want to. Intera enables people who know little about robotics to programme the robot, and is aimed at people occasionally programming the robot, with many prompts and confirmations needed before a new movement is added, and is good for producing a script to primarily carry out a single task. Giving it a range of different movements it may use, and selecting between them would need to be explicitly programmed by the operator using conditional statements in the visual programming environment, with a good understanding of the structure of the programming task. Use of external interfaces is limited to using the IO pins of the included PLC, or over network sockets or a MODBUS connection in very restricted operations.

### 3.2.2 UR10

The UR10 [52] is similar to the Sawyer, but different in some key ways. It is conformal like the Sawyer, but is stiffer when being manipulated in the mode where it can be back driven (referred to as freedrive for the UR10). It is a similar size to the Sawyer, but has six degrees of freedom rather than the seven that Sawyer has. It is a bare robot arm, with no buttons or display on the arm itself. The UR line of robots is consider by many to be the main competitor with Sawyer in the light manufacturing robotics market, and their success was closely related to Rethinks demise.

The UR10 uses a more traditional hand held pendant with a touch screen for operation and programming, which they call the polyscope. This is connected to the controller box by a long heavy cable. In order to put it in freedrive mode there is a physical button on the back of the polyscope that must be held down, or an on-screen button that can be held down. It is difficult to move the arm accurately with one hand (the other needing to hold down the button on the polyscope) if one person is operating the robot. It is more practical to have one person hold the button down and another person to manipulate the robot, especially for the joints close to the end of the arm where the shorter sections mean you have less leverage and you need to use two hands to apply appropriate force to get them to move. The polyscope also has a view dedicated to moving the arm around, with controls for individual joints, as well as traditional industrial arm jogging controls, providing the ability to move in relation to the base, world or tool and to linearly in all directions and to change the orientation while maintaining the position (all subject to the physical limitations of the robot kinematics). It is clear

from this that in the absence of any other physical affordance that the preferred way of moving the robot is likely to be jogging it from the screen, rather than physically manipulating it. There are issues with jogging using the polyscope too. It is hard to watch the robot, and move it using the on screen controls. Some of the icons on screen to activate movement are quite small, and it is hard to prevent your finger moving on the screen especially if the pendant moves at all while you are using it. The screen also provide no haptic feedback to indicate if you are on target, or how you should move to get to the target. Many other pendants provide a sophisticated joystick for performing jogging actions for these reasons.

The arm can be programmed for many traditional automation tasks using a graphic environment on the polyscope. This uses functional blocks like Intera on Sawyer, but is is a more traditional linear programming structure rather than using a tree structure. There is also a text programming interface called URScript. This can be added to graphical programmes on the pendant in limited ways, or written on a computer and moved over to the polyscope to run. To help with writing scripts there are simulators available that run on a computer and emulate the polyscope. Settings can be changed on the simulator, and graphical programming can be carried out (although freedrive is clearly not a meaningful option). Written scripts can be loaded into it and run , with one of the tabs on the polyscope providing a graphical rendering of the arms movement. This tab can also be used on the actual robot to preview what movement a program will produce before getting the robot to move. Another useful feature in the simulator is that you get access to the underlying system. When a graphical program is run it is first compiled into URScript. These scripts can be located in the system to see how Universal Robots create certain behaviour using URScript.

URScript commands or full programs can also be sent as ASCII text to a network socket on the controller itself to enable control systems to be built using any programming language that can talk to a network socket. There are a wide range of libraries available on the internet, with varying degrees of sophistication and support. Things are further complicated by the fact that some URScript commands only work if executed on the polyscope, but are blocked from being called remotely. Historically Universal Robots have offered little support and patchy documentation of these features, and have released new versions that change or break behaviour, making any community produced code very sensitive to which versions of the robot software they work with.

To use the UR10 with ROS the ROS Industrial consortium [36] has maintained ROS packages [26] [61] that provide access to the robot through a node in ROS. The current released version is ur_modern_driver. This provides a ROS interface for interacting with the robot. It publishes and subscribes to a range of topics to enable

control of the robot, and to get feedback from the on its joint positions as well as the state of the IO ports. Integration with MoveIt [22], a path planning with collision avoidance system that is widely used in ROS to generate trajectories that avoid obstacles in the environment using realtime sensor data has also been implemented. It still suffers from the same issues with backward compatibility, and new releases breaking functionality as it is not maintained or supported by Universal Robots. A technical reports on the current limitations of these approaches have been produced [8] which helped define a roadmap for further work improving this situation.

Following universal robots joining the ROS Industrial consortium [35] they announced in December 2018 that they were starting a ROSIN funded project [21] that would work to provide an open source ROS driver that will provide full access to UR robots features [53] and will provide a reference, open source implementation with full implementation of programmatic access to the UR robot range. It will even enable these robots to be used without the polyscope needing to be attached at all. This work is expected to be completed by Winter 2019, and will potentially benefit this project.

## 3.3 Controller Interfaces

For the initial control interface I looked at musical interfaces, as there is a long tradition there of developing novel hardware to enable the triggering and manipulation of sounds, and interfacing these with computers [39], from keyboards being used to play individual notes, to drum pads and control surfaces to manipulate other parameters while composing or performing. The existence of MIDI [11], which originally enabled inter-device communication, but was soon interfaced to computers to also enable computational control and response has led to a large ecosystem of devices and software that can interoperate and build complex musical performance systems.

In addition there are a wide range of ways to interface sensors with microcontrollers to computers to prototype hardware controllers, with the Arduino driven ecosytem targeting artists and designers specifically over the last fourteen years.

A grid of buttons seems an appropriate starting point, these were originally available commercially as drum pads, like Linn 9000 drum machine in the 1980s [40]. These were then adopted by companies like Akai, who developed an instrument with Roger Linn. The advantage of these are that they are cheap, have no moving parts so are more robust, and can produce velocity data about how hard they are hit or pressed, as well as when. They are good for triggering sounds, but provide no visual feedback about their 'state'. In 2006 monome [43] started producing a

device that had a grid of buttons that were illuminated by LEDs. This meant you could get visual feedback as well as tactile feedback. The design was deliberately minimalist, with just a plain grid of buttons, lit by single colour LEDS. The behaviour of the lights was not directly driven by the pressing of the buttons. Messages were sent to the computer that the device was plugged into when buttons were pressed, and messages sent back to the device controlled the lights on it. It came with example code for using it (primarily with Max/MSP a visual coding environment originally developed for music synthesis [23], but was primarily aimed at people who could write code and were interested in developing novel instrument systems. As a boutique electronics company the device was produced in small numbers, which sold out quickly as each batch was made, and was expensive. Despite the small numbers sold this was a highly influential device and with the release of tenori-on the following year, from the mainstream technology company Yamaha [72] interest in these kind of interfaces grew.

With the continued success of the Ableton Live, with its grid based approach to clip triggering, it was inevitable that this kind of controller would be produced that could be used with Live. The ability to map midi notes to triggering clips in Live made it easy to get any of these devices to control it if they produced midi data. More companies have now produced controllers specifically to work with Live, making it possible to use this software with only occasional recourse to the mouse or keyboard. The original minimal monome had (and continues to have) single colour leds, but they ones that are aimed more specifically at using with Live have red/green, or full RGB leds, enabling them to be used in many different configurations with the colours changing to indicate what will happen if you press them, or what is currently happening in the software and include midi mappings that already align with elements in Live, or have example documents to load that are already mapped to work well.

The Novation Launchpad range of controllers [28] are a family of controllers that range in sophistication and price. The basic models provide midi messages based on the keys being pressed, and controlling the lights by sending appropriate MIDI messages. The top of the range Launchpad Pro has full RGB LEDs and a comprehensive API documented in the programmers reference that is available from their website.[71]. It has a number of ready made configurations that can be switched between from the controller or programmatically. These are called modes and set the lights to reflect the current state of different parts of the Live interface, and to control elements in Live to carry out common tasks, like triggering clips or setting channel volume levels for example.

When using the Launchpad Pro with Live the 8 by 8 grid of square buttons can be mapped to the grid of clips that can be played - with each column representing

an audio channel. The software only allows one clip to be playing in a channel at a time, so triggering another clip in a vertical column where one is already playing causes that one to be stopped, and the new one to be started (there are options to set whether the new clip plays immediately, or the changeover waits until the next beat or bar to fall). The buttons flash to indicate that a particular clip is playing, and flash in a different colour if they have been triggered but are waiting for the fall of the beat or bar to start. Individual clips can be set in Live to play once when they are triggered, or to loop until told to stop, or another clip is triggered in that channel. Clips can be playing in all the columns, and the circular buttons around the edges can be used to trigger a row of clips to start simultaneously, or to change behaviour at a program level (speed up, slow down, switch view etc..). Some of these behaviours are things we want to emulate in the system, but others (like having multiple channels) don't map to the context of a single robot arm.

# Chapter 4

# The System

## 4.1 Proposal

In the first instance the project considered producing a system that would enable the robot arm to be used in a rehearsal context to explore what the capabilities of the robot arm, in terms of position and movement. The aim is to enable someone to interact with the robot arm in a playful way - trying things out rather than knowing what will happen, with minimal rules about what should and shouldn't be done. Systems that track people and produce things in response to this were felt to to be too complex for this initial interaction and exploration, as it is primarily aimed at the person finding out what the robot can do. As there is no clear goal (in terms of the task for the robot to perform) it is difficult to know what to observe and use, and this would not play to the unique capabilities of collaborative robots -the possibility of moving in close proximity to the robot (which would making tracking more difficult), and the opportunity to physically manipulate the robot by pushing and pulling it into positions.

For these reasons the primary mechanism for interacting with the robot will be by moving it around physically - it will not be possible to "jog" the robot to a new position with a controller - to do that you must get hold of it and move with it. We want people to be able to find positions for the robot to move to through this process and provide a very quick and simple way to store those, and to be able get it to move back to these positions. An appropriate initial interface for this is the grid of buttons. We want them to be able to try positions, and decide whether they should be used as an active engaged process - making decisions about what gets recorded and what doesn't, building up a series of positions to form a movement sequence, and then get that sequence to play back.

So we we will use the metaphor that Live does in its clip view - of buttons having data in them that we want to trigger - but for this there can only be one that is active at a time (unlike Live that works with sound, where multiple and layer them on top of each other to produce multitrack music). There will be no global edit/play switch, you can edit while playing.

The initial functionality that must be implemented:

- assign the current robot position to a button
- move the robot to the position stored in the button, or if more than one move sequentially through the positions loop through the assigned positions continuously
- delete the positions assigned to a button
- appropriate visual feedback on the Launchpad to reflect the actions that have been, or are being carried out

Desirable functionality:

- saving the positions recorded in a session, and reloading positions from a previous session

As the positions we are assigning will all be valid positions for the robot to move to (they must be, as the robot is currently in that position) there is no need to check that what is being proposed is correct, or ask for confirmation that the action is correct and should be carried out as many of the other interfaces aimed at more task driven activities do - we want simple quick actions to perform all these tasks. In improvisation "mistakes" are simply a change in the path being followed, the important thing is not to break the flow of the process by popping up lots of helpful checks before taking action.

Robot Operating System will be used to provide an infrastructure to work within, enabling modular elements to be built, tested and integrated into the system, and the Launchpad Pro will be used device for operating the system. A computer will be used to run ROS, but once it is set up and ready to be used the people working with the robot should not need to use the computer interface at all - all operations will be carried out through the robot arm, and Launchpad Pro

## 4.2 Implementation

### 4.2.1 First Prototype

#### Launchpad PRO

To produce behaviour that follows the Live paradigm we will associate the 8 by 8 grid of buttons with data about robot positions that the robot can move to. The button needs to show whether that button has content that can be used - so it should light up once at least one robot position has been assigned to it. The content alters what processes are appropriate, and what will happen. If there is just one position associated with the button then moving to that position makes sense, but getting it to loop would not make any difference - it only has one position, so it would not do anything once it got there. So there should be different colours to represent how many positions are associated with that button. As we need modal behaviour for the button (you need to select whether to change or use data when you press the button), we need a way to indicate what action should be performed when a button is selected. We will use the circular buttons around the edge for that purpose. Holding those down will be like using the modifier keys on a computer keyboard (shift, control etc..) and will alter the behaviour of the program when the selected button is pressed. To keep the system simple to start with the rows and columns will be used as four buttons - the left and right columns, and the top and bottom rows each producing a different behaviour. This is primarily for two reasons. The first is to keep the system simple and easier for people to remember what buttons do what, the second is that this system may be used by a single performer/operator, so they are moving between moving with the robot and pressing keys on the controller. The modifier key is a secondary consideration over which button to trigger, and it was felt that finding a single correct one in a row of other buttons would make it fiddly and error prone for people to operate.

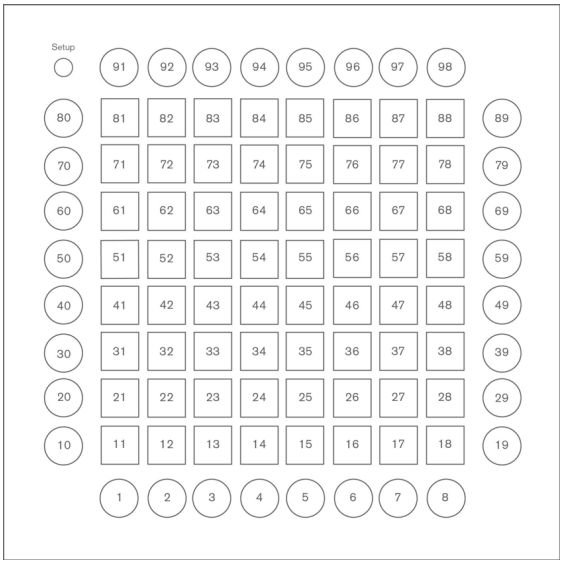


Figure 4.1: Launchpad button mappings in programming mode [71]. Circular buttons use midi control messages, square buttons use note messages

All of the Live specific modes were too prescriptive for our use, but there is a generic programmers mode we could use that just enables midi access to the buttons and lights. This sets all the lights in the buttons to off, and assigns the MIDI mapping in Figure 4.1 to the buttons. This means that when the buttons are pressed this note on/off or control change message is produced, and sent out over USB. In addition MIDI messages can be sent to the device using those note values to control the light in the button. To set the Launchpad to this programmer mode a series of button presses can be used on the device itself, or sending it the MIDI message 0xF0 0x00 0x20 0x29 0x02 0x10 0x2C 0x03 0xF7.

A ROS node, ros_launchpad, was written to interface with the Launchpad. Python was used, with the MIDO library providing midi functionality to communicate with the launchpad [69]. A Button class [66] was created that encapsulates the different colour states for the buttons, and sending the midi messages to set the button to that colour. The Launchpad class [67] uses that to build a list of buttons on the launchpad, and contains the functionality for initiating various generic actions aimed at a robot arm by publishing to various ROS topics, subscribing to topics to get feedback about them completing, and setting the button state based on this. The node creates a launchpad object, advertises itself on the ROS system, connects the the launch pad and sets it to it programming mode. It processes midi data that is sent from the launchpad as buttons are activated and has the logic code for working out what should be done based on that and the data held in the launchpad object.

#### Sawyer Arm

The initial implementation used the Sawyer robot, using the ROS interface to build the system we needed. The Sawyer git repository [49] had examples of code to control the robot in various ways. Building on these a node was created [65] to build the basic functionality that could assign joint positions to lists and play them back. ROS topics linked this to the Launchpad node that had been created, and we had a basic demo of the baseline functionality. This provided the ability to store positions in buttons, adding to positions multiple times to build up sequences, and play them back, and delete sequence that were no longer needed.

Unfortunately just after this was completed the robot was physically damaged while being transported to an event, and the week after that Rethink Robotics closed, and so getting the robot externally repaired was not an option. We disassembled and replaced the damaged connector on the robot, but at this point it was clear that other internal damage had been sustained and we did not have the information to troubleshoot that further. Development using Sawyer was no longer practical. While we could run a simulation of Sawyer to test the functioning of code, that negates the whole purpose of the project, and there seemed little chance of replacing or repairing Sawyer.

### 4.2.2 Second Protoype

#### UR10 Arm

Following the purchase of two UR10 arms the work now was to adapt the code to work with the UR10 arm instead. As we had used ROS and put the code for the launchpad in its own node there was no need to work on this code in the first instance to use a new robot, the work would need to be done on the node that interfaced with the robot. From the description above it is clear there were a number of routes to achieve this, as the UR10 robot is not inherently ROS based.

As the UR10 is a bare arm with no controls on it, and operating freedrive with the pendant is not easy, a more user friendly end effector needed to be added to make it possible to initiate freedrive from the arm, and make it easier to physically move the robot once it was in freedrive mode. Initial research was done into how best to control the robot and interface it into ROS to use the existing launchpad node.

The basic options were:

1. Writing a ros node that interfaces directly to the robot using a python socket to send URScripts to the robot. This was rejected as while it enables us to control

the robot, we would need to attend to a lower level of detail then we would like to, and would be re-implementing things that had already been done by the ros industrial package to integrate this into ROS. While it is not required that we use ROS we would end up needing to replicate at least some of its functionality (for example inter application communication) in order to build a larger system, so it was decided to continue along the ROS route.

2. Using the Ros Industrial ur_modern_driver to control the robot, using ros_control and either implementing our own trajectories or tools like MoveIt to produce them This offers the most flexibility, in terms of creating trajectories for the robot, but overcomplicates things too early in the project. MoveIt works with the robot, and introduces potentially useful features like collision avoidance, but it is too narrow in its use cases to be directly useable. It currently only works planning for a single target - a series of moves would need to be implemented as a series of calls to MoveIt - waiting for each one to complete before sending the next . This produces a short pause between each move that cannot be removed, and means we cannot have a blend between each point on the trajectory (having a smooth movement that slightly cuts the corner for intermediate positions). Also because the MoveIt package is using more sophisticated path planning techniques that are not simply interpolating between the joint positions it means that at times the generation of a trajectory fails even when there is a valid route. There always will be a valid route, as all the points that we are moving too are valid poses, and we want to move in joint space, not cartesian space - we currently want the robots more 'natural' movement between waypoints, rather than a specific tool path.

3. Using ROS Industrial ur_modern_driver to control the driver, sending URScript programmes to that node to forward on to the robot This is the most productive route for a first prototype. Sending all the waypoints in a movement as a defined function in URScript that is immediately executed enables smooth movement between positions as it executes the path. In addition to that the ur_modern_driver provides data on ROS topics including the current robot and joint states. While the new ROS driver became available in very early beta form, it is not yet ready to use for this project, but should provide enhanced functionality by switching to it once it is more developed.

So the option that was chosen was number three, generating URScript text, and sending it to the ur_modern_driver node.

A Waypoints class was created. This contains a list to store waypoints in, and currently has four functions add_waypoint, generate_trajectory_script, generate_palindrome_script, clear_waypoints.

add_waypoint appends whatever is passed to the function to the end of the way-

points list. When it is called in the main program a list of the six joints state are passed to it.

generate_trajectory_script iterates through the elements that are stored in the waypoints list (if any), and generates ASCII text that is valid URScript based on the waypoints stored in list. The movej command from URScript is used to move the robot. This takes a list of six values to move the joints to, and four possible values to control maximum rotational acceleration, maximum rotational speed, target time for the movement, and blend to be used at intermediate waypoints. If a time is set then the acceleration and velocity parameters are ignored.

passing a list that consists of these values:

[[-0.833, -2.240, -1.316, -2.484, -1.088, 0.000],
[-0.833, -1.951, -1.173, -2.229, -1.088, 0.000],
[-1.205, -1.966, -1.818, -2.411, -1.529, 0.000]]

will produce this output:

def myProg():
movej([-0.833, -2.240, -1.316, -2.4842, -1.088, 0.000],a=1.4,v=1.05,t=0,r=0)
movej([-0.833, -1.951, -1.173, -2.229, -1.088, 0.000],a=1.4,v=1.05,t=0,r=0)
movej([-1.205, -1.966, -1.818, -2.411, -1.529, 0.000],a=1.4,v=1.05,t=0,r=0) end

(note these values have been shortened for readability, the actual values are between 14-20 decimal places)

Launch files were created to simplify running the nodes necessary, configured correctly, for the system to work. ROS has a visualisation tool called rqt_graph, which produces a graphical representation of the running system. The system is shown in Figure 4.2. The ovals are the nodes that are running, and the rectangles show topics that nodes are either publishing or subscribing to. Publishing is indicated by an arrow pointing to the topic from the node, subscribing by an arrow pointing to the node from the topic. Where a topic is being both published and subscribed to then there is communication happening between the nodes on that topic, in the direction of the arrows. Where topics are only being published or subscribed to then these topics are not currently actively involved in the functionality of the system, but indicate what topics are available for adding functionality.

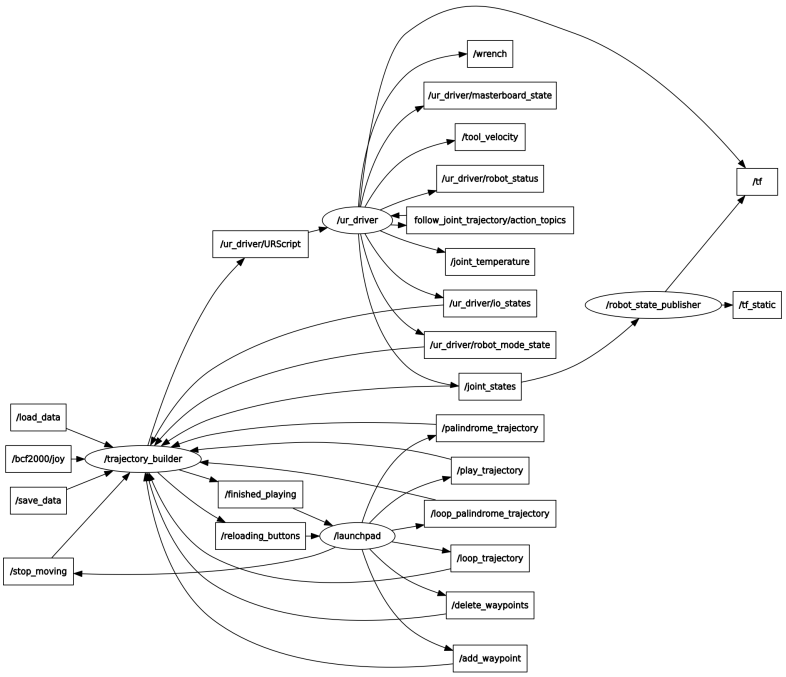


Figure 4.2: rqt_graph visualisation of the running system

### The end effector

To make it easier to move the robot when in freedrive the robot needs an end effector that is easy to get hold of and enables you to apply enough force to pull or push the robot around more easily. A commercially available hand wheel looked like a good option and a model was found that was a good size for getting hold of and was small enough to not collide with robot itself as the wrist joints moved around. To mount it on the end of the arm it was attached to a bolt, and an aluminium plate was designed to fit the fixing points on the end of the arm and take the bolt. This was cut on a water jet cutter in the workshop and attached to the end of the arm. A water jet cutter is similar to a laser cutter, with an x-y moving head, but instead of using a laser it produces a narrow, high speed jet of water that has had fine grit added. This will cut cleanly through materials that cannot be cut using a laser cutter, including metal and stone. This provided good physical affordance enabling pulling and pushing to be done much more effectively.

The desired behaviour is that when someone gets hold of the handwheel, freedrive is enabled automatically. Rather than using a specific switch that somebody needs to press as they hold the handle, it would be better to detect whether someone is touching the handwheel and respond to that. In order to do that we need some

mechanism to detect touch.

There are a number of ways that this can be done electronically:

1. Two generic input/output pins on a microcontroller can be used. One of the pins is configured as an output, the other as an input. Connecting a high resistance between the pins (in the range of 100K and 50M ohm) and attaching the touch element (a conductive surface like tin foil) to the input pin creates an RC circuit that will effect the time it takes for the voltage on the input pin to rise once the output pin is set to HIGH. The time is dependent on the resistance used (the R element), and the capacitance of the area between the end of the resistor and the input pin (including the foil). This capacitance will change as different things are brought near and touch the foil, including human body parts. The system is calibrated by taking readings when nothing is near the foil, and readings when someone approaches and touches the foil. A suitable threshold is chosen to avoid false positive readings (the values fluctuate quite a lot), while being sensitive enough to respond to close proximity and/or touch. You can improve the reliability by adding a small capacitor in parallel

While this system can work, and there is a dedicated Arduino library that implements it [13], I have used this in previous work and found it to be very sensitive to changes in conditions and had difficulty getting it to work reliably.

2. Dedicated chips that are designed to implement this functionality. There are a number of small breakout boards that use dedicated touch sensing chips that produce a digital signal that indicates whether something is touched, usually using the Microchip AT42QT1010 chip [42]. A range of breakout boards were tried, but they were all configured to detect a tap rather than the prolonged holding down, and recalibrated themselves automatically after around a second. This meant they turned off even though they were still being touched. They would not be suitable as moving the robot around could take a significant time - almost always more then a second, and often into 10-20 seconds. One board was sourced that described itself as continuous touch [59], but even that one timed out between six and ten seconds in the configuration being used with the handwheel, so was not useable either.

3. Using a microcontroller that includes dedicated touch functionality on some of its pins. There are a number of microcontrollers that include this - a commonly used one being the Microchip ATSAMD21G18 [41], which is an Arm Cortex M0 chip which has a Peripheral Touch Controller that implements sophisticated touch algorithms. This is available on many branded boards sold by various companies including the Adafruit feather, Arduino Nano. Proof of concept was tested with an Adafruit feather M0 [4] and testing confirmed that the touch functionality

would function correctly, with a sustained touch registering correctly for as long as was felt to be necessary (at least 120 seconds of unbroken holding without any problems).

Once the handwheel had been selected and the mounting system for it had devised it was decided that the electronics should be integrated into the back of the two support arms of the hand wheel in order to keep the hand wheel close to end of the robot and the electronics clear of people touching them. This meant the area available either side was 18mm by by 31mm (although you could allow something to protrude beyond 31mm a bit, as long as it was fairly flat). The Adafruit feather board is 23mm wide, which was too big to fit flat in this area and mounting it perpendicularly would make it prone to be hit as people put their hand through to hold the wheel. A much smaller board that appeared to be suitable was tried next - the Adafruit Trinket M0 [7], at 27mm by 15mm fit easily in the space. It uses the very similar ATSAMD21E18, which also has the Peripheral Touch Controller (PTC). Unfortunately the Adafruit library [5] that works with the PTC only ran for just under a minute and then stopped reporting values, something had caused the code to freeze. It was unclear why this was happening, and as the library is working at a very low level to configure things it was difficult to see what might be changed to fix this. The very small size of the board also means there are only five IO pins, and this may prove to be too limiting as we developed more functionality in the handle, so another board was tried. The Teensy 3.2 [60] uses a Cortex M4 chip, the NXP MK20DX256VLH7. This includes touch enabled pins, and the software libraries available with the teensy add-on for Arduino includes support for using them. The detection of touch was reliable, and there is optimised code for controlling Neopixels that use direct memory access, which enables the movement of data to take place without blocking other operations. At 17.78mm by 35.56mm the board is the right width, and is only protruding by a few millimetres from the support arm on the handwheel, so this board was used.

To detect touch all around the handle conductive tape was used. Nylon conductive tape [3] works well, as it does not have the sharp edges that foil tapes do, but it does fray a bit over time with handling, so would need to be replaced occasionally, or a suitable covering found (as capacitive touch will work through a non conducting barrier).

In order to indicate when freedrive was activated a strip of Adafruit Neopixels were added. These are RGB leds that each have a microcontroller embedded in the LED, enabling a single digital pin to control a large number of LEDs using a single pin - the constraint being the memory limit on the microcontroller used for generating the data to control that many LEDs, and the power they would require to operate. The early prototype used traditional neopixel strips around the inside

edge of the handwheel. The original neopixels were replaced with side mounted LEDs as the original ones were not very visible as they faced in to the centre of the handwheel, rather than out towards the person. The 90 led/metre strips were used as this gave a good density of lights, with 27 fitting neatly around the inside edge of the handle. These are currently held in place with double sided tape. This is not a long term solution, but it has not been possible yet to find something that will stick to the silicon sleeve that the neopixels are enclosed in, and the material the hand wheel is made of. The default state of the LEDs was set to red, turning to green when the wheel was touched.

Another iteration of the design would be needed to address the long term issues with the nylon conductive tape and attaching the neopixels more securely.

The UR10 has a port by the end effector for interfacing with the controller. It has power, that can be set to operate at 12 or 24 volts, two configurable digital inputs that can be configured to trigger actions, and read from URScript code, and two configurable outputs that be driven from URScript code. In addition the input/output can be configured to drive the controller or responded to the state of the controller without further intervention from code.

This port could be used to power the Teensy, and enable it to signal to the controller when the handle is touched.

A small step down power supply was identified that would fit on the other handwheel arm, that works with an input voltage 7.5 - 36 volts. This means it would work whichever voltage was selected without the device breaking, or failing to operate if the 'wrong' voltage was selected. The output voltage is 5 volts, in order to power the neopixels. The teensy is a 3.3 volt board, but will accept a 5 volt power supply on its Vin pin to power the board, and has a trace that you can cut to isolate the teensy from the 5V in from the usb socket, to prevent back feeding the usb connection on the computer.

To interface the teensy pins to the UR10s inputs and outputs optocouplers were used. An additional diode was added to reduce the voltage supplied to the neopixels to ensure that the 3.3 volt signal from the Teensy would drive them reliably. The circuit was built on breadboard to test functionality. Once it was confirmed that the circuit worked a version was built on veroboard for testing and developing the system.

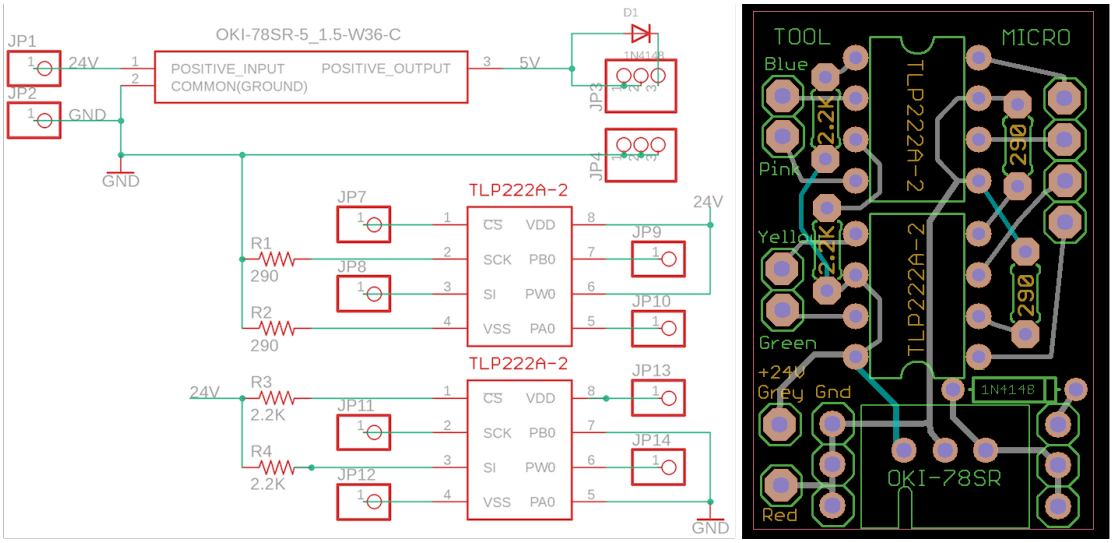


Figure 4.3: Interface circuit schematic and PCB design

From the circuit that had been tested a printed circuit board was designed by my collegue Michael Margolis, to produce a neater, more compact and reliable solution. This was sent off to an external printed circuit board fabrication service, and replaced the veroboard version when it arrived.

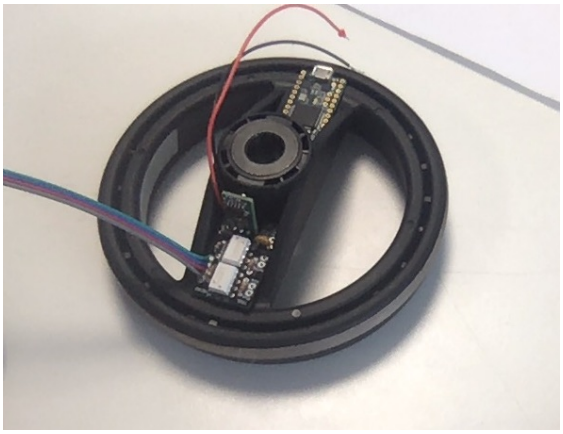

Figure 4.4: Interface PCB and Teensy in position

The code on the teensy [70] has a startup routine that takes one hundred readings of the touch pin when starting up, and calculates the average of them. It then adds an offset to enable it to detect the change when someone touches the wheel. The offset was reduced to make the detection more sensitive, until it detects just before the touch happens. This is so that the UR10 is definitely in freedrive mode before any force is applied, as the it generates an error and stops the robot if force is already being applied as it enters freedrive mode.

One of the UR10 inputs is configured to enable freedrive mode, so writing HIGH to the pin on the Teensy that is connected to the appropriate optocoupler puts the UR10 into freedrive. In addition a UR10 output can be set to follow whether a URScript is running. Reading the appropriate digital input on the Teensy enables us to set the lights to a different colour to indicate that the robot is carrying out a task. Blue was chosen to give strong differentiation between the three states.

## 4.3 Final Functionality

Once all the coding and physical fabrication was completed the following functionality was available:

### 4.3.1 End effector

Handwheel end effector for interacting with the robot, with ring of leds to indicate state - red for ready but inactive, green for touched and in freedrive mode, blue for code running moving the robot.

Touching the handwheel at any point when the arm is moving will cause it to stop. Touching the handwheel when the arm is stationary will engage freedrive, until the handle is released.

### 4.3.2 Launchpad

Hold down right modifier button on Launchpad and press any square button - current position is added to that button and colour of button altered to reflect that Press any square button that is lit - initiate moving to that position, or through that series of positions, once. Flash button while arm is moving, stopping once final position is reached. If the arm is currently moving stop that movement first, and update lights to reflect that. If the position is the one it is moving too just stop action Hold down bottom modifier buttons and press a square button - if only one position stored just move to the position, if multiple positions stored loop over them continuously until another action is selected, or that one is pressed again. Hold down right modifier buttons and press a square button - similar behaviour to loop, but iterate through multiple positions palindromically (backwards and forwards through the list). Hold down the top row of modifier buttons and press a square button - delete any waypoints stored in this button. Note that this currently doesn't stop any movement that is going on - if that button is currently looping it will continue to do that as the positions have already been read from the list. Once that movement is stopped though the positions will no longer be available.

### 4.3.3 Additional functionality that was not tested

#### Saving the stored waypoints as a text file

Saving and reloading settings from a text file was implemented in the node ur_trajectory_generator, so that once interesting moves had been found they could be saved and reused. The node subscribes to the topics save_data and load_data. Publishing a String of a name to the save_data topic saves all the current data stored in buttons to a JSON document with that name in the current home folder. Publishing a name on the load_data topic attempts to open a document of that name in the home folder and restore the values in it to the correct positions, and set the lights on the launchpad appropriately. As this was only implemented at the ROS topic interface level it was not tested with end users.

#### Changing the global speed of the arm during execution of a sequence

The polyscope on the UR10 has a GUI slider element that controls the speed of the arm in realtime , between 0-100%. There is a URScript command that can set the position of this slider. By sending that command as a single line of URScript code (rather than as a defined function) the speed of the arm can be changed without stopping a function that has been sent to the arm. This is implemented in the UR_trajectory_generator node, and has been controlled by a slider on an additional midi interface. In addition the max acceleration and velocity of joints can set for each URScript movej command. This was also connected up to two additional sliders on the midi interface, but this was not used in the tests as the midi device has many additional sliders and rotary encoders, and most of them were not being used, so a more appropriate device would be found before using this functionality with end users.

#### Multiple Arms

As we have two UR10s I created a launch file to run two robots, using one with the handle to programme positions, both of them to respond to the generated UrScript commands and move in sync. In order to do this two ur_modern_driver nodes are launched in separate namespaces (Left and Right), see Figure 4.5, each configured with the IP address of a different robot. The ros_launchpad and ur_trajectory_generator nodes were run outside of these namespaces and the relay node from the topic_tools package was use to subscribe to the URScript topic and republish it inside each namespace so that each instance of the driver receives the same instructions. This configuration was not used as part of the test, but was demonstrated to the participants after they had completed the test.

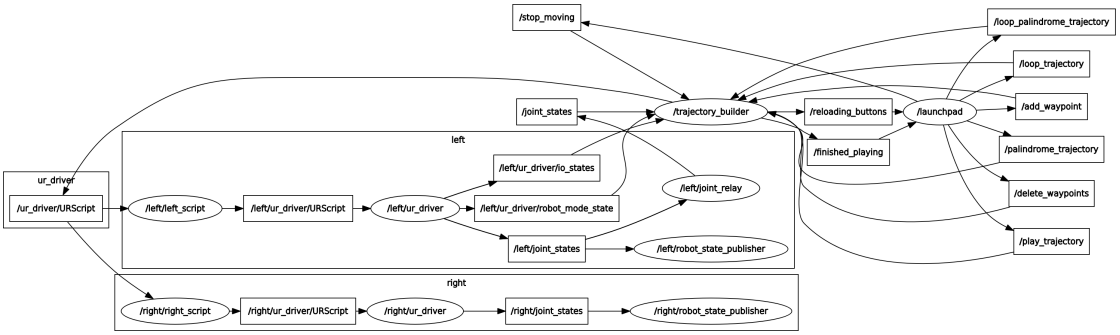


Figure 4.5: System running two arms

# Chapter 5

# Testing

## 5.1 Testing

Using the written instructions included in the Appendix I ran some user tests to see if the system was understandable and easy to use. People were given the sheet and asked to work through it. The sheet starts by giving a brief description of the behaviour of the handle/robot combination and then introduces the Launchpad and provides instructions to add a position to a button, and then another position to a different button. It then asks what happens when you do various actions, to encourage people to explore and try things out rather than performing a series of prescribed actions.

The tests were carried out with four different people. An MSc Robotics student, two children, and a member of staff.

While carrying out these tests the the Launchpad had been placed outside of the working area of the robot arm, so that as people tried out what the buttons did we did not need to consider where the movements would take the robot arm (they would not be hit).

The first test was the student. They were able to complete the steps. While they were using it I made a number of observations. They were able to add a second waypoint to a button, and commented on the change of colour reflecting this, but on adding the third the colour of the button did not change. This was the correct current functionality - the button colour changed between one and two waypoints, but did not continue to change as the number increased. They stopped at this point and were unsure whether the action had succeeded or not. They were unsure whether to add it again, but after considering for a moment they pressed

the button to move the robot to the waypoints, and were able to deduce that the third point had been successfully added, but their flow had been interrupted.

They understood the delete functionality, and asked about the possibility of deleting elements in a list, rather than the whole list.

While they were using the system I observed a bug that I had not picked up before. When running a palindromic loop, pressing the same button to do a straight loop did not produce the correct changes on the lights, although the robot behaved as expected. The button stopped flashing, but the robot now looped as the button press had initiated, and would not stop until you pressed the button again, or pressed another button. It was not clear what had happened at this point and caused confusion. By now they were more confident about quickly pressing buttons to get things done, and so moved on from the situation by pressing a different button that held a single waypoint, which moved the system back into a correct state, the lights reflecting the current action..

The first child was able to follow the instructions and draw the correct conclusions about the systems behaviour, particularly in terms of the difference between the two looping behaviours. They read the text descriptions on the buttons to try and work out what action they would produce. While they made sensible choices based on them, those descriptions relate to using the device with Live, and do not map to this system at all - I had learnt to ignore what was written on them. They observed at the end that it would be good if the written instruction had explained that the text on the buttons should be ignored.

Following this their younger sibling, who had watched from a distance while this had been happening, asked to try it. They were able to operate the system based on observing it having been used and skimming the document for some prompts at first.

The final test was a member of staff. They were able to follow the instructions and successfully identify what was happening. They experimented with how you assign positions to buttons - trying things like pressing buttons to assign positions while the arm was moving between previously assigned positions, and were pleased that all the permutations they tried behaved as they expected.

Following this final test the children asked if they could have another go, and spent time playing around with the arm.

## 5.2 Improvements Implemented From Testing

Following the first test I immediately implemented continuing to change the colour of a button as the number of of waypoints stored in it increased, as a sinlge colour change to indicate the difference between a button having one position or multiple did not provide enough feedback. A list of colours was created, and the number of waypoints stored was used as the index to retrieve the colour to use (with a test for the end of the list to avoid errors). This makes it easy to extend the number of colours to be as long as feels sensible as the system is used, and to change the colours easily. In addition to making the user more confident that their actions are having the desired effect, it will also introduce more visual difference between the buttons, making it easier to recognise where things are stored. In addition the logic was updated to fix the bug that had been identified.

Following this when the system was tested with the two children it was clear that the colour changing each time you added a waypoint helped make the system more readable when first starting.

Another observation I made, was that as the robot arm was programmed to move in a wider range of movements the operators were approaching the Launchpad form different directions, rather than sitting in front of it and staying in the same orientation in relation to it. Because it is square the orientation was not obvious and working out which of the edge buttons to use to make things happened took a bit of working out. Another change was made to improve the interface based on these observations. The four edge banks of buttons were set to different colours, and to be pale versions of those colours when not pressed. This helps to suggest which buttons to press to get things to happen when first using the device, but also to identify what each one will do when pressed, and makes it easier to clearly see the orientation of the launchpad as you approach it. In addition the surface of the buttons was painted black to obscure the misleading labels (the text was on a black background so that it lit up when the button was illuminated, so painting over it was the neatest way to remove it.

# Chapter 6

# Conclusions

## 6.1 Discussion

The system that was developed has met the original goals. The robot can be programmed quickly and easily to move to multiple positions, by directly manipulating the robot and using the launchpad interface to save, play loop and delete elements, without needing to use the computer screen or polyscope after the initial set up. When the final version of the new ROS driver is completed it should be possible to make the polyscope unnecessary, which would probably be desirable in a performance situation.

The system met all the initial proposals in terms of functionality, and developed additional functionality as well (saving/loading, changing global speed of movement, working with multiple robots). I did not introduce this to the users who tested the system as I could not see a simple way to implement user access to them through the launchpad and I did not want to add screen based interaction to the system. This is one of the areas for future work. The people who used it were able to perform the tasks that were specified in the tests, and make predictions about how the system would behave and test those predictions out - this is important for people to feel empowered to attempt to use the system in ways that they want to, rather than in ways they have been told. Various playful ways of using the controller have emerged - for example triggering moves in time to music without waiting for the previous move to complete enables jittery dance like moves in time to music without laborious coding.

The system has not been tested with any dancers yet, due to the delay caused by the damage to the Sawyer arm.

## 6.2 Further Work

Following this initial development and testing cycle there are a number of areas that it is clear would already benefit form further development, and opportunities for additional exploration.

### Refactor existing code

The current implementation works, but refactoring the code would make it more robust and easier to maintain and extend. Currently the logic about what happens when a button is pressed is split between the two nodes that have been written. This has led to maintaining state data in each node that is related, and needs additional topics to ensure it stays in sync. This makes the code harder to understand and less modular - changing the controller would mean we would need to duplicate some of the logic about what happens into any node we create for interfacing with alternative controllers. It feels like it would be better to have the launchpad node just publishing the data about button presses from the launchpad into ROS, and subscribing to a message that controls the leds. The logic about what to do should all be in one place. The ur_trajectory_node is a better place for that as it is the one that should be dealing with all the specific things that relate to the particular task that has been developed. In addition the colours of the buttons are specified using the Launchpads inbuilt system with 127 colours that use fairly arbitrary values, rather than RGB. This is implemented in the API to reduce the amount of data that needs to be sent over the relatively slow midi connection when changing the colours of lots of buttons. The use of those values in the code that was written is inconsistent, and sometimes just appears as magic numbers in code. When clearing up where the logical decisions are made, a more uniform system for specifying the colours should be implemented.

### Handle

The current method of attaching the LEDs to the hand wheel is a temporary method, and the conductive tape for detecting touch is prone to wear. The silicon sleeve that the neopixels are in is resistant to most glues, and the few that do work do not adhere to the handle well. Removing the LEDs from the silicon sleeve is an option, but then an alternative protective cover would be needed. Large clear heat shrink has been used in previous work with neopixels to encapsulate them in more robust material, and worked well, but the closed circle of the handle prevents this method being used. A metal handwheel was considered to enable touch detection (due to it conducting), but its other physical properties were not good - heavy, cold and hard. I think the best solution would be to 3D print a

handle, embedding conductive elements inside it during the printing process, and including a rebate for accommodating the LEDs to make them less exposed so they could be attached without needing the silicon, or the strength of the glues available would be adequate as they would be less exposed. The MarkForge carbon fibre 3D printer that we have at the university would be able to print something that would be strong enough to replace the handwheel currently in use as it use a nylon / carbon fibre mix.

Currently two of the available UR10 IO pins are used for communication, the other two are still available for additional functionality. Once they are used then there are no more available, and none of them support more complex communication like serial data. It would be better aesthetically, and practically to not add cables down the arm, so radio would be a good option. This was originally considered for the handle, but implementing the core functionality that includes stopping the arm if someone touches it is better done through the more robust and deterministic route of a wired connection. Adding a radio unit would enable more sophisticated use of the LEDs on the handle, which in a performance situation may be desirable. The recently released Arduino Nano 33 IOT [10] will fit in the space available on the current wheel. It uses the ATSAMD21G18 which has the touch functionality, and includes an additional wifi radio module. Another option that would physically fit is the pycom board, that uses the esp32, a sophisticated module that includes processors and wifi module. A simple protocol for transmitting data to set the values of each led independently has been devised and tested for the nano33 [68], but deployment to the arm has not yet been carried out. When 3d printing a new handwheel additional protection for the embedded electronics can also be incorporated

### Editing Sequences

The current editing system is very quick and easy, but has limited functionality. It would be good to be able to edit a sequence in a more granular way than just deleting a sequence in its entirety. The ability to delete the last element in a list would be useful - especially if you realise you have just added a waypoint to the wrong location. It would also be useful to be able to step through the series of waypoints moving from one point to the next, waiting at each position until a 'next' button is pressed, with the option of deleting the current position the arm is in from the sequence. It would also be useful to be able to move the arm manually at that point and insert the new position at that position in the list, rather than the end. At this point if we continue to just use the Launchpad as the interface we would need to move away from using the edge buttons in banks of eight all performing the same action, and/or use some of the square buttons for other

functionality. Another way to implement some of those behaviours would be to enable playing individual buttons in sequence between two buttons that you press (from button 4-7 for example). Alternatively we could use a separate interface that enable this functionality displaying how many waypoints in a sequence, with forward, back delete and and add buttons. This is a classic dilemma for simple systems, and there is a danger of implementing lots of additional functionality quickly making the system too complicated to use and losing sight of the original purpose.

### Playing back waypoints

While the current interface is quick to use it is difficult to remember what has been stored in which buttons, and there is no way to find out other than getting the arm to move. Changing the buttons colours to reflect how many waypoints in has helped by making the interface more varied, but it is still fairly limited. It would be good to investigate options for visualising what is in buttons more fully. MoveIt has a mechanism for displaying the calculated trajectory in Rviz as a ghost-like paler animation over the visualisation of the current static position of the arm - it may help to have this kind of visualisation available for the buttons, or an image of all the arm positions in a sequence superimposed in a single image.

### Varying speed of movement

Basic programmatic access to global speed control has been implemented, but it is not clear what would be good interface for controlling this - it is important that the speed is not changed too quickly as this causes the robot to judder as it may involve fast accelerations or braking to conform to the change, and this can cause the conformal safety limiter to step in and stop the robot from operating until you dismiss a dialog box on the polyscope.

### Generating trajectories

As all these additional features become possible the question of using them assigned individually to waypoints or series of waypoints crops up. Acceleration and maximum speed can already be passed to the movej commands, and this functionality has been exposed as a variable that can be set in the function that generates the scripts, but it is unclear what would be a good interface to sue to do this. At this point you would probably consider using other mechanisms to generate a trajectory or movements on the robot. A new UR node for ROS is now available that has been produced in collaboration with Universal Robots. This provides new mechanisms to control the robot using ros_control [19]. The new driver also includes code to run on the polyscope that the node can communicate with to

enable features that can currently only be instigated from the polyscope. Future work would be to refactor the code to use this node and explore the additional options it provides.

### Triggering sequences in performance

The current project has looked at how you can enable improvisational activity with a robot arm in rehearsal, the next stage would be to consider how this could translate into performance, and what changes or enhancements should be added. Having the controller as a fixed physical device in one location will probably be less appropriate, and we are likely to start looking at sensing where the performer is in relation to the robot, and how their body can trigger events. This would be a large piece of work and would be a suitable task for further student project or programs of study.

### Launchpad PRO

While the Launchpad PRO has been useful as a prototyping tool, and its robust construction makes it very appropriate, it is not ideal. As other parameters are being considered for manipulation we will either need to add an additional controller (which we did for testing the speed control functionality internally), or replace it with something else.

Adafruit now produce a modular grid button kit with RGB leds that can be assembled in four by four units [6] to whatever size and layout you need. Although the buttons are considerably smaller the ability to use these and laser cut a case to include additional interface components (encoders, potentiometers) would probably be a good next step. It would be fairly simple to add a microcontroller with a radio link that was compatible with whatever was used in the handle to build an integrated radio system across the components, enabling this to be used without a cable.

### Appropriate Stands for the arms

Currently the robot arms are mounted on lab tables that are 1.2 meters square, 80cm high. This means they are physically fairly inaccessible. While this is deliberate and appropriate for traditional operation and teaching using the pendant and programming interfaces, it is less appropriate for this projects use case. It is difficult to get close to the robot to physically manipulate it, and many orientations cannot be achieved without using the jogging functionality on the polyscope as the robot is too high and far away from the operator. A pedestal base would enable people to get much closer to the robot - this would be desirable from a

practical considerations, enabling easier manipulation, and from a choreographic view as well, enabling closer robot-person interaction. Having the base of the robot 50cm from the ground would make it just higher than a person when full extended vertically, making all the possible positions available. There are commercial units available designed for the UR10, but it would also be easy to design a custom unit using aluminium profile systems, for example the item range [38]. Some of the commercially available units are just designs with a bill of materials for these profile systems.

### Multiple robot arms

The basic proof of concept of multiple arms was powerful, creating a very different feeling to a single robot operating. The speed slider on each robot could be changed to make one robot move more slowly then the other to change the relationship between the robots, with one then seeming to follow after the other. Many ideas arise from this proof of concept. Making one robot reflect the angle of some joints to make it move in more like a mirror image than an exact clone would be interesting. Adding a handle to the other arm and devising a more distributed system that enables each robot to be used to program, and the various permutations that that could support.

# Bibliography


[1] S. Calinon A. Billard and R. Dillmann. *Springer Handbook of Robotics*, chapter Learning From Humans. Number 74. Springer Publishing Company, Incorporated, 2nd edition, 2016.

[2] Ableton. Live. `https://www.ableton.com/en/live/what-is-live/`, Accessed 2018.

[3] Adafruit. Conductive tape. `https://www.adafruit.com/product/3960`, Accessed Aug 2019.

[4] Adafruit. Feather m0 board. `https://learn.adafruit.com/adafruit-feather-m0-radio-with-rfm69-packet-radio`, Accessed Aug 2019.

[5] Adafruit. Freetouch. `https://github.com/adafruit/Adafruit_FreeTouch`, Accessed Aug 2019.

[6] Adafruit. Neotrellis. `https://learn.adafruit.com/adafruit-neotrellis`, Accessed Aug 2019.

[7] Adafruit. Trinket m0. `https://learn.adafruit.com/adafruit-trinket-m0-circuitpython-arduino/overview`, Accessed Aug 2019.

[8] Thomas Timm Andersen. Optimizing the universal robots ros driver. Technical report, Technical University of Denmark, Department of Electrical Engineering, 2015.

[9] Margo K. Apostolos. Robot choreography: The paradox of robot motion. *Leonardo*, 24(5):549, 1991.

[10] Arduino. Arduino nano 33 iot. `https://store.arduino.cc/nano-33-iot`, Accessed Aug 2019.

[11] The MIDI Association. The midi associaton. https://www.midi.org, Accessed Aug 2019.

[12] Autodesk. Powermill software. https://www.autodesk.com/products/powermill/overview, 2018.

[13] Paul Badger. Capacative touch sensor library. https://github.com/PaulStoffregen/CapacitiveSensor, Accessed Aug 2019.

[14] Guha Balakrishnan, Amy Zhao, Adrian V. Dalca, Frédo Durand, and John V. Guttag. Synthesizing images of humans in unseen poses. *CoRR*, abs/1804.07739, 2018.

[15] A. Billard and D. Grollman. Robot learning by demonstration. *Scholarpedia*, 8(12):3824, 2013. revision #138061.

[16] Aude Billard. On the mechanical, cognitive and sociable facets of human compliance and their robotic counterparts. *Robotics and Autonomous Systems*, 88:157 – 164, 2017.

[17] J. Burrows. *A Choreographer's Handbook*. Taylor & Francis, 2010.

[18] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *CVPR*, 2017.

[19] Sachin Chitta, Eitan Marder-Eppstein, Wim Meeussen, Vijay Pradeep, Adolfo Rodríguez Tsouroukdissian, Jonathan Bohren, David Coleman, Bence Magyar, Gennaro Raiola, Mathias Lüdtke, and Enrique Fernández Perdomo. ros_control: A generic and simple control framework for ros. *The Journal of Open Source Software*, 2017.

[20] Mark Coniglio. Troikatronix isadora. https://troikatronix.com, 2018.

[21] ROS-I Consortium. Official ros support: making the most out of the new ur e-series - anders beck, universal robots. https://www.youtube.com/watch?v=gjrnOdymUUE, Jan 2019.

[22] PickNik Consulting. Move it planning framework. https://moveit.ros.org, Accessed Aug 2019.

[23] Cycling74. Max/msp. https://cycling74.com, Accessed 2019.

[24] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[25] Beatrice de Gelder. *Emotions and the Body*. Oxford University Press, Jan 2016.

[26] Shaun Edwards Stuart Glaser Felix Messmer, Kelsey Hawkins and Wim Meeussen. universal robots. `http://wiki.ros.org/universal_robots`, Accessed Aug 2019.

[27] Dan Trueman Fiebrink, Rebecca and Perry Cook. The wekinator: Software for using machine learning to build real-time interactive systems. `https://www.wekinator.org`, 2018.

[28] Focusrite. Novation launchpad. `https://novationmusic.com/launch/launchpad`, Accessed Aug 2019.

[29] Madeline Gannon. Quipt. `http://www.madlab.cc/quipt/`, Accessed Aug 2018.

[30] Madeline Gannon. Mimus. `https://atonaton.com/mimus`, Accessed August 2018.

[31] Petra Gemeinboeck and Rob Saunders. Movement matters: How a robot becomes body. In *Proceedings of the 4th International Conference on Movement Computing*, MOCO '17, pages 8:1–8:8, New York, NY, USA, 2017. ACM.

[32] Nicholas Gillian and Joseph A. Paradiso. The gesture recognition toolkit. *J. Mach. Learn. Res.*, 15(1):3483–3487, January 2014.

[33] Daniel Holden, Ikhsanul Habibie, Ikuo Kusajima, and Taku Komura. Fast neural style transfer for motion data. *IEEE Computer Graphics and Applications*, 37(4):42–49, 2017.

[34] Dong Hu, Xin Liu, Shujuan Peng, Bineng Zhong, and Jixiang Du. Automatic character motion style transfer via autoencoder generative model and spatio-temporal correlation mining. *Computer Vision*, pages 705–716, 2017.

[35] ROS Industrial. Ros industrial tweet. `https://twitter.com/ROSIndustrial/status/1001904831987175425`, May 2018.

[36] ROS Industrial. Ros industrial. `https://rosindustrial.org`, Accessed Aug 2019.

[37] Robots and robotic devices – collaborative robots. Standard, International Organization for Standardization, Geneva, CH, February 2016.

[38] Item. Item mb building kit system. `https://uk.item24.com/en/productworld/building-kit-system/`, Accessed Aug 2019.

[39] Kim Bjørn; Mike Metlay; Paul Nagle; Jean-Michel Jarre. *Push turn move*. BJOOKS, 2017.

[40] Roger Linn. Roger linn design. `http://www.rogerlinndesign.com/past-products-museum.html`, Accessed Aug 2019.

[41] Microchip. Atsamd21g18. `https://www.microchip.com/wwwproducts/en/ATsamd21g18`, Accessed Aug 2019.

[42] Microchip. Microchip at42qt1010. `https://www.microchip.com/wwwproducts/en/AT42qt1010`, Accessed Aug 2019.

[43] Monome. monome editions. `https://monome.org/docs/editions`.

[44] OSRF. Ros 2. `https:/index.ros.org/doc/ros2/`, Aug 2019.

[45] Rob Saunders Petra Gemeinboeck and Elizabeth Jochum, editors. *Movement That Shapes Behaviour*, 2019.

[46] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[47] Universal Robot. History of the cobots - the cobots from universal robot. `https://www.universal-robots.com/about-universal-robots/news-centre/the-history-behind-collaborative-robots-cobots/`, 2018.

[48] Rethink Robotics. Intera website. `https://www.rethinkrobotics.com/intera/`, 2018.

[49] Rethink Robotics. Rethink robotics github. `https://github.com/RethinkRobotics`, Aug 2018.

[50] Rethink Robotics. Sawyer robot. `https://www.rethinkrobotics.com/sawyer/`, 2018.

[51] Rethink Robotics. Baxter. `https://robots.ieee.org/robots/baxter/`, accessed Aug 2019.

[52] Universal Robots. Ur10 robot. `https://www.universal-robots.com/products/ur10-robot/`, Access Aug 2019.

[53] ROSIN. Universal robots ros-industrial driver. `http://rosin-project.eu/ftp/universal-robots-ros-industrial-driver`, Accessed Aug 2019.

[54] Manuel Ruder, Alexey Dosovitskiy, and Thomas Brox. Artistic style transfer for videos. *Pattern Recognition*, pages 26–36, 2016.

[55] Runway. Runway. https://learn.runwayml.com, Accessed Aug 2019.

[56] Paolo Salaris, Naoko Abe, and Jean-Paul Laumond. *A Worked-Out Experience in Programming Humanoid Robots via the Kinetography Laban*, pages 339–359. Springer International Publishing, Cham, 2016.

[57] Tomas Simon, Hanbyul Joo, Iain Matthews, and Yaser Sheikh. Hand keypoint detection in single images using multiview bootstrapping. In *CVPR*, 2017.

[58] Chris Smith. Ros nodejs package. http://wiki.ros.org/rosnodejs, Accessed Aug 2019.

[59] Sparkfun. Capacative touch breakout board. https://www.sparkfun.com/products/14520, Accessed Aug 2019.

[60] Paul Stoffregen. Teensy 3.2. https://www.pjrc.com/store/teensy32.html, Accessed Aug 2019.

[61] Simon Rasmussen Thomas Timm Andersen. ur_modern_driver. http://wiki.ros.org/ur_modern_driver, Accessed Aug 2019.

[62] Purdue University. Huang yi and kuka the robot to dance in poetic two-step. https://www.purdue.edu/convocations/event/huang-yi-kuka/, 2018.

[63] Victoria and Albert Museum. Slave/master. https://www.vam.ac.uk/event/zwEaoBwK/slave-master-ldf, Sept 2017.

[64] Shih-En Wei, Varun Ramakrishna, Takeo Kanade, and Yaser Sheikh. Convolutional pose machines. In *CVPR*, 2016.

[65] Nick Weldin. launchpad_sawyer.py. https://github.com/mdxdesigneng/launchpad/blob/movement_interface/scripts/launchpad_sawyer.py, Aug 2018.

[66] Nick Weldin. button.py. https://github.com/mdxdesigneng/launchpad/blob/ur/scripts/button.py, Aug 2019.

[67] Nick Weldin. midi_launchpad. https://github.com/mdxdesigneng/launchpad/blob/ur/scripts/midi_launchpad.py, Aug 2019.

[68] Nick Weldin. nano33iot code. https://github.com/mdxdesigneng/launchpad/tree/ur/handle/microcontroller/nano33iot, Aug 2019.

[69] Nick Weldin. ros_launchpad.py. https://github.com/mdxdesigneng/launchpad/blob/ur/scripts/ros_launchpad.py, Aug 2019.

[70] Nick Weldin. teensyur10.ino. https://github.com/mdxdesigneng/launchpad/blob/ur/handle/microcontroller/teensy/teensyUR10/teensyUR10.ino, Aug 2019.

[71] John Wilson and Lottie Thomas. Launchpad pro programmers reference guide. https://customer.novationmusic.com/sites/customer/files/novation/downloads/10598/launchpad-pro-programmers-reference-guide_0.pdf, Accessed Aug 2019.

[72] Yamaha. Tenori-on. https://www.yamaha.com/en/about/design/synapses/id_005/, Accessed Aug 2019.

[73] Huang Yi. Huang yi studio. http://huangyi.tw/huangyi_and_kuka/, 2018.

[74] Massimiliano Zanoni, Michele Buccoli, Augusto Sarti, Fabio Antonacci, Sarah Whatley, Rosemary Cisneros, and Pablo Palacio. Report on music-dance representation models, December 2017. This corresponds to Deliverable 3.3 (D3.3) of the WhoLoDancE project.

[75] Zenoot. Dances with robots: changing perceptions at the v and a. https://zenoot.com/dances-with-robots-changing-perceptions-at-the-va/, 2017.

# Appendix

This sheet of instructions was given to people who tested the system.

## Improvising with a robot

### Moving the Robot

The robot can be moved around by holding the handle on the end of the arm. It should detect when you hold it, the light turns green. While the light is green you can push the robot into a new position. When you let go it will stay in the new position you have moved it to. Once you have got it into a position that you want it to remember go to the Launchpad.

### Programming

The Launchpad controller enables you to store the position the robot is currently in. Each of the 64 square buttons can hold positions that the robot should move to.

To add a position to a square button hold down one of the circular buttons on the right side of the controller (the column of buttons will light up), and while holding that down press the button that you want to store the position in (like using the shift key on a computer keyboard).

Move the robot to a different positions and store them in different buttons. Play around and see what happens.

## Consider

What happens when you press a button that you have stored a position in ?

What happens if you now press the second button you stored a position in ?

What happens if you press the second button again ?

What if you try to add a position to a button that already has one ?

Do any other buttons effect what happens when you press the square buttons ?

What other things would you like to be able to do ?