Lively: Enabling Multimodal, Lifelike, and Extensible Real-time Robot Motion

Andrew Schoen
University of Wisconsin–Madison
Madison, Wisconsin, USA
schoen@cs.wisc.edu

Dakota Sullivan
University of Wisconsin–Madison
Madison, Wisconsin, USA
dsullivan8@wisc.edu

Ze Dong Zhang
University of Wisconsin–Madison
Madison, Wisconsin, USA
zzhang978@wisc.edu

Daniel Rakita
Yale University
New Haven, Connecticut, USA
daniel.rakita@yale.edu

Bilge Mutlu
University of Wisconsin–Madison
Madison, Wisconsin, USA
bilge@cs.wisc.edu

ABSTRACT

Robots designed to interact with people in collaborative or social scenarios must move in ways that are consistent with the robot's task and communication goals. However, combining these goals in a naïve manner can result in mutually exclusive solutions, or infeasible or problematic states and actions. In this paper, we present *Lively*, a framework which supports configurable, real-time, task-based and communicative or socially-expressive motion for collaborative and social robotics across multiple levels of programmatic accessibility. *Lively* supports a wide range of control methods (*i.e.,* position, orientation, and joint-space goals), and balances them with complex procedural behaviors for natural, lifelike motion that are effective in collaborative and social contexts. We discuss the design of three levels of programmatic accessibility of *Lively*, including a graphical user interface for visual design called *LivelyStudio*, the core library *Lively* for full access to its capabilities for developers, and an extensible architecture for greater customizability and capability.

CCS CONCEPTS

• **Human-centered computing** → **Open source software**.

KEYWORDS

Robot motion; robot control; Perlin noise; lifelikeness

ACM Reference Format:
Andrew Schoen, Dakota Sullivan, Ze Dong Zhang, Daniel Rakita, and Bilge Mutlu. 2023. Lively: Enabling Multimodal, Lifelike, and Extensible Real-time Robot Motion. In *Proceedings of the 2023 ACM/IEEE International Conference on Human-Robot Interaction (HRI '23), March 13–16, 2023, Stockholm, Sweden.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3568162.3576982

1 INTRODUCTION

As robots increasingly work in human environments, they will need to execute a wide range of highly configurable behaviors while communicating effectively with their users. A worker collaborating with a robotic arm may have preferences for how the robot positions itself when they are nearby [25]. A collaborative robot assisting a person unloading the dishwasher might use slight movements of its gripper to communicate that it is ready to pick up or receive

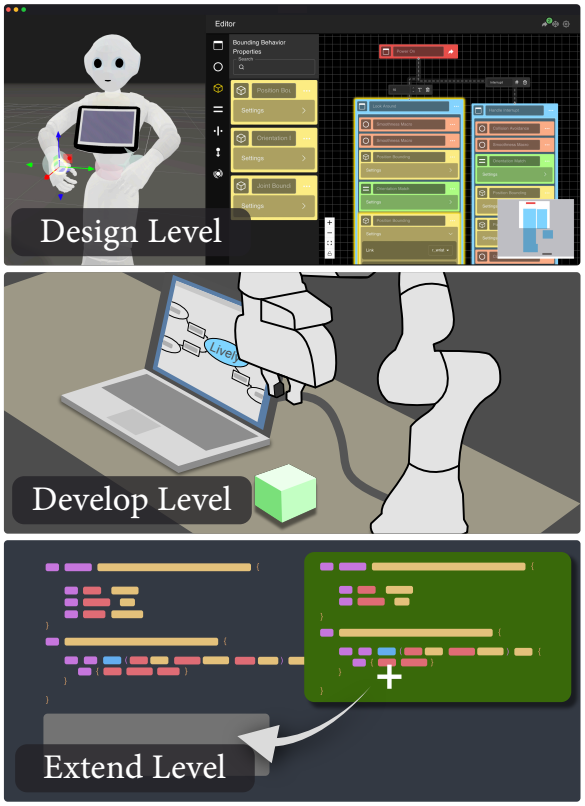

Figure 1: We present *Lively* for real-time motion generation that balances task and communicative goals while maintaining feasibility. We provide three levels of interfaces to address varying use cases. The *Design Level* enables programming robots using a state-based approach. The *Develop Level* is configurable and portable, usable in applications such as ROS-based control and web-based simulation. The *Extend Level* supports the addition of new characteristics and goal specifications for greater customizability and extendability.

HRI '23, March 13–16, 2023, Stockholm, Sweden.

ACM ISBN 978-1-4503-9964-7/23/03…$15.00
https://doi.org/10.1145/3568162.3576982

items [47]. A social robot may display idle motion with its body to indicate that it is active and lifelike [29]. When conversing, a robot may look away to signal that it is thinking [1]. Prior research in human-robot interaction has found such "lifelike" motions to improve perceptions of the robot [43, 48]. Thus, lifelike motion or configuration of a robot's links and joints are key design elements for robots utilized in human environments. Successful execution of combined tasks and social actions requires balancing these types of goals with practical concerns, such as avoiding collisions and maintaining smooth motion. In this paper, we explore how *Lively* can support the generation of lifelike but feasible task motions for collaborative and social robots.

Since physical task-based activities are frequently spatially rooted in the workspace, robot control requires converting these Cartesian goals into joint-space instructions. For example, the ability of a robot's arm to deliver an object to a collaborator depends on its ability to first reach the position of the object, and then travel to the person's outstretched hand. Similarly, a social robot may point in a certain direction using referential gestures by changing the position and orientation of its hand or gaze. This conversion is commonly achieved with an approach known as *Inverse Kinematics (IK)*. Conventional IK approaches structure this conversion as a search in joint-space constrained by the position and orientation of the robot's gripper. This approach encourages solutions exhibiting desired position and orientation goals on the gripper, but cannot guarantee finding a solution in all cases.

To communicate certain attitudes or states with physical motion, such as the human-robot interaction scenarios discussed above, the entire kinematic chain may be required, so simply considering the position and orientation of the gripper is insufficient.

Combining these social and task-based goals into functional robot motion requires not only knowledge of how motion is interpreted but also the technical ability to translate those qualities onto robot platforms. While robotics application developers may possess skills in both areas, domain experts may not have the same level of technical ability to bring their vision to fruition. An interface that is intuitive to both roboticists and other experts, such as animators, artists, or designers, can bridge this divide. Additionally, novel approaches to designing and implementing robot motion may be needed as robot capabilities evolve. Therefore, a design system with the flexibility to grow with these new approaches is required.

We present a new motion specification and generation framework, called *Lively*, that combines task-based and social goals while maintaining kinematic stability in real time (Figure 1). The framework leverages Perlin noise [30, 32] and integrates an existing per-instant pose optimization tool called RelaxedIK [37] to achieve both primary and secondary motion goals in real-time. To support robot-application designers and developers, we developed three levels that expose the capabilities of *Lively* to users with different needs and levels of expertise. At the first level, *LivelyStudio* provides nonexpert users with an accessible, interactive, visual interface to design primary and secondary motions and control modalities used with the robot. At the second level, we present a development- and execution-focused framework, and at the final level, we provide an architecture that supports extendability and customizability.

The contributions of our work are summarized as follows:

- A *visual interface* called *LivelyStudio* that allows designers to interactively construct state-based robot programs [1];
- An *open-source robot-agnostic library* that can be used by developers to specify real-time robot behavior that combines goal-oriented joint-space or Cartesian control with motion quality attributes in a feasible manner[2];
- A modular software architecture that supports straightforward augmentation and contribution for custom control.

In the remainder of the paper, we review previous approaches to this problem, contrasting them with *Lively*. We discuss the implementation of *Lively* and outline its cases for use along three different levels of programmatic accessibility, including the design of a tool called *LivelyStudio*, iteratively designed with a formative evaluation with roboticists and animators.

2 BACKGROUND

In this section, we review related work on expressive and functional motion including lifelike motion, inverse kinematics, and the operationalization of each.

2.1 Lifelike Motion

Whereas primary motion is an intentionally performed behavior, such as the process of handing a letter to a friend, standing in place, or looking to the right, secondary motion is defined as activity resulting from that primary motion [23]. Secondary motion covers a wide range, such as the rippling or creasing of one's shirt as the arm is outstretched, the idle shifting of posture while standing, or slight movements of the pupils.

Secondary motion is known to be highly important to how humans interpret animated or robotic characters. In their paper, Heider and Simmel animated a set of shapes to perform choreographed motions, such as following one another and moving into boxes while exhibiting additional subtle affine and rotational movements [20]. Most participants viewing the animation described the behavior of the simple shapes in human or anthropomorphic terms. Similarly, work with puppets has informed our understanding of how small motions and characteristics can influence viewers [12, 14]. The effectiveness of secondary motion motivated its inclusion in the principles of 3D animation by Lasseter [26].

Animation utilizes many principles for secondary motion and lifelike behavior, initially requiring hand-drawn or hand-animated specification of behaviors. However, a growing number of methods make this process less demanding. Witkin *et al.* proposed a method to warp a keyframe animation to match new spatio-temporal constraints by systematically mapping underlying motion curves [51]. This allows an animator to adjust a character's posture from happy to sad throughout an animation using only a sparse set of inputs instead of enumerating keyframes. Gleicher presented a method that maps motion from one articulated figure to another, even if they have vastly different scales or geometries [17]. The method uses non-linear constrained optimization to minimally displace an input motion (e.g., motion capture data) to match the specifications of the new articulated figure. Additionally, motion has been added to computer generated characters using Principal Components Analysis

[1] Code available at https://github.com/Wisc-HCI/LivelyStudio
[2] Code/Documentation available at https://github.com/Wisc-HCI/lively

[15], and traditionally animated characters have been augmented with secondary motion through a 3D intermediate process [22]. However, these solutions all represent post-hoc methods of adjusting input motions, and are allowed certain freedoms given their virtual, non-rigid context.

Considerable work has also focused on effective ways of augmenting agents and characters with secondary motion in a generative manner. The most common method to do this was created by Ken Perlin [30, 32]. Originally designed for texture generation, Perlin noise was quickly adopted for motion as a way of creating personality in animated characters [6, 31]. Perlin noise is particularly well-suited for this domain, being a non-repeating, but smoothly changing generative method. Furthermore, by modifying the speed at which the input value (usually a function of time) changes, animators can predictably control the characteristics of the noise function. By using smooth noise, such as Perlin noise, as a function of time, offsets from static or dynamic configurations (*i.e.,* character joints) can be calculated, thus augmenting these characters with subtle motion. This process was extended by *Improv*, which featured a method for incorporating smooth noise into animated characters' behaviors [33]. Studies in robotics have shown smooth noise to improve a variety of outcomes in robotics, including likeability and presence [8, 48]. Many commercially-available collaborative and social robots do not have fully articulated faces with which to communicate social-emotional states, so it is particularly important that there be alternative ways for modeling them.

Specific characteristics of motion, such as "jerkiness" or "velocity," have been outlined as important for the recognition of certain emotional states in humanoid robots [3, 4]. When viewed by individuals, faster speed in robots was interpreted as greater excitement or arousal [44]. Originating in dance theory, Laban Movement Analysis (LMA) [50] and the component of motion shape have since been validated as informative for affect detection in humans and used in animation [7, 10, 28, 49]. While not motivated by LMA, the directionality of a simple robot was shown to have a strong emotional impact [19], and has been used to generate profiles of expression movement in mobile robots [24].

Smooth, lifelike motion can also function as a signalling mechanism for system states [5, 21]. For example, if the robot is on but not moving, secondary motion may serve as an indicator to users that the robot is merely idle, while also preventing surprise when the robot moves. Idle behaviors have also been extracted from human ethnographic work [46], and have been shown to improve aspects of child-robot interactions [2].

While smooth joint noise can improve the liveliness of agents, it does not capture the full range of expressivity. According to LMA, many features of movement, such as shape directionality, are not relevant in the joint-space of the robot, but rather the *pose* (e.g., Cartesian space) of the robot, making applying these types of features difficult if operating in joint space. Other informative features, such as speed or jerkiness, may be obscured if joint-based noise causes varying speed or jerkiness in Cartesian space.

Similarly, the addition of smooth noise for secondary motion in joint-space can result in problematic configurations or collisions, even if joint limits are respected. For humanoid or bipedal robots, simply adding offsets to individual joints on the lower limbs quickly results in unstable posture, and even falls.

2.2 Solutions to Lifelike Motion in Robotics

One solution to these challenges is to simply pre-record or define keyframes for specific motions and interpolate between them as needed. This approach has been employed in prior research [48] and in proprietary software (e.g., Softbank Robotics' NaoQi Autonomous Life [41]). As an alternative to manually generating activities, *Geppetto* utilized a user interface to enumerate and visualize possibilities for expressive gestures with the goal of allowing more productive exploration of the potential behavior set [11]. For bipedal robots, motion on limbs presents an additional challenge due to instability caused by uncoordinated joint movements. As a result, motion is typically either disabled from the waist down, entirely pre-defined, or the issue is avoided by adopting a sitting position and focusing activity on the upper body [3, 4]. While sufficient for short interactions, pre-scripting these behaviors can have a number of issues. First, without enough keyframes, the behavior can quickly become repetitive, which breaks the illusion of autonomy [13]. Second, when combining activities, conflicts between joints and kinematics might arise. This makes interleaving existing motion with novel, real-time instructions difficult. For example, an early approach attempted to resolve these conflicts between activities and motions through a hierarchical model [45]. While effective at interleaving the behaviors with motion, the system was not fast enough to run in real-time. These cases illustrate the limitations of previous efforts to balance lifelike motion with task-goals.

2.3 Inverse Kinematics

In contrast to specifying the gripper pose indirectly through the setting of joint angles, Inverse Kinematics (IK) solvers attempt to directly specify the gripper pose, and solve for the joint configuration that satisfies that pose. IK solvers, while more easily interpretable in Cartesian space than joint-space methods, can encounter issues such as kinematic singularities. These joint-space issues occur when the robot loses the ability to instantaneously move its gripper in some translational or rotational dimension, because (1) not all poses in the robot's area can be reached through a combination of joint states, and (2) a movement in Cartesian space may not be possible as a smooth interpolation of joint-space values.

A method that utilizes an IK solver is ERIK, which uses a pass-based approach to integrate joint movements with end-effector goals [40]. *RelaxedIK* is another IK solver with a different approach. Using an optimization-based method, *RelaxedIK* places importance on both accuracy of the motion (*e.g.,* matching the pose of the gripper), as well as the feasibility of motion (*e.g.,* avoiding self-collisions or kinematic singularities) [37]. It is generalizable such that additional objectives can be added, *e.g.,* handling dual-robot systems where one arm controls a camera, optimizing the location and orientation of the camera such that a remote user has a clear view of the task being performed by the other robot arm [38].

3 IMPLEMENTATION

Lively inherits its philosophy from *RelaxedIK* [37] by framing the goal of the joint-space calculation as an objective, but generalizing its implementation across a greater set of objective types and attributes of the robot's state. Furthermore, while *RelaxedIK* assumed a position and rotation goal on the gripper of each robot arm, and a

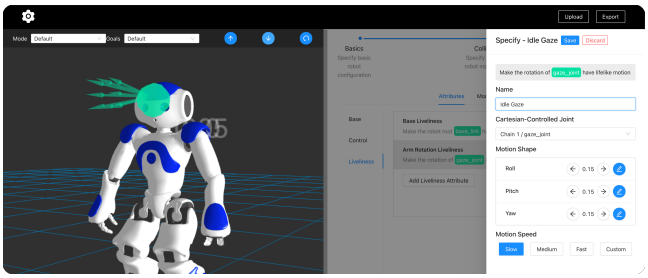

Figure 2: An early version of *LivelyStudio* that received feedback from animators and roboticists, which led to a redesigned 3D environment, more explicit state-based design process (states as graph nodes), and bundling of behavior attributes with specific goals and weights.

set of joint smoothness objectives, *Lively* makes fewer assumptions with its *à la carte* approach, giving the programmer greater ability to compose these goals in creative ways for behavior generation.

To explore the capabilities of the system, we will consider three main levels of possible interaction with the system: the *Design Level*, the *Development Level*, and the *Extension Level*.

3.1 Design Level

The outermost interaction level is the *Designer Level*, and is the least technical way to explore and utilize the system. We designed *LivelyStudio* as a method inspired by conversations with a set of experts across the fields of animation and robotics. It is meant to support and illustrate many of the capabilities of the *Lively* framework, while maintaining its accessibility. This is done by using a state-based approach, wherein users can compose combinations of social, task-based, or functional behaviors, called *Behavior Properties*, and specify how transitions may occur between these states.

3.1.1 Design Iteration. Our current version of *LivelyStudio* builds upon previous iterations through a small formative evaluation with four professional roboticists and animators involving a mixture of system overview, think aloud, and semi-structured interview, lasting 60 to 90 minutes. The initial design, shown in Figure 2, featured a simulator and configuration window, where users could independently curate a set of *Behavior Properties*, a set of states (called modes), and goals (task-based instructions). While states were supported through modes, there was no clear relationship between them, and animators in particular had difficulty translating their keyframe-focused experience to this design: "It's hard to see how poses would be created so separate from the animation (P3)." More generally, how specific goals could be combined with the *Behavior Properties* was unclear. Additionally, certain interface elements, such as the standard 3D viewer did not have the affordances desired by animators, or had minor usability issues. This feedback was used to create a more effective and intuitive version of *LivelyStudio* for users of varied backgrounds and levels of experience through a more explicitly state-based configuration process, and use of a new custom 3D viewer and updated components.

3.1.2 LivelyStudio Interface. The results of our formative evaluation suggested that a state-based visual programming environment that allows users to develop series of states similar to keyframing would be the most intuitive approach to the design. The state-based approach shares similarities with many other programming environments [9, 16, 34, 35], which may be familiar to roboticists, but also enables an intuitive design approach for users who are less familiar with typical programming environments like animators, digital artists, or other types of designers. *LivelyStudio*'s programming environment contains three primary parts: (1) a selection of state and behavior property nodes, (2) a state-based programming window, and (3) a robot scene. By defining states, and adding *Behavior Properties*, designers can define how a robot will move, or the position it should take in each (Figure 3). Improving on the early version of *LivelyStudio*, specific goals and *Behavior Properties* are merged for clarity, and weights are inferred from their relative ordering within states and through usage of priority groups. Designers can specify arbitrary Universal Robot Description Files (URDFs), but visualization of meshes is limited to a discrete set that could be expanded in the future.

3.1.3 Behavior Properties. LivelyStudio allows for a wide range of robot *Behavior Properties* with which users program robot motion. These 24 properties, which serve as building blocks for defining the behavior and motion of the robot, fit into six categories:

- *Basic behavior properties* revolve around the fluidity of robot motion by limiting rapid changes and considering possible collisions between the links of the robot.
- *Bounding behavior properties* limit the space within which joints can assume angles and links can move or be oriented.
- *Matching behavior properties* specify exact positions and orientations of links or angles of joints.
- *Mirroring behavior properties* allow users to mirror the current state of a link's position or orientation in a different link, or the current angle of one joint in another.
- *Liveliness behavior properties* allow the addition of smooth, coordinated motion to joint angles or link poses.
- *Force behavior properties* simulate the effects of physical forces acting upon the robot.

The function of each *Behavior Property* is visualized in Figure 4.

3.1.4 States and Transitions. The state-based programming window starts with a power-on (*i.e.,* initial) state, and a power-off (*i.e.,* final) state. Users can add additional state nodes to their program and populate them with *Behavior Properties*. For example, one state may contain a property that sets the gripper of a robot arm in a pick-up area, while another state sets the gripper position to be near a drop-off area. Once a series of states is created, the user can define how the power-on, power-off, and custom states are connected by dragging transitions from one state to another. These connections can also be given timers, which act as triggers to automatically begin a transition from one node to the next. In this way, a state can function both conventionally, defining a set of characteristics the robot will exhibit for an unspecified amount of time, but also as a single keyframe in a timed series. States can have any number of both timed and nominal transitions (simulating event triggers, *e.g.,* a person approaches), and the program will transition states given the first simulated event triggered or timer that expires, whichever occurs first. Of note, while this does simulate how the robot could

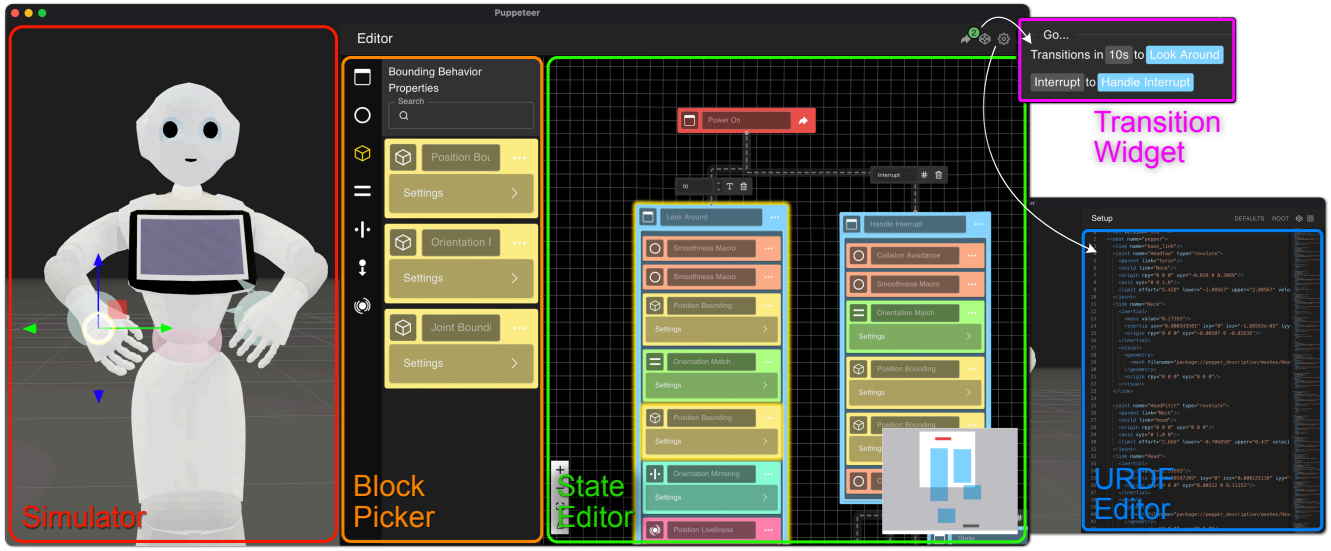

Figure 3: The layout of the *LivelyStudio* interface. From left to right, a *Simulator* window shows the robot in the currently selected state; the *Block Picker* allows dragging structural blocks like *States* or *Behavior Properties* like *Position Bounding*; the *State Editor* canvas that allows for states to be dragged around and modified. At the top-right, a menu that reveals a *Transition Widget*, which lists transitions from the current state, and a settings button that reveals a full URDF editor.

respond to events, *LivelyStudio* does not currently interface with physical robots, or listen to external events.

3.2 Develop Level

For robot programmers desiring greater control over the robot than that afforded by the previously described *LivelyStudio* interface, or looking to control a robot in a more conventional ROS-based approach by creating a control node that publishes joint values, the *Development Level* allows for direct control using *Lively*.

3.2.1 Design & Usage. *Lively* is written in Rust [27], and accessible as a crate, with bindings in both JavaScript through WebAssembly [18] and Python [42]. To use *Lively*, a *Solver* is imported and constructed with any valid URDF, persistent scene objects, objectives, and other solver settings. Execution of the *solve* method, which accepts the current goals, weights, time, and real-time collision data, returns a robot state that best satisfies those goals given the current weightings and previous robot state. This approach allows for *Lively* to be used in a variety of contexts, including ROS [36], web or simulation, and directly on hardware. Solve times with randomly arranged colliders are shown in Figure 5.

3.2.2 Objectives, Goals, & Weights. To achieve a high degree of customization and dynamic control, we introduce the concepts of objectives, goals, and weights. Whereas *LivelyStudio* abstracted away these features as *Behavior Properties* for the purpose of accessibility, the core framework allows for more direct control. The identities of the individual objectives match with the set of *Behavior Properties* enumerated in Figure 4, and goals are summarized in 1. Importantly, while *Behavior Properties* encoded the discrete goals (e.g., the position for *Position Match*, or the scalar for *Joint Match*) associated with each *Behavior Property*, and the weights are inferred by the ordered ranking within states, these are separated at the framework level. Thus, the previously mentioned position goal can be determined in real time through external means, such as sensing, and passed as an update within each iteration of the *solver*. Similarly, the developer in real time can adjust other goals, such as a position bounding ellipsoid (*Position Bounding*), joint values (*Joint Match*), or size (for *Position Liveliness*), and weights, allowing for prioritization of certain goals or the deactivation of others, based on the current development needs. Because objectives are organized by key, and atomic updates are possible for goals and weights, only the needed changes must to be included each round.

3.2.3 Objective Configuration. The complete set of objectives feature a wide range of configurable attributes, beyond simply their goals and weights. The simplest objectives focus on safe and smooth motion, corresponding to the set of *Basic Behavior Properties*, and do not accept additional parameters. Those corresponding to *Matching*, *Bounding*, and *Gravity Behavior Properties* are configured with the joint or link with which they are paired. *Mirroring Behavior Properties*, defining relationships between pairs of links and joints, accept a pair of each. Finally, *Liveliness Behavior Properties* feature an additional field, frequency. This value functions as a temporal scaling value that increases or decreases the rate of change in the Perlin noise generator functions for that objective. Combined with the goal values passed into liveliness objectives, developers can access a wide range of motion profiles. Importantly, because the formulation of the liveliness objectives is not dependent on having a concrete goal attached to the same link or joint, it is possible to add movement to otherwise uncontrolled parts of the robot.

3.2.4 Collision Avoidance. *Lively* implements the *PROXIMA* collision detection algorithm, which allows for time-efficient collision and proximity detection for robots [39]. The *Collision Avoidance* objective serves to utilize the data generated from this collision

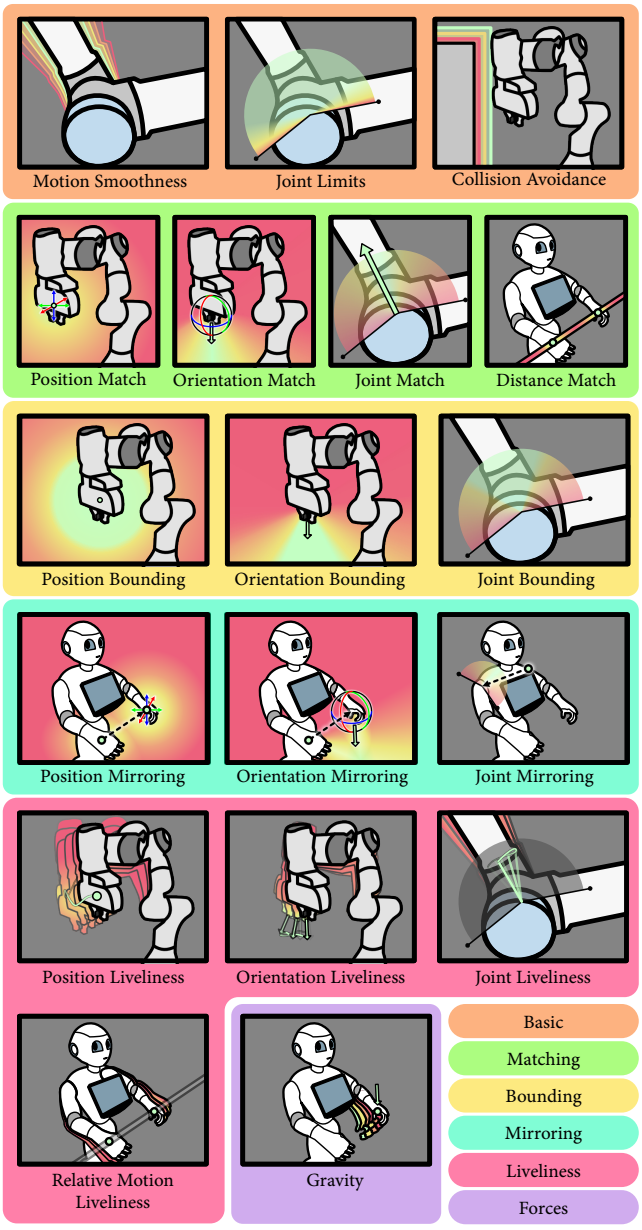

Figure 4: *LivelyStudio*'s set of *Behavior Properties* that match *Objective Functions* within *Lively*. Note, Velocity Minimization, Acceleration, and Jerk Minimization come in both joint-based and robot root variants, and while usable separately, are included within the *Smoothness* macro property.

detection algorithm to prevent collisions. *Lively* employs a three-fold approach to handling modeling collision objects. The first is input from the URDF during the initialization of the solver, which supports default shapes like boxes and cylinders as parts of the collision model when parsed. For cross-platform and web-based reasons, mesh-based colliders are ignored during URDF import. Additional colliders can be specified during *solver* initialization, including basic shapes and convex hulls, and can be attached to the world or any link in the robot. Finally, as an optional input to the *solve* method, developers can provide real-time updates to the collision model, adding, deleting, and moving colliders.

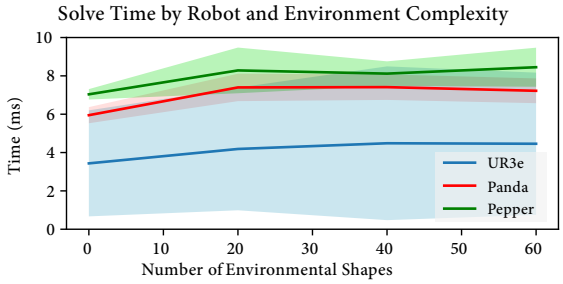

Figure 5: Solve times for the UR3e, Panda, and Pepper robots, with randomized locations of environmental colliders. Of note, speed is largely unaffected by shape count.

3.3 Extend Level

For robotics developers seeking to modify the behavior of the existing *Lively* objectives, or wanting to increase functionality by creating completely new objectives, *Lively* has a modular and configurable approach to supporting the *Extension Level*.

3.3.1 State Model. As discussed, *RelaxedIK* utilizes an optimization approach, with the robot state S_R being represented as a vector in the joint space S_J of the robot internally. *Lively* takes a similar approach, but an additional six dimensions representing the transform of the root link are added to create the optimized vector x. However, this vector representation is not always the most natural way to evaluate the state, and to ease the computation each objective performs, this vector is converted into a more comprehensive state representation containing joint states, link transforms, and proximity information, described in Table 2. This state, as well as previous states, are provided in each call to objectives.

This formulation of the state allows for straightforward creation of additional objectives. It is also possible that additional features of state may be needed for the creation of certain new objectives. The *Robot Model* handles the generation of new robot states from the vector x. For example, if a force-based objective was desired, the robot model would have to be extended to output a state that provides the data the objective would have to operate on.

Table 1: Goal Types

Entry	Description
Translation	A 3-vector representing coordinates
Rotation	A Quaternion representing rotation
Scalar	A float value
Size	A 3-vector representing scale of a 3D shape
Ellipse	A structure designating a rotated ellipsoid, with *Translation*, *Rotation*, and *Size* components
RotationRange	A structure including a center *Rotation*, as well as a float value indicating allowed delta in radians from that rotation.
ScalarRange	A structure including a center float value, and float value representing allowed delta from that value.

Table 2: State Properties

Entry	Description
Frames	A lookup table of each link's position in both world and local coordinates
Joints	A lookup table of each joint's value
Origin	The transform of the root link. This data is also included in frames
Proximity	A vector of data representing pairwise proximity between the robot's links and other robot parts and the environment. Each entry contains distance, as well as the closest points between the pair of colliders
Center of Mass	A 3-vector representing the center of mass of the robot in the world frame

Table 3: Objective Description

Entry	Description
update	Function, accepts the current timestep and performs any updates to its internals that are necessary, as in the case of Perlin noise-based objectives
set goal	Function, accepts the goal value supplied by the user. Each objective accepts a specific goal type
set weight	Function, accepts a new weight value, if updated by the user
weight	Float, indicates the scaling value for the objective cost value
call	Function, accepts a $State$ and $Variable$ data object, returning a numerical cost value. The $Variable$ object contains a record of previous states and information about the robot

3.3.2 Objective Formulation. Similar to *robot state*, each objective adheres to a well-defined convention that can be used to extend the capabilities of *Lively*, as shown in Table 3. As previously discussed, each objective is paired with a specific goal type (e.g., *Position Bounding* with *Ellipse*, and *Position Liveliness* with *Size*), and the goals are enumerated in Table 1. Additional goal types can also be added to support new objectives and functionality, as long as they have a predictable structure (*e.g.,* a *pointcloud* goal could be an array of any length with structure $[\{x : f64, y : f64, z : f64\}, ...]$).

4 CASE STUDIES

4.1 Design Level

Users of a wide range of experience levels can engage with our system using *LivelyStudio*. Artists, character designers, and animators, who may not be familiar with traditional programming tools, may particularly benefit from *LivelyStudio*'s accessible user interface.

4.1.1 Kiosk Robot. Suppose a user is creating a program for a social robot providing general assistance in a public area. Here, the robot may have states such as idle, greeting, or thinking. The user can begin by creating state nodes within the state editor. One such state could be labeled "Idle" to represent the idle status of the robot within the overall program. From here the user can begin adding *Behavior Properties* to the state. First, the user may apply the *Position Liveliness* property to the torso of Pepper as a visual indication that it is powered-on and functioning. Next, the user may add the *Joint Liveliness* property, and configure it to "Head Yaw" to make the robot's head sway from left to right and signify that it is looking around for people to assist. Finally, the user can select the *Smoothness Macro* property to ensure that the robot's motion remains smooth and natural, and the *Collision Avoidance* property to prevent collisions. The user may also create a "Greeting" state, which directs the robot's gaze toward a nearby person. Once these states are generated, the user can create a connection between them and add a label to identify a triggering condition. The user may want Pepper to transition from the "Idle" state to a "Greeting" state when a person approaches. During this transition, Pepper can reduce head sway from the "Idle" state, direct gaze toward the user in the "Greeting" state, and maintain the liveliness motion included in both states. This process can be repeated with any number of states and complex transition patterns.

4.1.2 Cobot Keyframing. In another example, a user may want to create a program for a robotic arm such as the Panda robot that functions as a series of states, similar to keyframing. The user can create an initial state, add the *Position Match* property, and configure a specific position for the gripper. The user can complete this process to define all waypoints for the gripper of the Panda robot as separate states. Given space constraints in the deployment environment, the user may also want to design their program to limit the space in which certain links will move. Thus, the user may apply the *Position Bounding* property to specific links so that the robot limits its spatial footprint while moving. Finally, the user may need the robot to interact with an object from a specific grasp point. Therefore, adding the *Orientation Match* property to the gripper enables it to manipulate an object from a reasonable angle. Once all the states are created, the user can create timed connections between states, such that transitions will occur automatically.

4.2 Develop Level

While all users may find use in *LivelyStudio*, those with substantive experience programming and planning robot motion will be able to leverage the capabilities of *Lively* directly. We consider two example use cases to explore how *Lively* may be used.

4.2.1 Real-Time Robot Control. Using a UR3-e series robotic arm, a developer seeks to devise a system that, on button-press, scans the area using a camera attached to the last robot link, and finds any of a set of items. Any item it finds is picked up and placed in a nearby box. The developer creates a ROS-based setup with two nodes. One node receives a camera feed and transform data from the robot, while publishing all valid items and their transforms that it detects. A second, *Lively*-focused control node listens to this set of items, and publishes transforms of the robot to be consumed by the first node. The control node defines a *Lively solver*, configured with the robot description, and an additional camera collider that is attached to the last link. The solver is configured with *Position Match* and *Orientation Match* objectives on the final link, and a *Position Liveliness* objective on the forearm link. Finally, the set of objectives is completed with *Smoothness*, *Joint Limits*, and *Collision Avoidance* objectives. On button press, a preset collection of positions and orientations are sequentially passed to the corresponding objectives in the *solve* method, along with instructions to turn the liveliness

weight to zero. The resulting state is parsed and converted into TF messages, which are passed via a topic to the data parsing node. Upon calculation and communication of scene items to the control node, the node selects the first item to move, passing the position and orientation to the solver, followed by the goal position of the items, then repeating until no items remain. Once complete, the position and orientation goals are moved to a neutral pose, and the weights relaxed, while the liveliness objective weight is increased.

4.2.2 Browser-Based WOZ. A developer wants to create a ROS-based wizard-of-oz GUI interface that allows actions to be selected and executed on a robot in real time, but also want the robot to respond to potential collision objects in the environment and exhibit certain lifelike motions. The robot, Pepper, has two arms, wheels, and a head, and the developer already has an existing library of joint-based trajectories. However, they want to include additional liveliness in orientation space around the head and position liveliness (a swaying motion) on the torso. Objectives for each controlled joint are created, as well as some basic objectives. The developer's GUI initializes a web-based version of the solver. A web-based ROS connection is formed to the robot, starting a subscription to sensor data, and a publisher that sends real-time joint instructions to the robot. Selecting an action updates the goals for each joint, and the set of all potential colliders that the robot gets from the sensors are updated each invocation of the *solve* method. Joint instructions from the result are passed to the robot after each solution is found. To accommodate all goals simultaneously, the system will attempt to reach the specified joint values, while adding in liveliness and avoiding collisions.

4.3 Extend Level

The current functionality of *Lively* and *LivelyStudio* address most user needs when programming robot motion. However, if additional functionality is desired, a developer could easily extend our system's capabilities by defining new objectives and goals. We outline two examples of extensions that would be feasible within *Lively*.

4.3.1 Center of Mass Objective. *Lively* can be greatly extended through the development of additional objectives. Because the robot state already includes a vector representing the center-of-mass of the robot, it is straightforward to create a new objective, implementing the methods defined in Table 3, that operates on it, which could be useful in cases where the robot's balance must be maintained, or as a way to center the robot near its base. The specified objective would accept a *Translation* goal, and use the default implementation of *update*. The *call* method would be implemented by calculating the distance between the goal value and the center-of-mass vector in the robot state, returning a cost that grows with distance. Finally, the objective is added to the set of Objectives. The resulting objective would attempt to produce poses that are centered as much as possible on the goal vector provided.

4.3.2 Perspective Noise. While the *Position Match* and *Orientation Match* objectives together are capable of creating a lifelike appearance, a developer may desire to create a lifelike behavior that exhibits positional and rotational motion around an offset focal point, as if inspecting the properties of an object located there. Doing so requires the addition of an new goal type, which would encode the focal length to maintain the position of the focus, and the amount of rotational/translational movement allowed. The objective's *call* method would use these goals and a Perlin noise generator function to project the needed position and orientation in space to achieve the specified rotation around the focus at a given time and compute the radial and translational distance from those values, returning a cost value. The resulting objective would attempt to produce poses that adhered to this dynamic pattern as a function of time.

5 DISCUSSION

Simultaneous coordination of functional and expressive robot motions is necessary but challenging. While a naïve approach may combine these types of motion, it may produce incompatible or undesirable results. In this paper, we presented a system that generates real-time, lifelike motion for collaborative and social robots, addressing key limitations of prior approaches at three levels of programmatic accessibility. At the most accessible level is *LivelyStudio*, a state-based visual programming and configuration environment that allows for exploration and design. Developers can utilize *Lively* directly in multiple programming and execution environments, in applications ranging from traditional keyframing-based to real-time control. Finally, we present an architecture for *Lively* that supports customizability and extendability.

5.1 Limitations & Future Work

The limitations of the work presented point toward future research and implementation opportunities. First, *Lively* currently takes a purely kinematic approach, but future extensions of *Lively* might apply dynamics to accommodate external or inertial effects on the robot's motion as well as higher-level behavioral objectives for avoiding non-stationary objects and obstacles with base movement. Additionally, while supporting a high degree of configurability for liveliness-focused *Behavior Properties*, *LivelyStudio* could be made more effective through the use of programming by demonstration, in which designers could demonstrate an example of liveliness that they would like to see articulated with a physical robot connected to the system, which could be converted into corresponding *Behavior Properties* automatically. The growing field of soft robotics presents interesting and relevant challenges for effective control that should be explored with respect to *Lively*. Finally, we plan to holistically evaluate *LivelyStudio* and *Lively* with (1) a usability evaluation of *LivelyStudio* that focuses on usability, learnability, and effectiveness of the system in supporting motion design, and (2) a long-term community-based, qualitative evaluation of the develop and extend levels to better understand their usage in a rigorous but also ecologically-valid manner, considering community engagement and usage on public-facing hosting locations.

Taken together, *Lively* and *LivelyStudio* aim to assist and motivate future work in systems exhibiting both social and task-based motion as a platform for design, development, and extension.

6 ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation (NSF) award IIS-1925043. We would like to thank Curt Henrichs and Yeping Wang for feedback and assistance, and the volunteer roboticists and animators for their feedback during development.

REFERENCES

[1] Sean Andrist, Xiang Zhi Tan, Michael Gleicher, and Bilge Mutlu. 2014. Conversational gaze aversion for humanlike robots. In *2014 9th ACM/IEEE International Conf. on Human-Robot Interaction (HRI)*. IEEE, 25–32.

[2] T. Asselborn, W. Johal, and P. Dillenbourg. 2017. Keep on moving! Exploring anthropomorphic effects of motion during idle moments. In *2017 26th IEEE International Symp. on Robot and Human Interactive Communication (RO-MAN)*. 897–902.

[3] Aryel Beck, Lola Cañamero, Antoine Hiolle, Luisa Damiano, Piero Cosi, Fabio Tesser, and Giacomo Sommavilla. 2013. Interpretation of emotional body language displayed by a humanoid robot: A case study with children. *International Journal of Social Robotics* 5, 3 (2013), 325–334.

[4] Aryel Beck, Antoine Hiolle, and Lola Cañamero. 2013. Using Perlin Noise to Generate Emotional Expressions in a Robot. *CogSci* (2013).

[5] Tony Belpaeme, Paul E Baxter, Robin Read, Rachel Wood, Heriberto Cuayáhuitl, Bernd Kiefer, Stefania Racioppa, Ivana Kruijff-Korbayová, Georgios Athanasopoulos, Valentin Enescu, Rosemarijn Looije, Mark Neerincx, Yiannis Demiris, Raquel Ros-Espinoza, Aryel Beck, Lola Cañamero, Antione Hiolle, Matthew Lewis, Ilaria Baroni, Marco Nalin, Piero Cosi, Giulio Paci, Fabio Tesser, Giacomo Sommavilla, and Remi Humbert. 2013. Multimodal Child-Robot Interaction: Building Social Bonds. *Journal of Human-Robot Interaction* 1, 2 (Jan. 2013), 1–21.

[6] Bobby Bodenheimer, Anna V Shleyfman, and Jessica K Hodgins. 1999. The effects of noise on the perception of animated human running. In *Computer Animation and Simulation'99*. Springer, 53–63.

[7] Diane Chi, Monica Costa, Liwei Zhao, and Norman Badler. 2000. The EMOTE model for effort and shape. In *Proceedings of the 27th annual Conf. on Computer graphics and interactive techniques*. 173–182.

[8] Raymond H. Cuijpers and Marco A. M. H. Knops. 2015. Motions of Robots Matter! The Social Effects of Idle and Meaningful Motions. In *Social Robotics*, Adriana Tapus, Elisabeth André, Jean-Claude Martin, François Ferland, and Mehdi Ammi (Eds.). Springer International Publishing, Cham, 174–183.

[9] Chandan Datta, Chandimal Jayawardena, I Han Kuo, and Bruce A MacDonald. 2012. RoboStudio: A visual programming environment for rapid authoring and customization of complex services on a personal service robot. In *2012 IEEE/RSJ International Conf. on Intelligent Robots and Systems*. IEEE, 2352–2357.

[10] Marco De Meijer. 1989. The contribution of general features of body movement to the attribution of emotions. *Journal of Nonverbal behavior* 13, 4 (1989), 247–268.

[11] Ruta Desai, Fraser Anderson, Justin Matejka, Stelian Coros, James McCann, George Fitzmaurice, and Tovi Grossman. 2019. Geppetto: Enabling Semantic Design of Expressive Robot Behaviors. In *Proceedings of the 2019 CHI Conf. on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3290605.3300599

[12] Brian R Duffy. 2003. Anthropomorphism and the social robot. *Robotics and Autonomous Systems* 42, 3-4 (March 2003), 177–190.

[13] Brian R Duffy. 2008. Fundamental Issues in Affective Intelligent Social Machiness. *The Open Artificial Intelligence Journal* 2, 1 (2008).

[14] Brian R Duffy and Karolina Zawieska. 2012. Suspension of disbelief in social robotics. In *2012 RO-MAN: The 21st IEEE International Symp. on Robot and Human Interactive Communication*. IEEE, 484–489.

[15] A. Egges, T. Molet, and N. Magnenat-Thalmann. 2004. Personalised real-time idle motion synthesis. In *12th Pacific Conf. on Computer Graphics and Applications, 2004. PG 2004. Proceedings*. 121–130.

[16] Dylan F Glas, Takayuki Kanda, and Hiroshi Ishiguro. 2016. Human-robot interaction design using interaction composer eight years of lessons learned. In *2016 11th ACM/IEEE International Conf. on Human-Robot Interaction (HRI)*. IEEE, 303–310.

[17] Michael Gleicher. 1998. Retargetting motion to new characters. In *Proceedings of the 25th Annual Conf. on Computer Graphics and Interactive Techniques*. 33–42.

[18] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation*. 185–200.

[19] J. Harris and E. Sharlin. 2011. Exploring the affect of abstract motion in social human-robot interaction. In *2011 RO-MAN*. 441–448.

[20] Fritz Heider and Marianne Simmel. 1944. An experimental study of apparent behavior. *The American journal of psychology* 57, 2 (1944), 243–259.

[21] Hiroshi Ishiguro and T Minato. 2005. Development of androids for studying on human-robot interaction. In *International Symp. on Robotics*, Vol. 36. 5.

[22] Eakta Jain, Yaser Sheikh, Moshe Mahler, and Jessica Hodgins. 2010. Augmenting Hand Animation with Three-Dimensional Secondary Motion. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symp. on Computer Animation* (Madrid, Spain) *(SCA '10)*. Eurographics Association, Goslar, DEU, 93–102.

[23] Ollie Johnston and Frank Thomas. 1981. *The illusion of life: Disney animation*. Disney Editions New York.

[24] Heather Knight and Reid Simmons. 2014. Expressive motion with x, y and theta: Laban effort features for mobile robots. In *The 23rd IEEE Iinternational Symp. on Robot and Human Interactive Communication*. IEEE, 267–273.

[25] Przemyslaw A Lasota and Julie A Shah. 2015. Analyzing the effects of human-aware motion planning on close-proximity human–robot collaboration. *Human factors* 57, 1 (2015), 21–33.

[26] John Lasseter. 1987. Principles of traditional animation applied to 3D computer animation. In *ACM Siggraph Computer Graphics*, Vol. 21. ACM, 35–44.

[27] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.

[28] Ayelet Melzer, Tal Shafir, and Rachelle Palnick Tsachor. 2019. How Do We Recognize Emotion From Movement? Specific Motor Components Contribute to the Recognition of Each Emotion. *Frontiers in Psychology* 10 (2019), 1389. https://doi.org/10.3389/fpsyg.2019.01389

[29] Marek P Michalowski, Selma Sabanovic, and Reid Simmons. 2006. A spatial model of engagement for a social robot. In *9th IEEE International Workshop on Advanced Motion Control, 2006.* IEEE, 762–767.

[30] Ken Perlin. 1985. *An image synthesizer*. Vol. 19. ACM.

[31] Ken Perlin. 1995. Real time responsive animation with personality. *IEEE transactions on visualization and Computer Graphics* 1, 1 (1995), 5–15.

[32] Ken Perlin. 2002. Improving noise. In *ACM Transactions on Graphics (TOG)*, Vol. 21. ACM, 681–682.

[33] Ken Perlin and Athomas Goldberg. 1996. Improv: A system for scripting interactive actors in virtual worlds. In *Proceedings of the 23rd annual Conf. on Computer graphics and interactive techniques*. 205–216.

[34] David Porfirio, Allison Sauppé, Aws Albarghouthi, and Bilge Mutlu. 2018. Authoring and verifying human-robot interactions. In *Proceedings of the 31st Annual ACM Symp. on User Interface Software and Technology*. 75–86.

[35] Emmanuel Pot, Jérôme Monceaux, Rodolphe Gelin, and Bruno Maisonnier. 2009. Choregraphe: a graphical tool for humanoid robot programming. In *RO-MAN 2009-The 18th IEEE International Symp. on Robot and Human Interactive Communication*. IEEE, 46–51.

[36] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[37] Daniel Rakita, Bilge Mutlu, and Michael Gleicher. 2017. A motion retargeting method for effective mimicry-based teleoperation of robot arms. In *Proceedings of the 2017 ACM/IEEE International Conf. on Human-Robot Interaction*. ACM, 361–370.

[38] Daniel Rakita, Bilge Mutlu, and Michael Gleicher. 2018. An Autonomous Dynamic Camera Method for Effective Remote Teleoperation. In *Proceedings of the 2018 ACM/IEEE International Conf. on Human-Robot Interaction*. ACM, 325–333.

[39] Daniel Rakita, Bilge Mutlu, and Michael Gleicher. 2022. PROXIMA: An Approach for Time or Accuracy Budgeted Collision Proximity Queries. In *Proceedings of Robotics: Science and Systems (RSS)*.

[40] Tiago Ribeiro and Ana Paiva. 2017. Animating the Adelino Robot with ERIK: The Expressive Robotics Inverse Kinematics. In *Proceedings of the 19th ACM International Conf. on Multimodal Interaction* (Glasgow, UK) *(ICMI '17)*. Association for Computing Machinery, New York, NY, USA, 388–396. https://doi.org/10.1145/3136755.3136791

[41] Softbank Robotics. 2022. Autonomous Life. http://doc.aldebaran.com/2-8/family/nao_user_guide/nao_life.html.

[42] Michel F Sanner et al. 1999. Python: a programming language for software integration and development. *J Mol Graph Model* 17, 1 (1999), 57–61.

[43] Vanessa Sauer, Axel Sauer, and Alexander Mertens. 2021. Zoomorphic gestures for communicating cobot states. *IEEE Robotics and Automation Letters* 6, 2 (2021), 2179–2185.

[44] Sara Baber Sial, Muhammad Baber Sial, Yasar Ayaz, Syed Irtiza Ali Shah, and Aleksandar Zivanovic. 2016. Interaction of robot with humans by communicating simulated emotional states through expressive movements. *Intelligent Service Robotics* 9, 3 (2016), 231–255.

[45] S Snibbe, M Scheeff, and K Rahardja. 1999. A layered architecture for lifelike robotic motion. *Proceedings of the International Conf. on Advanced Robotics* (1999).

[46] H. Song, M. J. Kim, S. Jeong, H. Suk, and D. Kwon. 2009. Design of idle motions for service robot via video ethnography. In *RO-MAN 2009 - The 18th IEEE International Symp. on Robot and Human Interactive Communication*. 195–199.

[47] Kyle Strabala, Min Kyung Lee, Anca Dragan, Jodi Forlizzi, Siddhartha S Srinivasa, Maya Cakmak, and Vincenzo Micelli. 2013. Toward seamless human-robot handovers. *Journal of Human-Robot Interaction* 2, 1 (2013), 112–132.

[48] Yunus Terzioğlu, Bilge Mutlu, and Erol Şahin. 2020. Designing Social Cues for Collaborative Robots: The Role of Gaze and Breathing in Human-Robot Collaboration. In *Proceedings of the 2020 ACM/IEEE International Conf. on Human-Robot Interaction*. 343–357.

[49] Arthur Truong, Hugo Boujut, and Titus Zaharia. 2016. Laban descriptors for gesture recognition and emotional analysis. *The visual computer* 32, 1 (2016), 83–98.

[50] Rudolf Von Laban and Roderyk Lange. 1975. *Laban's principles of dance and movement notation*. Princeton Book Co Pub.

[51] Andrew Witkin and Zoran Popovic. 1995. Motion warping. In *Proceedings of the 22nd annual Conf. on Computer graphics and interactive techniques*. 105–108.