Systems For Synthesizing and Studying Robot Motion

Christopher Bodden,* Bilge Mutlu,† and Michael Gleicher‡
Department of Computer Sciences, University of Wisconsin–Madison
1210 West Dayton Street, Madison, Wisconsin 53706 USA
{*cbodden, †bilge, ‡gleicher}@cs.wisc.edu

***Abstract*—We describe and detail two systems developed by the University of Wisconsin–Madison Graphics Group and Human-Computer Interaction lab designed to synthesize robot arm motion and evaluate human-robot collaborative performance. STrOBE is a general purpose robot trajectory optimizer. STrOBE optimizes a motion objective subject to a set of robot or environmental constraints using a nonlinear solver. RobotSim is an experimentation system implemented as a game. RobotSim measures the human-robot collaborative performance of robot arm motions. These systems have been developed with the intent of being released as open source software. We hope that releasing this software to the scientific community encourages further exploration of human-robot collaboration. This report serves as a resource for anyone looking to get started using these systems or looking for ideas for their own software.**

I. Introduction

Motion planning is an important step for performing any type of robot task. The ability to produce collision-free and minimal energy trajectories has been widely studied and achieved by state-of-the-art motion planners [1], [2]. Minimal energy trajectories are functionally important in many environments, such as manufacturing, because these trajectories reduce wear on the robot's actuators. However, in many of these environments robots interact and in some cases collaborate with humans. In collaborative scenarios, minimal energy may not always be the most desirable quality of a motion.

Communicating intent is an important component of motion in collaborative scenarios because it affects the clarity of the desired outcome [3], [4]. In this report we refer to the clarity of the motion's outcome as *intent-expressiveness*. The intent-expressiveness of robot motion has been studied in many contexts. The basic principle is rooted in neuroscience, which has suggested that the human brain is designed to detect and extract intention from biological motion [5]. Designing expressive motion has a long tradition in character animation (see [6] for a history of the development of the modern art). These animation principles have been applied to robot motion and have shown [7] that intent-expressive motions can improve people's perceptions of the robot's intelligence and their confidence in inferring the robot's goal. These findings have led to the development of motions designed specifically to signal the robot's intent [8] and methods to synthesize intent-expressive motions [9].

While many open-source software systems exist for controlling robots (such as Robot Operating System (ROS)[1]), systems to study the intent-expressiveness of robot motion are not widely available. Furthermore, such systems require a fair degree of generality to allow a wide range of objectives and constraints to be utilized. In our research to understand the qualities of effective human-robot collaboration, the University of Wisconsin–Madison Graphics Group and Human-Computer Interaction lab have developed several systems to address the needs of synthesizing intent-expressive motions and evaluating the effectiveness of these trajectories. We have built a general purpose trajectory optimizer that can synthesize robot motion using a wide variety of motion objectives and constraints. Our trajectory optimizer, Spacetime Trajectory Optimizer for 'Bot Expression (STrOBE), can optimize objectives and handle constraints using both the kinematic space of the robot and the Cartesian space the robot is operating in. These objectives and constraints can make use of the robot's joint and point velocities, as well as joint and point accelerations. Our evaluation framework, RobotSim, reads trajectories synthesized by STrOBE (or any other motion planner as long as they are formatted properly) and executes these trajectories on a

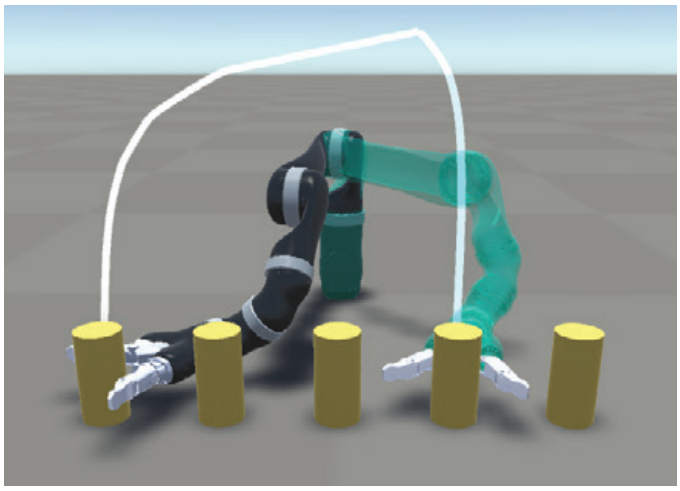

Fig. 1. Several samples of motions synthesized using STrOBE. These two motions use an objective designed to express the robot arm's intent to an observer. This objective is included as a built-in objective in STrOBE. The motions are simulated within the Unity game engine using elements from our evaluation framework, RobotSim.

[1] http://www.ros.org/

simulated robot game. RobotSim measures several metrics throughout the game to evaluate the motion's effect on collaborative performance with a human collaborator.

In this technical report, we describe the STrOBE and RobotSim systems. We discuss the design, implementation, and usage of each of these systems. We identify current limitations and how these limitation could be addressed. We hope that these systems, both planned for release as open-source software on the University of Wisconsin–Madison Graphics Group GitHub[2], will help the scientific community continue to explore human-robot collaboration and the design of intent-expressive motion.

II. STrOBE Trajectory Optimizer

To study how qualities of robot arm motion affect human-robot collaboration, our lab requires the ability to synthesize motions that have many different qualities and requirements. We require a system that provides flexibility in the choice of objective and constraints. To achieve this flexibility we developed an in-house trajectory optimizer built on the principles of spacetime constraints from animation literature. While other trajectory optimizers may be more efficient for optimizing to achieve minimal energy and avoiding collisions, our optimizer gives us the ability to drastically change the qualities and requirements of motions synthesized to achieve a number of desired outcomes. In this section we describe the design, implementation, usage, and limitations of STrOBE. We provide examples of synthesized trajectories and code to produce these trajectories.

A. A Brief Introduction to Spacetime

In the animation community trajectory optimization, known as *spacetime constraints* [10], provides a framework for exploring and understanding the properties of intent-expressive motions for robot manipulators. Trajectories are specified as the solution to variational optimization problems, that is, a trajectory that minimizes some objective function (defined over the trajectory) subject to a set of constraints. The constraints allow defining the requirements of the motion, while the choice of objective allows defining the desired properties of the movement. Flexibility in the objective allows choosing functions to create many effects, such as minimizing energy [10] or expressing intent [9], [11], [12].

We define a robot trajectory as a function that maps time to robot configurations, $\Theta : \mathbb{R} \rightarrow \mathbb{R}^n$ where n is the number of degrees of freedom, so that the configuration along a trajectory at time t is $\Theta(t)$. We denote the kinematic function that maps from configuration space to a position, the robot's forward kinematics, as FK, such that the end effector position at time t is $FK(\Theta(t))$.

Optimizing the variational problem results in a trajectory:

$$\Theta^* = \text{argmin}\, g(\Theta) \text{ subject to } c_i \diamond k_i,$$

for the time range t_0 to t_f. The objective function g is a function over the trajectory that returns a scalar value. Each of the constraints c_i is either $\diamond \in \{=, \leq, \geq\}$ to a constant k_i. These constraints allow the requirements for the motion and the environment to be defined. For example we can constrain the initial configuration of the robot with an initial pose constraint ($\Theta(t_0) = k_0$). A simple example of an objective function is to minimize the length of the trajectory in configuration space (minimal joint movement):

$$\int_{t_0}^{t_f} \|\Theta'(t)\|^2 dt. \tag{1}$$

By defining our problem as above, we can explore a large space of different objectives and/or constraints. In the next subsections we describe the design and implementation of our spacetime constraints optimizer.

[2]https://github.com/uwgraphics

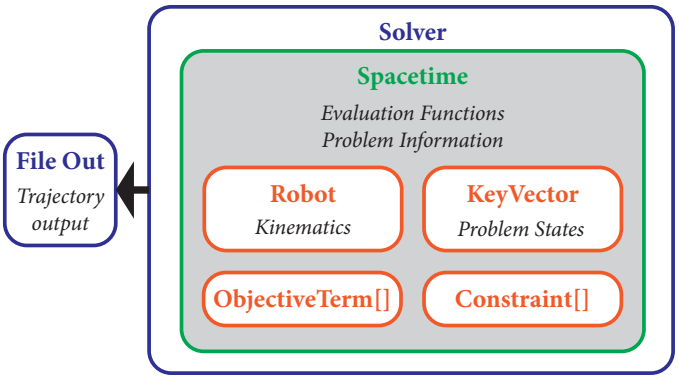

Fig. 2. The high-level design of our STrOBE trajectory optimizer. A Robot defines the state and point variables by its configuration and kinematics. A Spacetime problem is initialized with the robot and the number of keyframes for the KeyVector. ObjectiveTerms and Constraints are added to the problem as required by the motion design and environment. Finally, a Solver is passed a Spacetime problem and solves the problem using the Spacetime evaluation functions. Optionally, the solution can be written to a file.

B. STrOBE Design

Spacetime constraints provides a very generalized formulation for solving a wide variety of motion optimization problems. It allows us flexibility in the objective and the constraints. Thus, it is an ideal formulation to use for our trajectory optimizer. STrOBE is designed to solve this very flexible problem space.

To solve the variational optimization problem of spacetime constraints, we discretize the robot's trajectory and approximate the objective function with finite differences sampled along time. This discretization leads to a non-linear programming problem over the variables of the representation of the trajectory that can be modeled using automatic differentiation and solved using commonly available variants of Sequential Quadratic Programming (SQP) as described by [10] and [11]. These these earlier implementations use bespoke solvers. We have implemented STrOBE with freely available tools from standard libraries (the Python `ad` automatic differentiation package and the SLSQP solver from `scipy`). We note that for the class of spacetime objectives we consider, we cannot use per-frame iterative approaches [13].

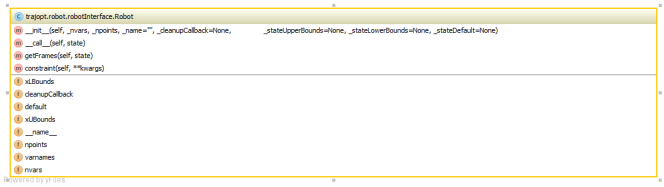

Fig. 3. Class diagram for the Robot class.

The basic, high-level design of STrOBE is shown is Figure 2. The system synthesizes trajectories by building the problem from the bottom up. First, a `Robot` is defined by its kinematics. All robots are dervived from the `Robot` base class. Next, each `ObjectiveTerm` for the desired properties of the motion must be defined. All objectives are derived from `ObjectiveTerm`. Next, the requirements of the motion (due to the environment, robot, goals, etc.) are defined as `Constraints`. All constraints are derived from `Constraint`. With these building blocks defined, the `Spacetime` problem is initialized with the robot and number of keyframes. A `KeyVector` is created from the robot's state space repeated for the number of keyframes. `ObjectiveTerms` and `Constraints` are added to the `Spacetime` problem using member functions. Finally, with the problem fully defined, the `Spacetime` problem is passed to the `Solver`. The `Solver` utilizes the evaluation functions provided by `Spacetime` to minimize the `ObjectiveTerms` over the `KeyVector` subject to the `Constraints`.

C. STrOBE Implementation

This section describes each of the classes and their member functions. We describe how these classes integrate into the overall system. We also discuss the built-in extensions to our base classes that we have developed for our own research. The built-in classes are included in the framework and provide a variety of useful robots, objectives, and constraints as well as a solver implementation.

1) Robot: `Robot` is an abstract class that defines the interface for specifying the kinematics and state space for the `Spacetime` problem. A `Robot` has a certain number of points (for example the joints of a robot arm) and a certain number of variables in the state space to be optimized over. These counts, a robot name, and a cleanup callback function (used to perform post processing functions such as removing erroneous spinning of joints) are used to initialize a `Robot`. `Robots` are treated as functions of state that when called return a list of the point positions of the robot for that state. A `Robot` also provides a `getFrames` function that returns the same list of points for a given state, as well as a list of 3×3 transformation matrices for each point. `Robot` has two members, `xUBounds` and `xLBounds`, that can be set to define the upper and lower bounds for each state variable. Finally, a `Robot` may define its own internal constraints that are called the same as `Constraint` objects. Figure 3 details the members of the `Robot` class.

TABLE I
STROBE CONSTRAINT AND OBJECTIVE PARAMETERS

Parameter	Description
t	The current keyframe.
state	The state vector at keyframe t.
points	The point vector at keyframe t.
frames	The transformation matrices for the robot at keyframe t.
stvel	The state velocities vector at keyframe.
stacc	The state accelerations vector at keyframe.
ptvel	The point velocities vector at keyframe t.
ptacc	The point accelerations vector at keyframe t.

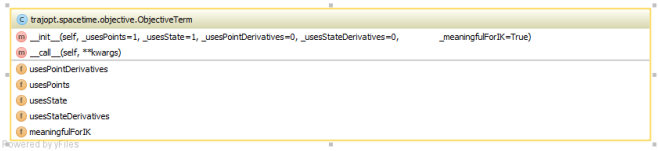

Fig. 4. Class diagram for the ObjectiveTerm class.

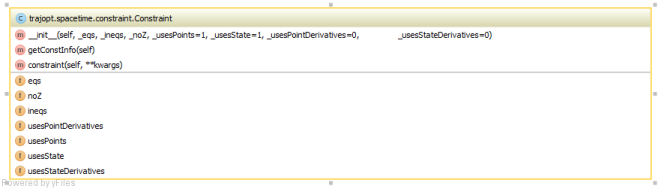

Fig. 5. Class diagram for the Constraint class.

STrOBE also includes an extension of `Robot` called `Arm`. The `Arm` class implements a `Robot` and provides additional functions to evaluate the robot state using the kinematic description (axes of rotation, joint offsets, and rotations) of a robot arm. Three of our built-in robots discussed later extend the `Arm` class.

2) ObjectiveTerm: `ObjectiveTerm` is an abstract class that defines the interface to specify properties of the motion that we are trying to achieve through optimization. The interface is fairly simple, an `ObjectiveTerm` is a callable object with variable parameters. When called an `ObjectiveTerm` calculates the scalar value of this term in the objective for the current state and returns the squared sum of that value, $\|ObjectiveTerm\|^2$, to the caller. Table I lists the parameters that are available to `ObjectiveTerm` calls for performing calculations. An `ObjectiveTerm` is initialized by providing four flags to specify whether the term uses the current state, current point positions, current state velocity, and/or current point velocity. These flags determine what information is needed by the `ObjectiveTerm` to calculate the objective value. Figure 4 details the members of the `ObjectiveTerm` class.

3) Constraint: `Constraint` is an abstract class that defines the interface for a strict requirement of the motion that must be met. `Constraint` can be either an equality, inequality, or both. By convention, equality constraints are satisfied when they evaluate to 0 and inequality constraints are satisfied when they evaluate to greater than or equal to 0. Like

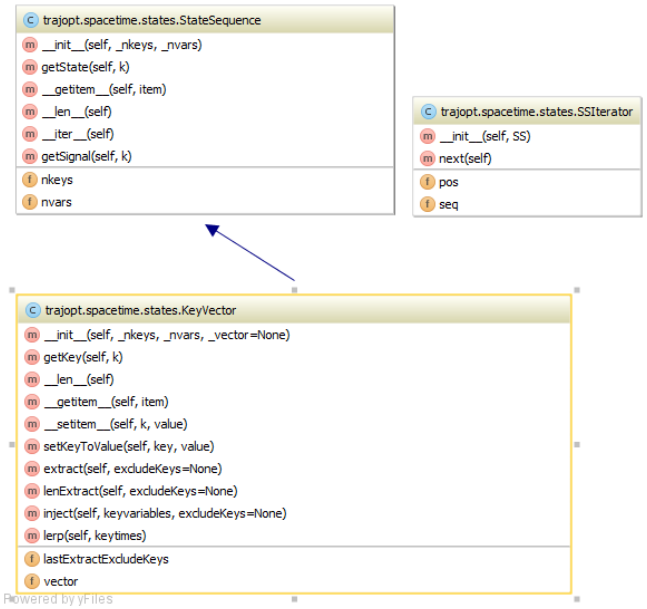

Fig. 6. Class diagram for the KeyVector and StateSequence classes.

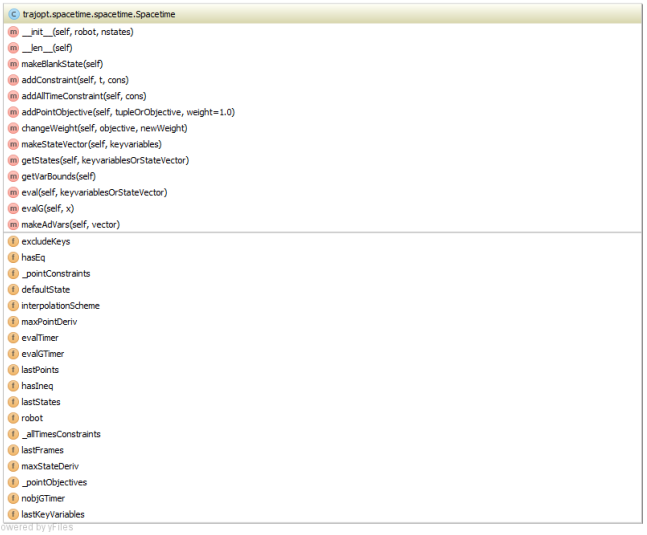

Fig. 7. Class diagram for the Spacetime class. The class contains the robot and problem description. It provides member functions, used by a Solver, to evaluate the current solution state.

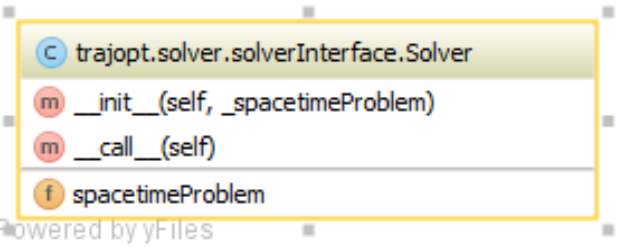

Fig. 8. Class diagram for the Solver class. The class contains is initialized with a Spacetime problem and utilizes the Spacetime evaluation functions to optimize the KeyVector for the given ObjectiveTerms and Constraints.

`ObjectiveTerm`, a `Constraint` is initialized by providing four flags to specify whether it uses the current state, current point positions, current state velocity, and/or current point velocity. Additionally, `Constraint` is initialized with three boolean values that determine whether it considers the third dimension (Z axis), whether it has an equalty component, and whether it has an inequality component. `Constraint` is not a callable object, but instead is evaluated by calling the `constraint` function using the variable parameters from Table I. Calls to the `constraint` function return two lists of evaluations, an equality list and an inequality list. If a `Constraint` is not an equality or inequality it should leave that list empty. Figure 5 details the members of the `Constraint` class.

4) KeyVector: `KeyVector` and `StateSequence` provide storage and functions for the robot configuration in each keyframe of the `Spacetime` problem. `StateSequence` is an abstract class that stores the number of keyframes and the number of state variables. `KeyVector` implements `StateSequence` and stores the value of all active state variables for each of the keyframes in a large list. `KeyVector` provides functions to extract and write the state at a given keyframe. Figure 6 details the members of the `KeyVector` and `StateSequence` classes.

5) Spacetime: `Spacetime` contains all of the problem parameters (the robot, constraints, and objective terms) and provides functions to evaluate the current problem state. Figure 7 details the members of the `Spacetime` class.

A `Spacetime` problem is initialized with a `Robot` and the number of keyframes to optimize over. The `Spacetime` class will use the `Robot` number of variables and the number of keyframes to create the initial `KeyVector`. A `Constraint` can be added at a specific keyframe by passing the keyframe and `Constraint` to the `Spacetime` member function `addConstraint`. A `Constraint` can be added at all keyframes by passing the `Constraint` to the `Spacetime` member function `addAllTimeConstraint`. An `ObjectiveTerm` can be added to the problem by calling the `Spacetime` member function `addPointObjective` with the `ObjectiveTerm` and a weight for the term. The weight can be changed by calling the `Spacetime` member function `changeWeight` with the `ObjectiveTerm` and the new weight. Finally, the `Spacetime` member list `excludeKeys` can be used to specify keyframes that should should not consider `Constraint` objects when optimizing the trajectory.

When a `Spacetime` problem is being solved, the `Solver` can make use of the `Spacetime` member functions `eval` and `evalG`. `eval` returns the sum of `ObjectiveTerm` values, the list of equality `Constraint` evaluations, and the list of inequality `Constraint` evaluations for the current state vector. `evalG` returns these three evaluations as well as the derivatives for each of the three evaluations.

6) Solver: `Solver` defines the interface to a solver that minimizes the summation of each `ObjectiveTerm` while still satisfying each `Constraint` for the `Robot` and number of keyframes defined in the `Spacetime` problem. The `Solver` is initialized with a `Spacetime` problem (and any additional parameters needed by the specific solver). The `Solver` is a callable object and when called performs optimization by utilizing the `Spacetime` evaluation function, `eval` and `evalG`. The output of the call should return the optimized the state vector (a `Robot` state for each keyframe) for the given `Spacetime` problem. Figure 8 details the members of the `Solver` class.

We include an implementation of a sequential quadratic programming (SQP) solver, `SLSQPSolver`. We have used this solver exclusively to date. However, there is no reason why other solvers could not be implemented as long as they implement the `Solver` class correctly. `SLSQPSolver` provides additional solver parameters:

- `_iter`: The maximum number of iterations.
- `_acc`: The accuracy threshold.
- `_x0`: The intial condition of the state vector.
- `_verbose`: Toggles whether solver termination information is output.
- `_callback`: The optional post processing callback.
- `_doBounds`: Whether to use the `Robot` state variable bounds (`xUBounds` and `xLBounds`).

7) Built-In Extensions: While the space of potential extensions for objectives, constraints, and robots is virtually infinite, we have built some of these constructs into STrOBE for our own research. These are included in the optimizer because we have found them to be particularly useful. These built-in extensions can use used by including them from the proper "built-in" Python file. Figure 9 details our built-in extensions.

Our built-in `ObjectiveTerms` are:

- `StateValue`: Minimizes the distance from the problem's state vector to a given state vector.
- `StateVelocity`: Minimizes the state space velocity (straight lines in state space).
- `StateAcceleration`: Minimizes the state space acceleration (smooth curves in state space).
- `PointVelocity`: Minimizes the point velocity of a given `Robot` point.
- `PointAcceleration`: Minimizes the point acceleration of a given `Robot` point.
- `ElbowVelocity`: Minimizes the `PointVelocity` for all of the `Robot` points.
- `AncaLegible`: The legibility objective proposed by Dragan et al. [9].
- `ProbFieldLegible`: Minimizes the value of the derivative of a Guassian distribution centered at a goal position for a selected `Robot` point.
- `LegibleS`: Minimizes the vector from a selected `Robot` point to a goal position.
- `LegibleG`: Minimizes the vector from an user inferred position to a goal position.

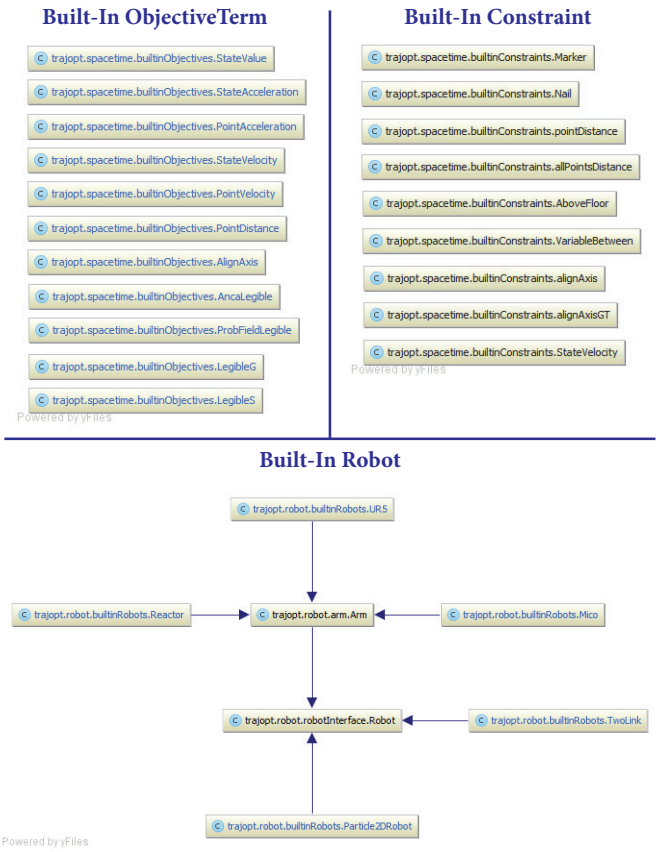

Fig. 9. The listing of built-in ObjectiveTerms, Constraints, and Robots currently provided by the STrOBE trajectory optimizer. Each extends either the base ObjectiveTerm, Constraint, or Robot classes to achieve a desired motion property, motion requirement, or robot kinematics, respectively.

- `PointDistance`: Attempts to keep a selected `Robot` point at least a given distance from a specified point.
- `AlignAxis`: Keeps the dot product (alignment) of the selected axis of a `Robot` point and a given vector as close to 1 as possible.

Our built-in `Constraints` are:

- `Marker`: Not really a constraint! This simply marks a given position, but returns empty lists.
- `Nail`: An equality `Constraint` that keeps a selected `Robot` point at a specified position.
- `alignAxis`: An equality `Constraint` that keeps the dot product of the selected axis of a `Robot` point and a given vector equal to a given value.
- `alignAxisGT`: An inequality `Constraint` that keeps the dot product of the selected axis of a `Robot` point and a given vector greater than a threshold.
- `AboveFloor`: An inequality `Constraint` that keeps a selected `Robot` point greater than a threshold along the Y axis.
- `VariableBetween`: An inequality `Constraint` that keeps a selected state variable within a certain range.
- `StateVelocity`: An inequality `Constraint` that keeps a selected state variable's velocity below a certain thresh-

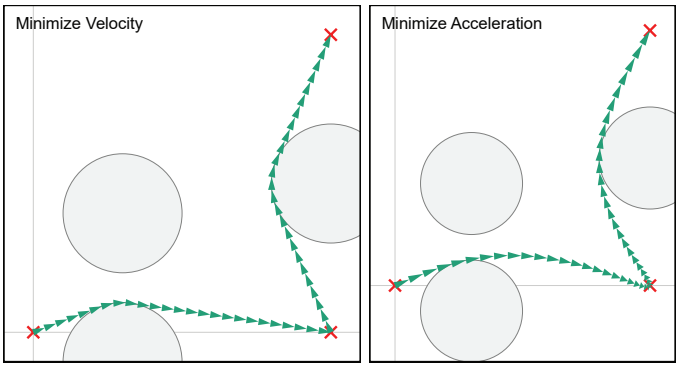

Fig. 10. An example of different trajectories synthesized for a point robot with STrOBE. The point robot moves from the starting location through an intermediate waypoint using two different objectives: minimum velocity (left) and minimum acceleration (right). The point robot must also avoid three obstacles in its path (gray circles). The start, intermediate, and end points are specified using position constraints at given time steps (begin, end/2, end). The obstacles are specified by all-time minimum point distance constraints.

old.

- `pointDistance`: An inequality `Constraint` that keeps a selected `Robot` point's distance from a given position greater than a minimum distance.
- `allPointsDistance`: An inequality `Constraint` that keeps each of the `Robot` points' distances from a given position greater than a minimum distance.

Finally, our built-in `Robots` are:

- `Particle2DRobot`: A 2D particle robot specified by a single point.
- `TwoLink`: A simple 2D robot with two links and two joint variables (a base joint and a joint connecting the links).
- `UR5`: An extension of the `Arm` subclass of `Robot`. Specifies the kinematics for a Univeral Robots UR5 and has six joint variables.
- `Mico`: An extension of the `Arm` subclass of `Robot`. Specifies the kinematics for a Kinova Mico and has six joint variables.
- `Reactor`: An extension of the `Arm` subclass of `Robot`. Specifies the kinematics for a Trossen Robotics PhantomX Reactor and has five joint variables.

D. STrOBE Usage

This section describes how to make use of our trajectory optimizer and provides examples of using STrOBE to produce various trajectories. Using STrOBE to synthesize trajectories is a straightforward process:

1) Create a `Robot`.
2) Create a `Spacetime` problem initialzed with the `Robot` and number of keyframes.
3) Add `ObjectiveTerms` and problem `Constraints` to the `Spacetime` problem.
4) Initialize the `Solver` with the `Spacetime` problem and any relevant parameters. Note: It is a good idea to help the solver by setting known states (due to constraints for example) of the state vector in the initial condition.
5) Call the `Solver` to solve the `Spacetime` problem.

Figure 10 shows a simple example of optimizing a point robot's motion for two different objectives in an environment with three obstacles. Figure 11 shows motions synthesized for two of the built-in robot arms using two different objectives. To provide a concrete example of using STrOBE, the following Python script produces the solution on the left in Figure 10:

```
from trajopt.solver.slsqp import SLSQPSolver
from trajopt.spacetime.spacetime import Spacetime
from trajopt.robot.builtinRobots import Particle2DRobot
from trajopt.spacetime.builtinObjectives import PointVelocity
from trajopt.spacetime.builtinConstraints import Nail, pointDistance

# Timesteps and Waypoints
nsteps = 51 # number of keypoints

last = nsteps – 1
middle = last/2

startPoint = (0,0,0)
midPoint = (5,0,0)
endPoint = (5,5,0)

pointID = 0 # the point we are interested in
pointVelocityWeight = 1.0

# Create Robot
robot = Particle2DRobot(1)

# Create Spacetime Problem
st = Spacetime(robot, nsteps)

# Setup Objective (minimum velocity)
st.addPointObjective(PointVelocity(pointID), pointVelocityWeight)

# Setup Constraints
st.c0 = Nail(pointID, startPoint, True) # start
st.c1 = Nail(pointID, midPoint, True) # intermediate waypoint
st.c2 = Nail(pointID, endPoint, True) # end
st.c3 = pointDistance(pointID,1,5,2.5,0,True) # obstacle
st.c4 = pointDistance(pointID,1,1.5,–0.5,0,True) # obstacle
st.c5 = pointDistance(pointID,1,1.5,2,0,True) # obstacle

st.addConstraint(0, st.c0)
st.addConstraint(middle, st.c1)
st.addConstraint(last, st.c2)
st.addAllTimeConstraint(st.c3)
st.addAllTimeConstraint(st.c4)
st.addAllTimeConstraint(st.c5)

# Help the solver a bit by configuring known states in x0
st.defaultState[last] = [endPoint[0], endPoint[1]]
st.defaultState[middle] = [midPoint[0], midPoint[1]]

# Create Solver and solve the Spacetime problem
solver = SLSQPSolver(st,_callback=True)
solution = solver()
```

E. STrOBE Limitations

STrOBE does have some limitations which could be addressed to improve its usefulness. Major limitations in the current version are no guarantee of convergence, lack of robot dynamics, lack of support for non-uniform time steps, lack of

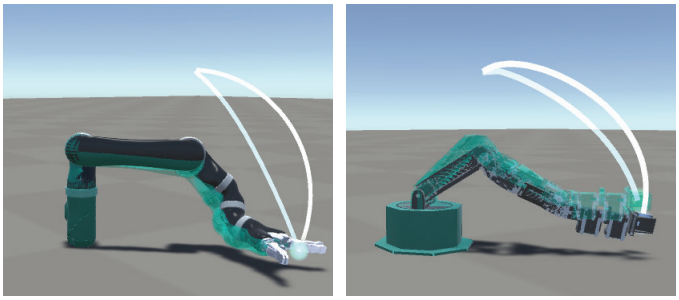

Fig. 11. An example of different trajectories synthesized for two robot arms included in STrOBE as built-in robot arms, a Kinova Mico (left) and a PhantomX Reactor (right). The robot arms move from the starting configuration to the ending configuration using two different objectives: minimum joint velocity (white) and minimum end-effector point velocity (green). The start and end configurations are specified using state constraints at given time steps (begin, end).

collision detection, and lack of support for complex geometry. Additional features that could improve usability include parsing Unified Robot Description Format (URDF) files to import new robots as well as ROS and MoveIt integration.

The extremely general nature of our problem definition and implementation is a great asset, but also carries an important limitation. STrOBE is not guaranteed to converge to a solution. This is why it is extremely important to "help" the solver by setting up the initial condition, `defaultState`, to contain any known (constrained) states. Another technique we use to encourage convergence is to start from an initial trajectory. We often perform interpolation from the known starting state to an ending state to initialize the solver.

Robot dynamics are not currently implemented within the framework. However, robot dynamics can be implemented within the spacetime framework by describing them as constraints of the robot. This approach was described by Witkin and Kass [10] in their implementation of spacetime constraints. To date we have not had a need for robot dynamics as our research has focused mostly on end-effector paths. Thus, robot dynamics are not currently implemented.

Currently, STrOBE assumes that keyframes are spaced one time step apart. However, it is conceivable that instead of finding the robot state at the next time step one may want to allow timesteps to vary in length. In particular, we may want to distribute keyframes more densely at times with high curvature. We may also want to add timing constraints that adhere to the dynamic limits of the robot. Adding support for timing information is an important feature for a future version.

STrOBE does not currently support complex geometries and as such there is no collision detection. Currently, the system does support adding constraints to maintain a minimum distance from a point. So, collisions can be avoided by adding point distance constraints. However, an important future feature is to support complex geometric models, so planning can be achieved in complex scenes.

Finally, adding ROS support would greatly improve the system's ability to be used with a wide variety of robots as well as driving robots directly from the optimizer. This could potentially be achieved without much modification to the system since ROS supports Python.

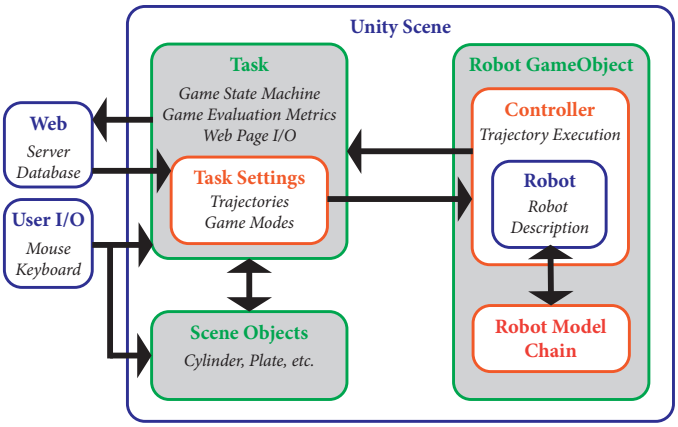

Fig. 12. The high-level design of our RobotSim testing framework. Within a Unity scene two main components drive the game, the Task and the Robot. The Robot is a chain of Unity GameObjects that includes the robot geometry and a Controller script. The Controller script receives new trajectories from the game and executes them on the specific Robot. The Task handles game play, human I/O, game settings, measurement, and communication with the external web site.

III. RobotSim Study Framework

Our main interest in trajectory optimization to-date has been optimizing robot arm motions to convey intent. To evaluate the effectiveness of intent-expressive trajectories synthesized by STrOBE we designed a task and modeled it in the Unity game engine[3]. This section describes the design and implementation of our human-robot collaborative performance study framework, RobotSim. This section assumes a reader has a working knowledge of the Unity game engine. If you would like to try the game yourself a demonstration is available online[4].

A. RobotSim Design

RobotSim is designed as a collection of components that can be used to simulate a robot task and also allow for humans to interact with the simulation. The framework is composed of two main concepts, `Robot` and `Task`. Currently in RobotSim we provide three robots, and one task. Each robot is actuated by a `Controller`, which moves between trajectory `Poses` by fitting and following a `SmoothLinearSpline` or using basic point interpolation (`PiecewiseLinearFunction`). The robot executes these trajectories to perform a portion of a task. The human playing the game performs the other portion of the task. Measurements are collected at the frame-rate of the game (typically 60 Hz) to evaluate the collaborative performance of the human and robot. Figure 12 provides a basic overview of the RobotSim framework.

Currently, the RobotSim framework supports three robot arms: a Trossen Robotics PhantomX Reactor[5], a Kinova

[3]https://unity3d.com/
[4]http://graphics.cs.wisc.edu/Demos/RobotSim/
[5]http://www.trossenrobotics.com/

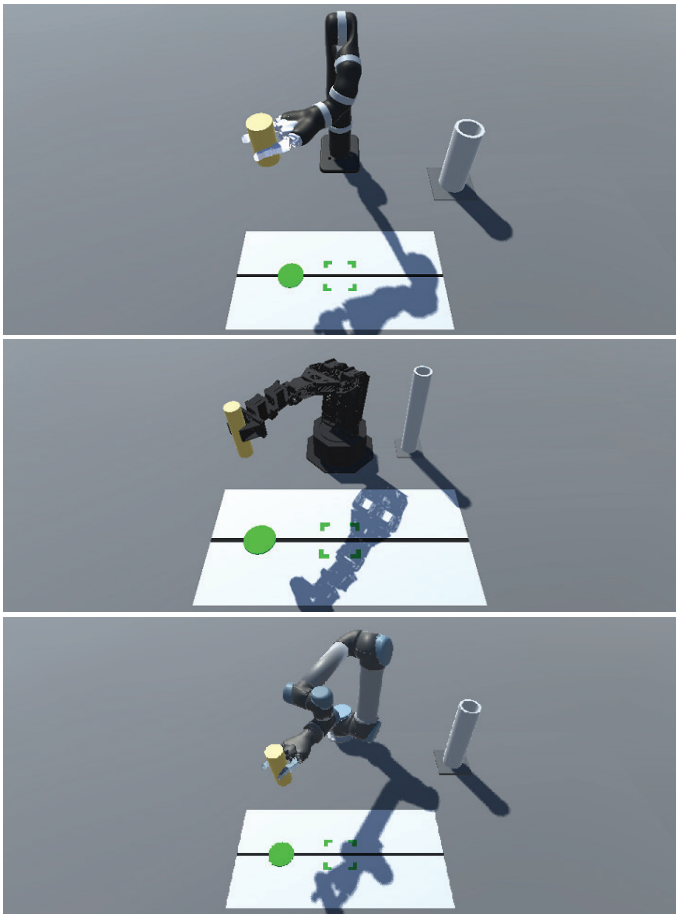

Fig. 13. The RobotSim testing framework with each of the three supported robot arms: Mico (top), Reactor (middle), and UR5 (bottom). The robot arm picks up the cylinder from the feeder and places it on the track (the black line). Users control the plate to try and "catch" the cylinder.

Mico[6], and a Universal Robots UR5[7]. The choice of these robots has mostly been due to availability. Our lab currently owns a PhantomX Reactor and a Mico arm. This allows us to test robot trajectories in simulation before testing them in person, which typically requires longer study times due to the serial nature of in-person robot studies. The choice of arms also provides a good sample of different robot arm kinematics with different degrees of freedom (DoF): the Reactor has only 5-DoF and a human-arm-like shoulder-elbow-wrist configuration, the Mico has 6-DoF and a non-anthropomorphic series of axial wrist joints with angular offsets, and the UR5 has an anthropomorphic design with more capability than the Reactor because of its 6-DoF.

The RobotSim framework includes a single collaborative task, `Game_1DWorkspace`, at this point in time. In this task, the robot places cylinders at locations along a linear track, and a participant moves a plate (controlled by his/her mouse) where they think the robot is going to place the cylinder. We typically motivate this task as performing a chemistry experiment by combining the reactive components (the plate and cylinder). The game measures how the participant's inference of the goal position changes over time. These measurements allow us to compare collaborative performance created by various motion types. To encourage participants to make their best guess at all times without waiting, we have included an optional "contact loss" mechanism. This mechanism simulates loss of control of the plate by creating static on the game camera and preventing the user from moving the plate. Figure 13 shows the simulated task for each of the three robot arms.

[6]http://www.kinovarobotics.com/
[7]http://www.universal-robots.com/

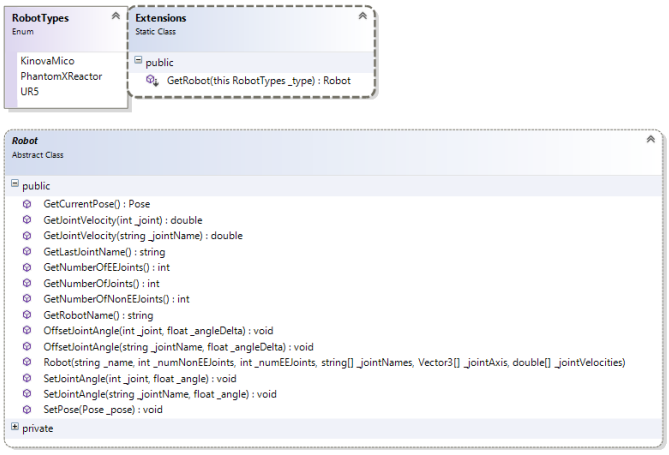

Fig. 14. Class diagram for the Robot class.

B. RobotSim Implementation

In this section we detail the various classes and other structures used in RobotSim. The function, structures, and usage of each component are briefly described.

1) Robot GameObject Prefab: Each robot in RobotSim has a chain of Unity `GameObjects` which include the meshes and textures used to render the robot. These `GameObjects` are attached to each other as a kinematic chain starting from the base link and ending at the end-effector. The chain's parent is an empty `GameObject` with the same name as the robot. This root `GameObject` is tagged as a `Robot` and includes a `Script` component of type `Controller`. In the options for the `Script` component, the correct `RobotTypes` is selected from a drop down list. The `Controller` script provides control of the model of the robot. This chain is saved as a Unity prefab object for easily importing into new scenes.

The three available robot models currently supported by RobotSim and their kinematic descriptions are from open-source ROS packages. The three supported robots are: a Universal Robots UR5[8], a Kinova Mico[9], and a Trossen Robotics PhantomX Reactor[10].

2) Robot: `Robot` is an abstract class designed to allow communication between the generic `Controller` class and the `Robot GameObject Prefab`. For each actual robot,

[8]https://github.com/ros-industrial/universal_robot
[9]https://github.com/RIVeR-Lab/wpi_jaco
[10]https://github.com/a-price/reactor_description

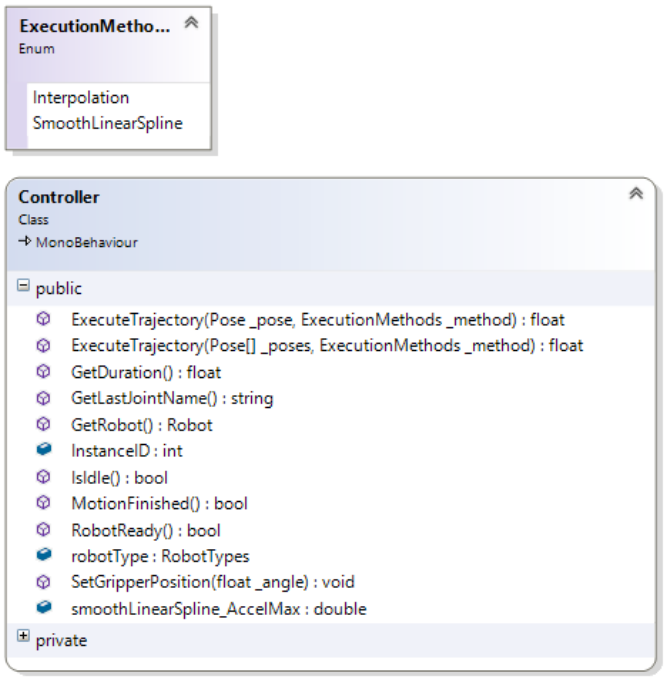

Fig. 15. Class diagram for the Controller class.

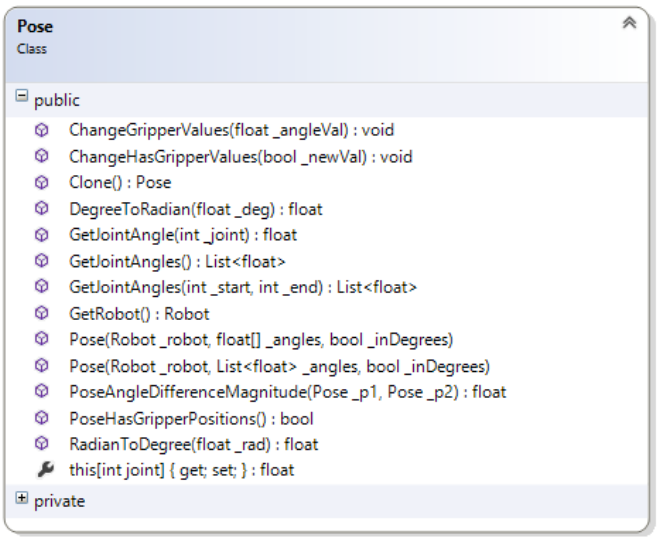

Fig. 16. Class diagram for the Pose class.

the `Robot` class is extended and the following members are initialized: `NAME`, `NUM_NON_EE_JOINTS`, `NUM_EE_JOINTS`, `JOINT_NAMES`,`JOINT_AXIS`, and `JOINT_VELOCITIES`. These members provide information about the kinematics and prefab link names to the program. See *Adding A New Robot* (Section III-D) for more details on these members.

Figure 14 details the public members provided by `Robot`. These functions allow querying the parameters described above as well as retrieving (`GetCurrentPose`) or setting (`SetPose`, `SetJointAngle`) the current joints angles. Also shown in the figure is the `RobotTypes` enumeration, which stores the currently available robot types.

3) Controller: `Controller` is the class responsible for executing trajectories on the current scene `Robot`. `Controller` is derived from the Unity `MonoBehaviour` class; so, it updates every frame when Unity calls its `Update` function. The `Controller` should be attached to the root `GameObject` of the robot prefab. The public member `robotType` allows selecting the correct robot from the available options in the `RobotTypes` enumeration.

`Controller` public member functions are shown in Figure 15. These members provide the ability to execute trajectories (`ExecuteTrajectory`) that are specified as either a single `Pose` or a `Pose` array. The `Controller` takes the `Poses` to be executed and constructs a `TimeFunction` for each `Robot` joint from the current position of the `Robot` to the last `Pose` making sure each `Pose` is passed through. The `TimeFunction` used is determined by which `ExecutionMethods` enumeration option is passed to the execution function.

Other members allow querying whether the `Robot` is currently idle or ready to execute a new trajectory, setting the end-effector position, querying the currently executing trajectory's duration, and querying the `Robot` object that the `Controller` is currently controlling.

4) Pose: `Pose` is a class to store a single configuration of the current `Robot`. A `Pose` contains a double value for each of the non end-effector joints and can include gripper positions as well. `Poses` are initialized from a joint angle list or array. These structures should be of length `NUM_NON_EE_JOINTS` if no gripper information is to be changed or of size `NUM_NON_EE_JOINTS` + `NUM_EE_JOINTS` if the gripper value should change. Joint values can be passed as either degrees or radians. Values for specific joints can be updated or retrieved individually. `Pose` provides methods to convert between radians and degrees as well as take the difference in magnitude between two `Poses`. Figure 16 details the public interface for this class.

5) TimeFunction: A `TimeFunction` is an abstract class to store a continuous function of time. A `TimeFunction` allows assembling a `Pose` array into continuous trajectory functions for each joint. This is necessary for the RobotSim framework because the game updates at high frame rates. A function is necessary to fill in the information between subsequent `Poses` to allow the `Robot` to move smoothly.

`TimeFunction` is currently implemented by two classes, `SmoothLinearSpline` and `PiecewiseLinearFunction`. `PiecewiseLinearFunction` provides simple linear interpolation between subsequent `Poses`. `SmoothLinearSpline` provides simple linear interpolation except near boundaries. Corners are smoothed using 5^{th} degree polynomials. The options provided by the `ExecutionMethods` enumeration should be used to select between these two function types for trajectory execution. Both function types are constucted by passing time (in seconds) as the `x_data` and the joint angle values as the `y_data`. `SmoothLinearSpline` includes an

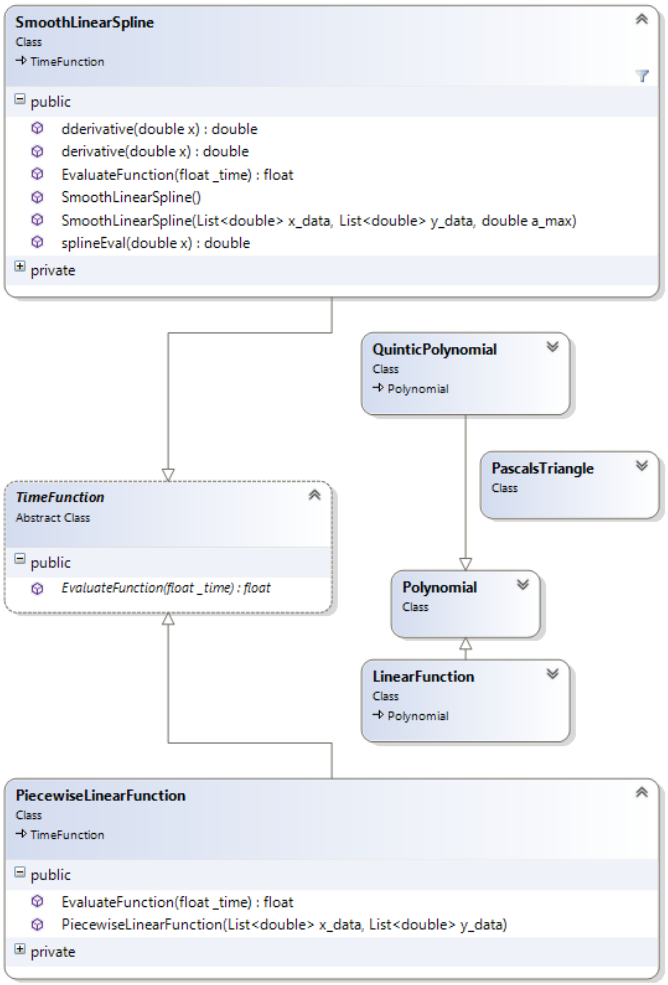

Fig. 17. Class diagram for the FunctionsOfTime classes.

additional parameter to limit the maximum corner curvature.

`Polynomial` and its subclasses, `LinearFunction` and `QuinticPolynomial`, are used to support `SmoothLinearSpline` and `PiecewiseLinearFunction`. All of these function classes are simplified C# implementations of constructs found in the open-source *ecl_geometry*[11] ROS package. Figure 17 details the `TimeFunctions` and their members.

6) Game_1DWorkspace & Settings_1DWorkspace: `Game_1DWorkspace` contains the game's state machine and measures data throughout each trial motion. `Settings_1DWorkspace` sets and maintains the game's settings. `Settings_1DWorkspace` reads selected trajectory files and provides functions to query and advance the current trajectory file being executed by the game. The public members for these classes are shown in Figure 18.

Settings should be passed to the game by passing a single string to the `ConfigRun` function provided by `Game_1DWorkspace`. To call `ConfigRun` the serving web page should invoke the JavaScript

[11]https://github.com/stonier/ecl_core/tree/indigo-devel/ecl_geometry/

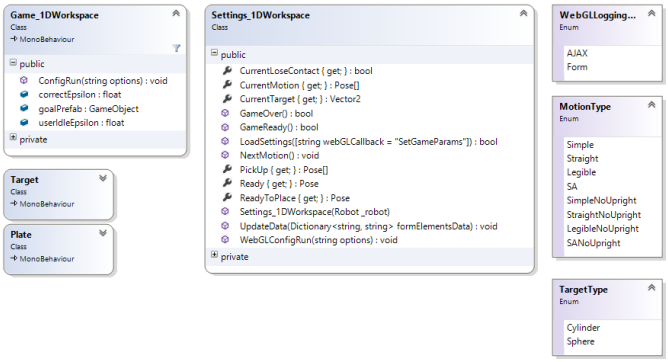

Fig. 18. Class diagram for the Game_1DWorkspace and Settings_1DWorkspace classes.

function `SendMessage("Main Camera","ConfigRun", <SETTINGS STRING>);`. This function is provided by Unity in the compiled WebGL game to allow a web page to interact with the game. The settings string should be formatted as `"<SETTING>=<VALUE>\n <SETTING>=<VALUE>\n ..."`. All of the possible `Settings_1DWorkspace` settings are described in Table II.

At the end of each motion, `Game_1DWorkspace` calls the `Settings_1DWorkspace` `UpdateData` function. `UpdateData` invokes the callback function `callName` specified in the settings. The callback function should be implemented externally in JavaScript to handle the data from the game for this trial. The external callback should either update a form element on the web page with hidden fields for each data value or provide an AJAX call to write the data to a database. The callback mode is specified by the option chosen from the `WebGLLoggingMode` enumeration. Table III describes all of the data values passed back after each motion.

7) Other Task Classes: Two additional classes support our simulated task, `Target` and `Plate`. `Target` is a Unity `MonoBehaviour` that should be attached to the scene's cylinder that the robot places. `Target` allows the cylinder to attach to the robot end-effector as well as translate up and down in the feeder. `Plate` is also a Unity `MonoBehaviour` that should be attached to the scene's user controlled plate. `Plate` translates the plate object to follow the user's mouse and allows disabling motion when contact loss occurs.

C. RobotSim Usage

RobotSim's primary purpose is to be utilized as a web-based game. To support this operation a web framework should be developed to serve the game and provide a backend to record measurements collected by the game. For our experiments we have developed a study framework using PHP and a MySQL database. The basic requirements of the web framework are to check browser requirements (Google Chrome or Mozilla FireFox with HTML5 and WebGL support), provide the game configuration parameters via JavaScript, and store collected data.

TABLE II
ROBOTSIM GAME SETTINGS

Setting	Default	Description
MotionType	0	Selects the motion type to use. This should be a single integer for an option in the MotionTypes enumeration.
AllowContactLoss	False	Is contact loss used {True, False}?
WebGLLoggingMode	0	Selects the external data callback mode to use. This should be a single integer for an option in the WebGLLoggingMode enumeration.
WebGLLoggingCallname	WriteObjectiveAJAX	The name of the external JavaScript callback to pass data to the web page.
WegGLLoggingFormname	LoggingForm	The ID of the form element that will store the data. This is used for WebGLLoggingMode.Form.
MotionIndices	new int[0]	A comma separated list of buckets, {1, ..., 6}, to select motions from (in the correct trial order).
LoseContactIndices	new bool[0]	A comma separated list, {True, False}, to determine if a trial has contact loss (in the correct trial order).

TABLE III
ROBOTSIM MEASURED DATA

Value	Description
TimePoints	A comma separated list of times. These are the times each of the other raw data points are recorded at.
MousePositionPoints	A comma separated list of mouse positions on the track.
ErrorPoints	A comma separated list of error magnitudes from the plate position to the robot's target position.
TimeFirstCorrect	The time when the participant predicted the target and did not move $\pm\epsilon$ from the target for the rest of the trial.
TotalMouseMovement	The integral of MousePositionPoints.
AreaUnderCurve	The integral of ErrorPoints.
CriticalPointPosition	The position of the plate when contact loss occurred (or would have occurred).
UserIdleTime	The time it took after the motion started for the participant to move the mouse.
AvgFPS	The average game frame rate.
MinFPS	The minimum game frame rate.
MaxFPS	The maximum game frame rate.
PathDuration	The total duration of the motion.
LostContactTime	The time when contact loss occurred (or would have occurred).
LostContact	Was there contact loss {1,0}?
WriteTime	The time/date the current trial completed.
TargetPosition	The target position of the robot's motion.
CoordSlope	The slope of linear regression performed on ErrorPoints from UserIdleTime to TimeFirstCorrect.
InvalidTrial	Did the participant move the mouse at all {1,0}?

Building the game is straightforward. Unity can build the game on a variety of platforms including WebGL, Linux, Windows, OSX, etc. To utilize the web-based build option navigate to `File` → `Build Settings...`, select the WebGL build option, and make sure the correct scene is selected to build. A prompt will ask for an output directory. Select a directory and build the game. The built files can be directly integrated into a web framework. Other platforms can potentially be used, but may require additional modifications.

Because of the modularity of the different components, it is relatively easy to take pieces from RobotSim and move them into other scenes built in Unity. The Robot and Controller scripts can be packaged into a Unity asset package with the necessary geometry and material files. These assets can easily be imported into other Unity projects and scenes. Please read the Unity documentation for more details on exporting/importing these assets packages.

In the next subsections we describe how to add new robots and motion types into the current framework. These are the most likely additional features a user might want to add and the most practical to describe since building new scenes is near limitless in scope.

D. Adding A New Robot

Adding a new robot is a fairly simple process. This section describes building a new robot that can be controlled using the `Controller` class and how to integrate motion files for a new robot into the `Game_1DWorkspace` task.

The first step to add a new robot is to build the robot `GameObject` chain. First add an empty `GameObject` into the scene and give it the name of the new robot, `<NEW ROBOT NAME>`. Select the `Tag` for this `GameObject` as `Robot`. Then add each of the meshes for the robot as a chain starting from the base mesh. The initial empty `GameObject` should be the parent (root node) of this chain. Care should be taken when following kinematic descriptions from other sources because Unity uses a left-handed coordinate system. Many description formats, such as ROS URDFs, are described in a right-handed coordinate system. It is a good idea to save this `GameObject` chain as a prefab for easily importing it into new scenes.

Once the chain is built, create a new script and define a class, `<NEW ROBOT NAME>`, that extends the abstract class, `Robot`. This new class will configure the parameters of the robot for the `Controller` class. The following parameters should be initialized as private members:

- `const string NAME`: The robot name, `<NEW ROBOT NAME>`.
- `const int NUM_NON_EE_JOINTS`: The count of non end-effector joints.
- `const int NUM_EE_JOINTS`: The count of joints for the end-effector.
- `static string[] JOINT_NAMES`: The name of the `GameObject` for each link in the robot chain starting from the base (do not include the empty `GameObject` root).
- `static Vector3[] JOINT_AXIS`: The axis of rotation (typically static option of Vector3) for each of the joints listed in the same order as `JOINT_NAMES`.
- `double[] JOINT_VELOCITIES`: The maximum joint velocities (in degrees / second) for each of the joints listed in the same order as `JOINT_NAMES`.

The new robot class constructor should use these parameters to initialize the base `Robot` class. Any additional functions can be added as necessary, but this is the minimum implementation.

Finally, add the new robot to the `RobotTypes` enumeration using the same name as the class created for the new robot, `<NEW ROBOT NAME>`. Also, add a case for the new robot enumeration to the switch statement in `Extensions` that returns a robot of the new class, `<NEW ROBOT NAME>`. Select the base `GameObject` and add a `Script` component of type `Controller`. In the options for the `Script` component choose the newly added `RobotTypes` from the drop down list.

To add motion files for a new robot to the `Game_1DWorkspace` task, create new directories, `Resources/Motions/<NEW ROBOT NAME>/1DWorkspace/<MOTION TYPE>/<BUCKET>/`, for each of the six buckets for each motion type that motions have been synthesize for. Add the properly named motion trajectory files (`POSX_POSY.txt`) into the correct bucket folder under the correct motion type. Be sure that the ready (`ready.txt`), cylinder pickup (`pickup.txt`), and ready to place (`readyToPlace.txt`) pose files are placed in the directory `Resources/Motions/<NEW ROBOT NAME>/1DWorkspace/`.

E. Adding New Motion Types

Adding new motion types is a simple process. First, create the proper new directories, `Resources/Motions/<ROBOT NAME>/1DWorkspace/<NEW MOTION TYPE>/<BUCKET>/`, for each of the six buckets. Second, Add the properly named motion trajectory files (`POSX_POSY.txt`) into the correct bucket folder under the new motion type. Finally, add a new option in the `MotionType` enumeration. The name of the enumeration must be the same as the directory for the motion files, `<NEW MOTION TYPE>`.

F. RobotSim Limitations

Currently there are several limitations of the study framework. The most important is the lack of support for joint types other than revolute joints and lack of support for complex joints. Another current limitation is the lack of support for complex end-effectors.

None of the robots we simulate use joints other than revolute joints. As such, we have had no need to develop the capability to include other joint types, such as prismatic. It is possible to extend our framework to include other joint types. This will require adding an additional vector to the `Robot` class and subclasses that specfies the joint type for each joint. Then the functions to set the robot model's pose can be updated to change behavior (such as translating instead of rotating) based on each joints' type. Complex joints (joints with more than one axis of rotation) are also not implemented. These could be implemented by splitting up each axis of rotation into a separate joint and connecting each individual joint with zero length "virtual" links.

Anther major limitation is support for complex end-effectors. Currently the robots we use have only 2 fingers with each finger having a single rotational actuator. By comparison, a complex end-effector may have many dependent links between the actuator and the gripper. An idea to implement these advanced end-effectors may be to define them as robots since they are essentially robots themselves.

G. RobotSim Web Study Tips

When running the study framework with live participants it is especially important to limit the possibility of bad data and cheating. A few simple mechanisms can be added to a web framework to greatly improve the quality of data collected by RobotSim. These mechanisms including checking for HTML5 support, preventing mobile devices, checking the user's browser, and handling refreshes by using a trial queue.

The simplest check for HTML5 support is to use JavaScript to attempt to create a `canvas` object on the page. The WebGL build uses a `canvas` to display the game; so, failing this check is a good indication that an out-dated browser is being used or the user's browser settings will not allow game play. Additionally, checks should be performed to screen out web browsers that are not currently supported by Unity (as of this writing only Google Chrome or Mozilla FireFox are supported). Finally, mobile devices should be prevented from playing the game. Mobile devices can be detected using PHP.

Other mechanisms to ensure data quality are a trial queue and frame rate tracker. The game provides and records the user's frame rate by default. It is advisable to conduct a trial run of the game before the study (using only a few trials) and filter out participants whose average frame rate is below a reasonable threshold (we typically use 15 fps). It is also a good idea to track the remaining trials (bucket selections) in a queue on the server. By popping an element off the queue every time a motion is completed by the game (this can be detected when the game updates the recorded measurements), the current game settings can be kept server side. If the participant refreshes the page, the server can check if the trial queue is empty. If the queue is not empty the game can be reloaded with only the remaining motion trials. If a queue is not used, a refresh will result in the game starting over.

These simple mechanisms can ensure high quality data. Typically we see less than 10% bad data. The game will help detect bad data by flagging motions that no mouse movement was detected as `InvalidTrial`.

IV. Discussion

We have described the design, implementation, usage, and limitations of two systems that were developed to synthesize and evaluate robot arm motions. STrOBE is a general purpose motion synthesizer implemented using spacetime constraints. RobotSim is a game built using the Unity game engine that measures the collaborative performance between a human and a robot arm. STrOBE can be used to produce trajectories that can be evaluated using RobotSim.

STrOBE can make use of potentially many different non-linear solvers. In our implementation we utilize a sequential quadratic programming optimizer freely available from the `Python` package `scipy.optimize`. While our primary objective has been to synthesize intent-expressive motions, STrOBE can be readily extended to include a variety of constraints and objectives. RobotSim is much more specialized for evaluating a specific quality of a trajectory, intent-expressiveness. However, the modularity of RobotSim does allow adapting the robots and the controller to other environment and tasks. For example, we have packaged the RobotSim robots and controller into a Unity package and imported them into other simulation tools developed with Unity.

We plan to release STrOBE and RobotSim as open source software in the near future. We hope that releasing this software to the research community will encourage further exploration of and discussion about achieving intent-expressiveness in robot motion. We also believe using a common evaluation task/framework, such as RobotSim, will be beneficial for ensuring accurate comparisons between different methods of synthesizing intent-expressive motion.

ACKNOWLEDGMENTS

I would like to thank my advisors Dr. Michael Gleicher and Dr. Bilge Mutlu. Without their support, guidance, and feedback none of my work at UW would have been possible. I am especially grateful to my wife, Jennifer, whose love and support allowed me to begin graduate school and has kept me pressing forward. Finally, I would like to thank my mom, dad, and sister for shaping me into the person I am today.

REFERENCES

[1] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "Stomp: Stochastic trajectory optimization for motion planning," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, May 2011, pp. 4569–4574.

[2] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, May 2009, pp. 489–494.

[3] D. Bortot, M. Born, and K. Bengler, "Directly or on detours? how should industrial robots approximate humans?" in *2013 8th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, Mar. 2013, pp. 89–90.

[4] C. Lichtenthäler and A. Kirsch, "Legibility of Robot Behavior : A Literature Review," Apr. 2016, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01306977

[5] S. J. Blakemore and J. Decety, "From the perception of action to the understanding of intention," *Nature Reviews Neuroscience*, vol. 2, no. 8, pp. 561–567, Aug. 2001.

[6] F. Thomas and O. Johnston, *The Illusion of Life: Disney Animation*. New York: Disney Editions, 1981. [Online]. Available: http://opac.inria.fr/record=b1129024

[7] L. Takayama, D. Dooley, and W. Ju, "Expressing thought: Improving robot readability with animation principles," in *2011 6th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, Mar. 2011, pp. 69–76.

[8] M. Zhao, R. Shome, I. Yochelson, K. E. Bekris, and E. Kowler, "An Experimental Study for Identifying Features of Legible Manipulator Paths," in *International Symposium on Experimental Robotics*, Jun. 2014.

[9] A. Dragan and S. Srinivasa, "Generating Legible Motion," in *Robotics: Science and Systems*, Jun. 2013.

[10] A. Witkin and M. Kass, "Spacetime constraints," in *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '88. New York, NY, USA: ACM, 1988, pp. 159–168. [Online]. Available: http://doi.acm.org/10.1145/54852.378507

[11] M. Gleicher, "Retargetting motion to new characters," in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/280814.280820

[12] A. Dragan, K. Lee, and S. Srinivasa, "Legibility and predictability of robot motion," in *Human-Robot Interaction*, Mar. 2013, pp. 301–308.

[13] M. Gleicher, "Comparing constraint-based motion editing methods," *Graph. Models*, vol. 63, no. 2, pp. 107–134, Mar. 2001. [Online]. Available: http://dx.doi.org/10.1006/gmod.2001.0549