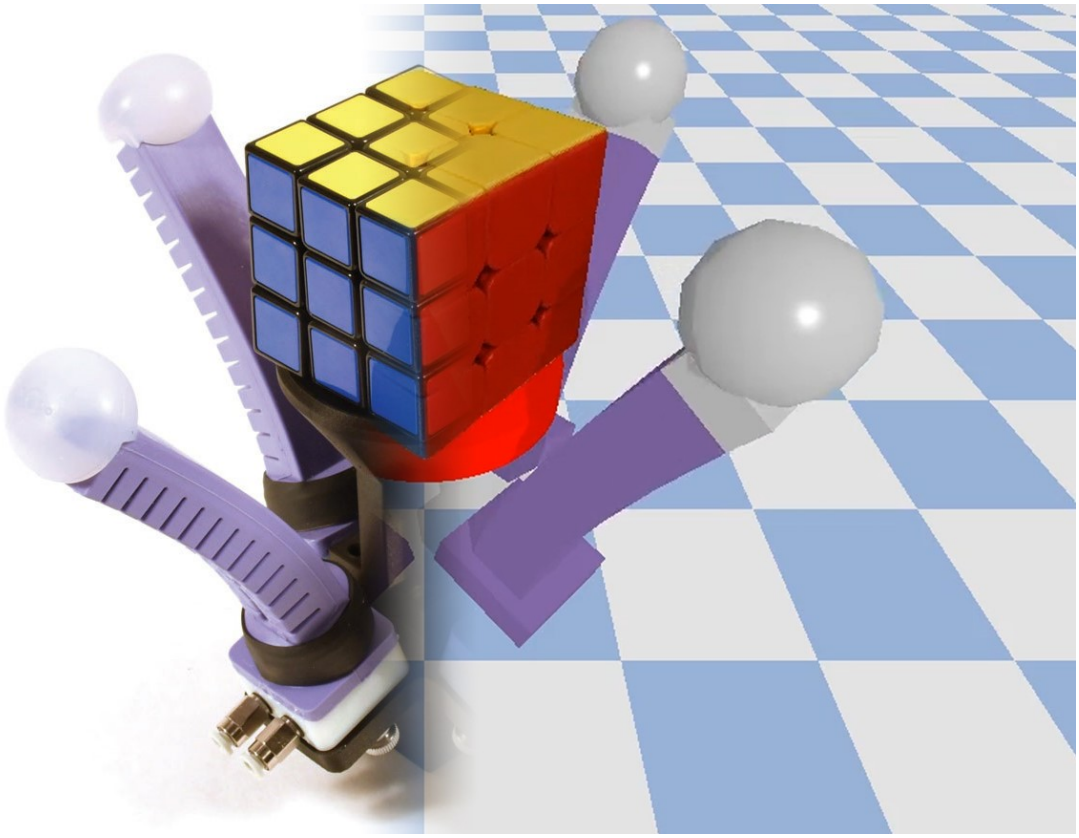
SoMo

Moritz A. Graule, Harvard Microrobotics Lab

Dec 01, 2022

CONTENTS



SoMo is a light wrapper around pybullet that facilitates the simulation of continuum manipulators.

SoMo (**So**ft**Mo**tion) is a framework to facilitate the simulation of continuum manipulator motion in PyBullet physics engine. In SoMo, continuum manipulators are approximated as a series of rigid links connected by spring-loaded joints. SoMo makes it easy to create URDFs of such approximated manipulators and load them into pybullet's rigid body simulator. With SoMo, environments with various continuum manipulators (such as hands with soft fingers or snakes) can be created and controlled with only a few lines of code.

Table of Contents

CHAPTER

ONE

GETTING STARTED

Here's how to get up and running with SoMo.

1.1 Installation

1.1.1 Requirements

- Python 3.6 +
- Tested on:
  - Ubuntu 16.04 and Ubuntu 18.04 with Python 3.6.9
  - Ubuntu 20.04 with Python 3.6.9, 3.7.9 and 3.8.2
  - Windows 10 with Python 3.7 and 3.8 through Anaconda
- Recommended: pip (`sudo apt-get install python3-pip`)
- Recommended (for Ubuntu): venv (`sudo apt-get install python3-venv`)

1.1.2 Setup

0. Make sure your system meets the requirements
1. Clone this repository
2. Set up a dedicated virtual environment using `venv`
3. Activate virtual environment
4. Install requirements from this repo: `$ pip install -r requirements.txt`
5. Install this module:
   - either by cloning this repo to your machine and using `pip install -e .` from the repo root, or with
   - `pip install git+https://github.com/graulem/somo`
6. To upgrade to the newest version: `$ pip install git+https://github.com/graulem/somo --upgrade`

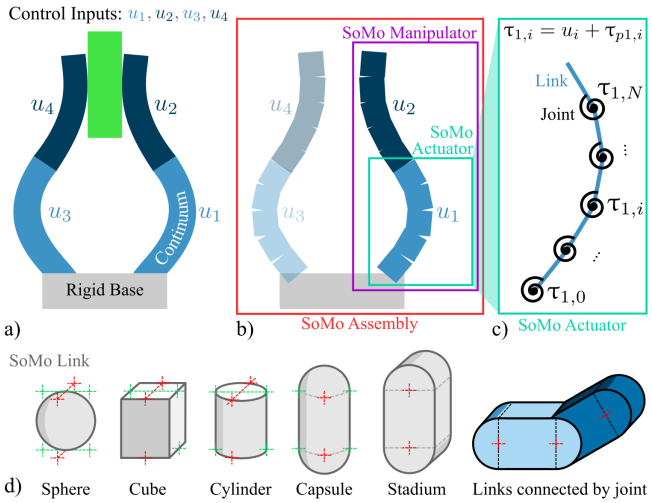
1.2 Basic Usage

This will walk you through setting up your first manipulator and complete simulation.

On this page

1.2.1 Set up a definition file

SoMo manipulators are defined as dictionaries describing one or more actuators, each of which is made up of several links and joints.



Here is an example of a typical definition of a soft finger made up of one bending actuator. You can copy this or directly download it:

`ihm_finger_def.yaml`

1.2.2 Build a SoMo Manipulator from a definition

Note: This documentation is coming soon!

Load the definition from a file directly

Note: TODO

Modify a definition before creating

SoMo definitions are just python dictionaries, so you can load them in, make modifications, then instantiate a manipulator object.

1.2.3 Control a manipulator

Note: TODO

1.3 Scaling the World

Bullet physics works best for objects larger than 0.1 simulation units. This is important for us becasue many real soft robots are on the order of 0.1m in length, leaving us no room to discretize them into smaller links. We must scale the world up to avoid numerical precision errors in bullet.

1.3.1 Bullet physics guidelines

Unfortunately the bullet physics wiki has been down for over a year now, so we must use an archived version of the page HERE. In addition, there is a small typo in that wiki that makes a big difference in how we scale inertias as discussed HERE. We use the convention agreed upon in the forum post.

1.3.2 Standard scaling laws

If we scale all lengths by **X**, we need to correct other variables:

In addition, we could scale masses by a factor **Y**, leading to more corrections:

Combining length and mass scaling, we get combined corrections:

Note: We could choose a constant density, thus setting Y=X:raw-html-m2r: `<sup>3</sup>`. However, we do not actually need to do mass scaling according to the forum post from above, so we chose to set Y=1 for simplicity.

We set the sizes of objects in our world according to these units in the various URDF and actuator definition files, and it's up to us to scale these dimensions accordingly. We also need to correct the gravitational constant when setting up our simulation. All other forces, torques, etc are calculated internally.

1.3.3 SoMo-specific scaling

Since we are discretizing the soft robots into rigid links with angular stiffness and damping terms, we need to correct these terms for the dimensional scale.

```
# Preserve Torque Scaling:
# Rotational Springs:
          T =           K * (Theta-Theta_0)
X^2 * Y * T = X^2 * Y * K * (Theta-Theta_0)

# Rotational Dampers:
          T =           B * (w-w_0)
X^2 * Y * T = X^2 * Y * B * (w-w_0)
```

Therefore, we get the following scaling laws for rotational springs and dampers:

We set these values in the joint definition file for each joint.

1.3.4 Scaling data back to real units

Since the simulations run with a certain length scaling, X, and mass scaling, Y, we need to scale the output data back to real units. Doing this is easy, just inverting all the relationships from above.

CHAPTER

TWO

EXAMPLES

Several examples of the power and versatillity of SoMo are shown here. You can find them in the examples folder in the github repo.

2.1 Basic Examples

These showcase basic functionalities.

2.1.1 Calibration to Hardware

Introduction

We are using a discretized model for soft actuators the converts continuous bending beams into rigid links and pin joints with torsional stiffnesses. To calibrate our simulated system to the real system, we need some way to relate deformation to joint angles.

Basic formulas

Deflection, δ , of cantilever beam as a function of position, x along the beam, where δ_L is the deflection at the tip, and L is the length of the beam:

Fig. 1: A simple beam of length L with a tip load F undergoes a tip deflection δ

Bending moment, M_b of a cantilever beam with a tip load, F_{tip}, as a function of position, x along the beam:

Linear bending stiffness, K_b of a cantilever beam at the tip:

Calibrating Joint Stiffnesses

For an actuator split evenly into N segments, we make a few assumptions:

1. Bending stiffness represents the linear bending stiffness at the tip of the actuator at its unloaded state
2. Assume small angles (for calibration purposes), so $sin(\theta) = \theta$
3. We want to match the deflection of each segment with an appropriate spring force given the bending moment in the actuator.

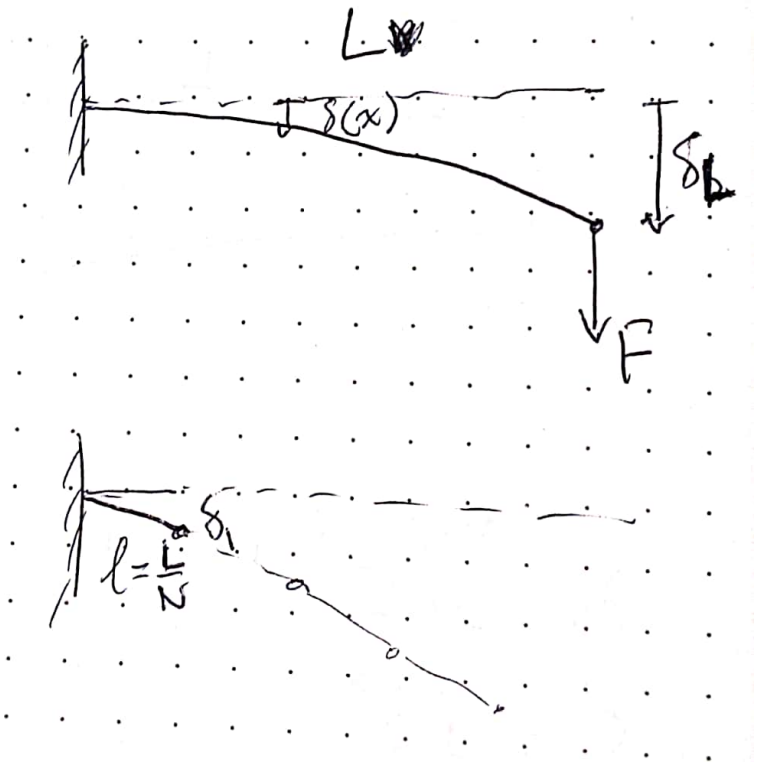


Fig. 2: The simple beam is discretized into N equal-length segments connected by pin joints.

We find the formula for the deflection at the tip of the first segment during a load at the actuator's tip to be:

For the first segment with a length, $\frac{L}{N}$, deflection δ_1, we can define a rotational spring constant, κ_1, that achieves an angle, θ_1 when a torque, τ_1, is applied on the joint.

Thus, we obtain an equation for the spring constant:

We can find what θ_1 needs to be using the formula:

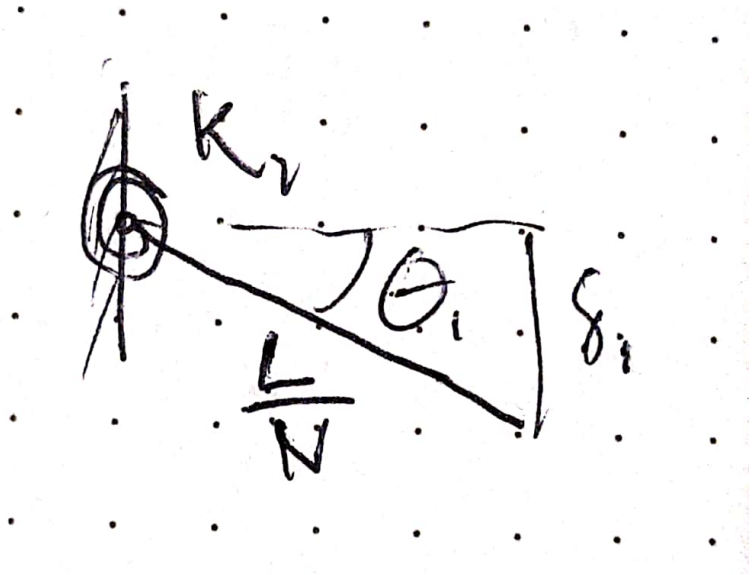


Fig. 3: A simple pin joint with rotational spring

and using the small angle assumption, we obtain $\sin(\theta) = \theta$, so:

Now, τ_1 is assumed to be the bending moment withheld by this first link (the moment at $x = 0$). :

Thus, we can move back to the joint stiffness:

Determining appropriate actuator torques

We need to determine the actuation moment applied to actuators. Many of our physical systems are air-driven, so we can use blocked-force measurements at various pressures to get a rough estimate of actuation torques at pressures of interest.

During a blocked force measurement, we assume all force produced is balancing the internal actuation moment. Given this assumption, we get:

Example:

From our paper on in-hand manipulation with soft fingers [Abondance *et al.*, 2020], we measured the linear bending stiffnesses of our 0.1 m long fingers in the grasping and side axes:

In our typical SoMo simulation of these fingers, we use 5 joints in each direction, so N = 5 and L = 0.1 m. Putting this through the formula for joint stiffnesses, we get:

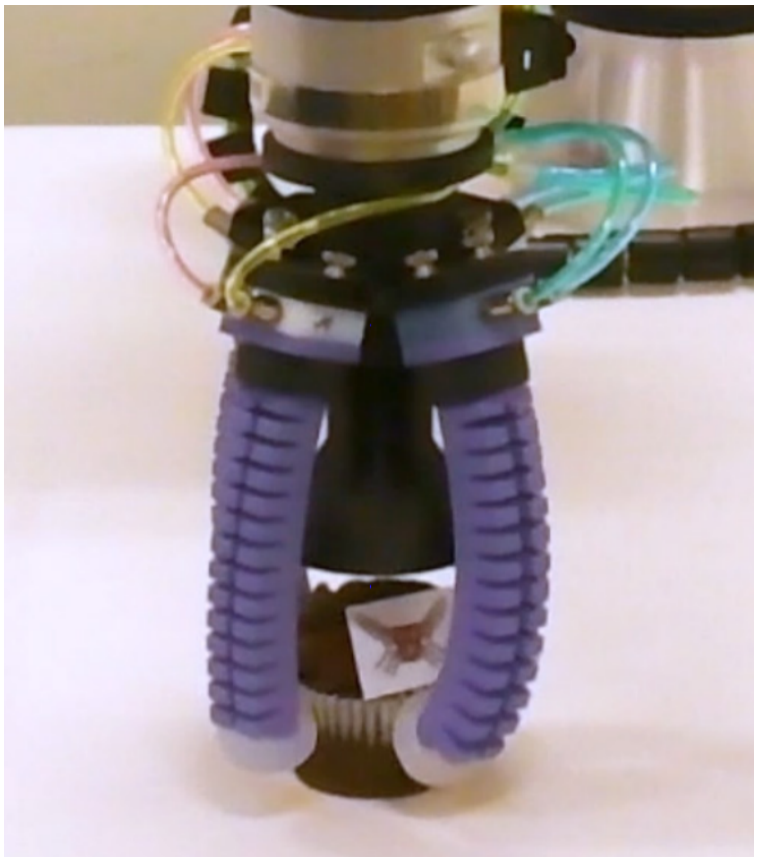


Fig. 4: Real soft robotic hand platform from [Abondance *et al.*, 2020], where the mechanical properties of the fingers have been characterized.

The last step is to scale the joint stiffnesses by the square of the world scale per the world scaling discussion. In many of our examples we use a world scale of 20, so scaling the stiffnesses by 400 results in:

To apply realistic actuation torques to the system, we calibrate the grasping axis on the 100 kPa value, which produced 0.75 N of force over the 0.1m length finger body.

Then we transform torques by the square of the world scale, so if we want to apply 100 kPa to the real life fingers, the the simulated fingers need:

For the side-axis, we explicitly control the actuation torques in simulation, but the real system uses a pressure differential. Based on a differential of 100kPa to achieve reasonable side-to-side motion in the hardware, we estimate approximately 3 times the value for the grasping axis based on observations, resulting in:

All these values produce physically accurate simulations that seem to work well!

References

2.1.2 Blocked Force Testing

Note: TODO

2.1.3 SoMo Assemblies

Note: TODO

2.2 Design Studies

SoMo can be used for a variety of design studies where building physical hardware would be prohibitively time-consuming.

2.3 Whole-Arm Manipulation

SoMo can be used for manipulation studies.

2.3.1 Playing Basketball

Note: Update with details.

2.4 Locomotion

SoMo can be used for locomotion studies

2.4.1 Snake Locomotion

Note: Update with details.

CHAPTER

THREE

CLASS REFERENCE

Each page contains details and full API reference for all the classes in the SoMo Framework.

For an explanation of how to use all of it together, see *Basic Usage*.

3.1 SoMo Manipulators

SoMo provides an easy way to generate continuum manipulators. **Manipulators** can be comprised of several serially-chained **actuators** (`SMActuatorDefinition`), made up of a series of **links** (`SMLinkDefinition`) connected by spring-loaded **joints** (`SMJointDefinition`),

To actually implement a manipulator, you can define it in a dictionary or yaml/json file, and load the definition as a `SMManipulatorDefinition` object. The lower-level definitions are taken care of internally.

class somo.sm_link_definition.**SMLinkDefinition**(*shape_type: str*, *dimensions: [Union[float, int]]*, *mass: Union[float, int]*, *inertial_values: [Union[float, int]]*, *material_color: [Union[float, int]]*, *material_name: str*, *origin_offset: [Union[float, int]] = None*, *visual_geometry_scaling_factor=1.0*)

SMLinkDescription is correct upon instantiation.

Example json representation: link_example.json # todo update example presentation {

shape_type: xx finish, dimensions: xx finish, mass: xx, inertial_values: xx, material_color: xx, material_name: ,

}

static assert_required_fields(*dict_definition: dict*)

static from_file(*file_path: str*) → *somo.sm_link_definition.SMLinkDefinition*

static from_json(*json_file_path: str*) → *somo.sm_link_definition.SMLinkDefinition*

reduce_height(*height_scaling_factor*)

to_json()

class somo.sm_joint_definition.**SMJointDefinition**(*joint_type: str*, *axis: Union[None, List] = None*, *limits: Optional[List] = None*, *spring_stiffness: Optional[Union[float, int]] = None*, *joint_neutral_position: Optional[Union[float, int]] = None*, *neutral_axis_offset: [Union[float, int, NoneType]] = None*, *joint_control_limit_force: [Union[float, int, NoneType]] = None*)

SMJointDefinition is correct upon instantiation.

Example json representation: link_example.json # todo fix {

xx

}

static assert_required_fields(*dict_definition: dict*)

static from_file(*file_path: str*) → *somo.sm_joint_definition.SMJointDefinition*

static from_json(*json_file_path: str*) → *somo.sm_joint_definition.SMJointDefinition*

to_json()

class somo.sm_actuator_definition.**SMActuatorDefinition**(*actuator_length: Union[float, int]*, *n_segments: int*, *link_definition: Union[*somo.sm_link_definition.SMLinkDefinition*, Dict, str]*, *joint_definitions: [Union[*somo.sm_joint_definition.SMJointDefinition*, Dict, str]]*, *planar_flag: Union[bool, int]*)

static assert_required_fields(*dict_definition: dict*)

static from_file(*file_path: str*) → *somo.sm_actuator_definition.SMActuatorDefinition*

static from_json(*json_file_path: str*) → *somo.sm_actuator_definition.SMActuatorDefinition*

to_json()

class somo.sm_manipulator_definition.**SMManipulatorDefinition**(*n_act: Union[float, int]*, *base_definition: Optional[*somo.sm_link_definitio*actuator_definitions: [Union[*somo.sm_actuator_definition.SMActuato*Dict, str]]*, *manipulator_name: str*, *tip_definition: Optional[*somo.sm_link_definitio*= None*, *urdf_filename: Optional[str] = None*, *tip_definitions: Union[None, List] = None*)

static assert_required_fields(*dict_definition: dict*)

static from_file(*file_path: str*) → *somo.sm_manipulator_definition.SMManipulatorDefinition*

static from_json(*json_file_path: str*) → *somo.sm_manipulator_definition.SMManipulatorDefinition*

to_json()

3.2 SoMo Assemblies

You can connect several **manipulators** object into **assemblies** such as a hand with several fingers, or a body with legs.

Note: Documentation for this is coming soon!

3.3 Generating Parameter Sweeps

Once you have your environment set up, you can easily run parameter sweeps using the built-in sweep framework.

Warning: This is a work in progress. Some parts of this module are not very elegant, but we are working on this!

3.3.1 Base Functionality

class somo.sweep.**BatchSimulation**

load_run_list(*todo_filename='runs_todo.yaml'*, *recalculate=False*)

run(*run_function*, *parallel=True*, *num_processes=None*)

class somo.sweep.**DataLabeler**(*label_functions*)

process_all(*config_file*, *label_function=None*, ***kwargs*)

Process all datasets within a config file

set_global_scale(*scale*)

class somo.sweep.**RunGenerator**

from_file(*config_file*, *todo_filename='runs_todo.yaml'*)

Generate a set of runs from a config file

generate_params(*config*)

Generate all permutations of a given set of sweep parameters

make_2d_slices(*config*)

Make a simple set of runs using all permutations of sweep parameters

make_simple(*config*, *save_todo=True*)

Make a simple set of runs using all permutations of sweep parameters

3.3.2 Experimental Classes

Warning: These classes enable experimental functionality. Use at your own risk.

class somo.sweep.**ContourPlotter**(*config_file*)

make_plots(*labels=None*, *show=False*, *recalculate=False*, *num_bins=12*)

plot_one(*success_filename*, *labels=None*, *show=False*, *replace=False*, *aux_savepath=None*)

Make a plot of the grasp type/success rate of 2D sweep data.

set_axes_equal(*in_set*)

set_colors(*status_colors=None*)

set_status_colors(*label_set=None*, *color_set=None*, *color_labels=None*)

set_status_colors_dict(*label_list*)

set_status_colors_label(*label_name='default'*, *color_set=None*)

class somo.sweep.**GridPlotter**(*config_file*)

make_plots(*labels=None*, *show=False*, *recalculate=False*)

plot_one(*success_filename*, *labels=None*, *show=False*, *replace=False*, *aux_savepath=None*)

Make a plot of the grasp type/success rate of 2D sweep data.

set_axes_equal(*in_set*)

set_colors(*status_colors=None*, *fingertip=None*)

set_fingertip(*state*)

set_status_colors(*label_set=None*, *color_set=None*, *color_labels=None*)

set_status_colors_dict(*label_list*)

set_status_colors_label(*label_name='default'*, *color_set=None*)

CHAPTER

FOUR

CONTRIBUTING

Contributing Checklist

- only through a new branch and reviewed PR (no pushes to master!)
- always use Black Code Formatter for code formatting
- always bump the version of your branch by increasing the version number listed in somo/_version.py

CHAPTER

FIVE

QUICK INSTALL

Check out the *Installation Instructions*

Note: Coming soon: pip install!

CHAPTER

SIX

EXPLORE THE EXAMPLES

Check out the *Examples*, or run any of the files in the examples folder. “examples/basic” is a great place to start!

CHAPTER

SEVEN

LINKS

- **Documentation:** Read the Docs
- **pip install:** View on PyPi (*Not Launched Yet*)
- **Source code:** Github

CHAPTER

EIGHT

CONTACT

If you have questions, or if you've done something interesting with this package, get in touch with Moritz Graule!

If you find a problem or want something added to the library, open an issue on Github.

CHAPTER

NINE

CITATION

When citing SoMo, use this citation:

```
@inproceedings{graule2020somo,
   title={SoMo: Fast and Accurate Simulations of Continuum Robots in Complex␣
↪Environments},
   author={Graule, Moritz A. and Teeple, Clark B and McCarthy, Thomas P and St. Louis,␣
↪Randall C and Kim, Grace R and Wood, Robert J},
   booktitle={2021 IEEE International Conference on Intelligent Robots and Systems␣
↪(IROS)},
   pages={In Review},
   year={2021},
   organization={IEEE}
}
```

CHAPTER

TEN

CITED IN...

SoMo has enabled other work:

- C.B. Teeple, R.C. St. Louis, M.A Graule, and R.J. Wood, **Digit Arrangement for Soft Robotic Hands: Enhancing Dexterous In-Hand Manipulation**, In Review, IROS 2021
- C.B. Teeple, G.R. Kim, M.A Graule, and R.J. Wood, **An Active Palm Enhances Dexterity for Soft Robotic In-Hand Manipulation**, ICRA 2021

CHAPTER

ELEVEN

SOMO IN ACTION

BIBLIOGRAPHY

[abondance2020dexterous] Sylvain Abondance, Clark B Teeple, and Robert J Wood. A dexterous soft robotic hand for delicate in-hand manipulation. *IEEE Robotics and Automation Letters*, 5(4):5502–5509, 2020.

PYTHON MODULE INDEX

S

INDEX