sorotraj

Clark B. Teeple, Harvard Microrobotics Lab

Feb 18, 2022

DOCUMENTATION

1 Quickstart Guide **3**
1.1 sorotraj . . . 3

2 Examples **7**
2.1 Build One Trajectory . . . 7
2.2 Build and Save Trajectories . . . 12
2.3 Convert Trajectories . . . 12
2.4 Advanced Examples . . . 13

3 API Reference **15**
3.1 Trajectory Builder . . . 15
3.2 Trajectory Interpolation . . . 18

4 Contributing **23**

5 Index **25**

6 Quick Install **27**

7 Explore the Examples **29**

8 Links **31**

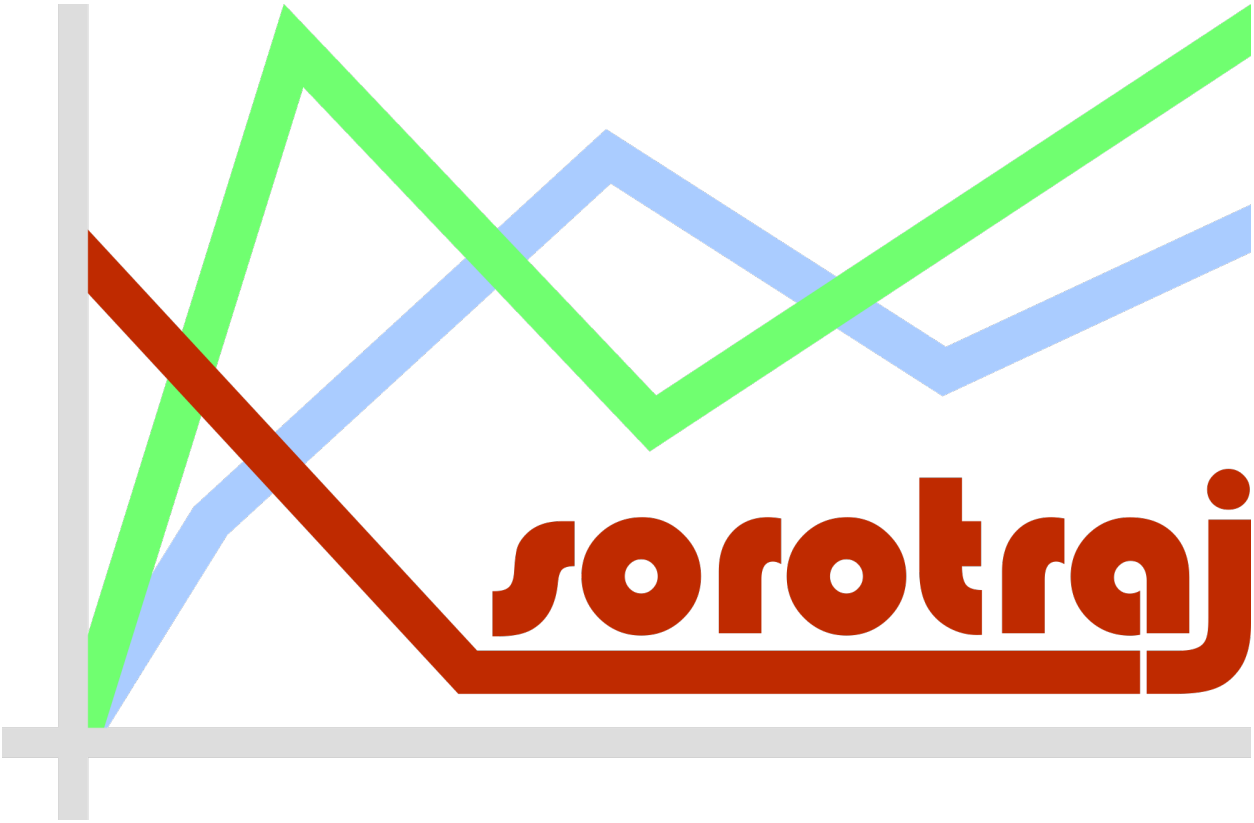
9 Contact **33**

10 Used In… **35**
10.1 References . . . 35

Bibliography **37**

Python Module Index **39**

Index **41**



Sorotraj allows you to generate trajectory functions to control soft robots. Trajectories are defined in a simple, human-readable yaml file, and are designed for compatibillity with both Ctrl-P pressure controllers (for real-world soft robots) and SoMo simulations (for virtual soft robots).

Table of Contents

CHAPTER

ONE

QUICKSTART GUIDE

(from "README.md" in github repo)

1.1 sorotraj

Generate trajectories for soft robots from yaml files (accompanies the Ctrl-P project and SoMo simulation framework)

1.1.1 Installation

Install the release version

This package is on pypi, so anyone can install it with pip: `pip install sorotraj`

Install the most-recent development version

1. Clone the package from the github repo
2. Navigate into the main folder
3. `pip install .`

1.1.2 Usage

Minimal Example

```
import sorotraj

file_to_use = 'traj_setup/setpoint_traj_demo.yaml'

traj = sorotraj.TrajBuilder()
traj.load_traj_def(file_to_use)
trajectory = traj.get_trajectory()
interp = sorotraj.Interpolator(trajectory)
actuation_fn = interp.get_interp_function(
                num_reps=1,
                speed_factor=2.0,
                invert_direction=False)
print(actuation_fn(2.155))
```

Check out the *examples* folder for more detailed usage examples

1.1.3 Set Up Trajectories:

Trajectories are made of three parts:

1. **main**: used in a looping trajectory
2. **prefix**: happens once before the main part
3. **suffix**: happens once after the main part

Here's an example of what that might look like defined in a yaml file:

```yaml
config:
    setpoints:
        # [time, finger1, finger2, n/c, n/c]
        main:
            - [0.0,   10, 12, 14,  16]
            - [1.0,    20, 0, 0,  0]
            - [2.0,   0, 20, 0,  0]
            - [3.0,     0, 0, 20, 0]
            - [4.0,     0, 0, 0, 20]
            - [5.0,    10, 12, 14, 16]

        prefix:
            - [0.000,   0, 0, 0,  0]
            - [1.0,    10, 12, 14,  16]

        suffix:
            - [2.000,   10, 12, 14,  16]
            - [3.0,  0, 0, 0,  0]
```

There are currently three types of ways to generate the **main** part of a trajectory:

1. **direct**: You enter waypoints directly
   - Define waypoints as a list of lists of the form: `[time in sec], [a_1], [a_2], ..., [a_n]`
2. **interp**: Interpolate between waypoints
   - Define waypoints as a list of lists of the form: `[time in sec], [a_1], [a_2], ..., [a_n]`
   - Set a few more parameters:
     - **interp_type**: (`string`) The type of interpolation to use. right now types include: `'linear'`, `'cubic'`, and `'none'`
     - **subsample_num**: (`int`) The total number of subsamples over the whole trajectory
3. **waveform**: Generate waveforms (very basic, still just in-phase waveforms across all channels)
   - Set up the waveform:
     - **waveform_type**: (`string`) Types include: square-sampled, square, sin, cos-up, cos-down, triangle, sawtooth-f, and sawtooth-r
     - **waveform_freq**: (`float`) Frequency in Hertz
     - **waveform_max**: (`float`) A list of the maximum values for the waveform, in the form: `[20, 0, 15, 5]`

  - **waveform_min**: (`float`) A list of the minimum values for the waveform, in the form: `[0, 20, 0, 15]`
- Set a few more parameters:
  - **subsample_num**: (`int`) The total number of subsamples over the whole trajectory
  - **num_cycles**: (`int`) The number of cycles of the waveform
  - **channels**: (`bool`) Flags to turn channels on and off. A list of the form: `[1,1,0,0]`

1.1.4 Convert Trajectories Line-by-Line

Check out the _build_convert*trajectories.py* example.

1. Set up a conversion function
   - Inputs: one original trajectory line (`list`)
   - Outputs: one new trajectory line (`list`)
2. Load the trajectory like normal
   - `traj.load_traj_def(file_to_use)`
3. Convert the trajectory by passing the conversion function
   - `traj.convert_traj(conversion_function)`
4. This conversion overwrites the original trajectory. Now you can save it like normal
   - `traj.save_traj(file_to_save)`
5. Convert the trajectory definition by passing the conversion function
   - `traj.convert_definition(conversion_function)`
6. This conversion overwrites the original trajectory definition and reguilds the trajectory. Now you can save the definition like normal
   - `traj.save_definition(file_to_save)`

1.1.5 Build an interpolator

```
interp = sorotraj.Interpolator(trajectory)
```

- **trajectory**: A trajectory object generated by `sorotraj.TrajBuilder`

```
actuation_fn, final_time = interp.get_traj_function(
                num_reps=1,
                speed_factor=1.0,
                invert_direction=False)
```

- **num_reps**: (`int`, default=1) Number of times to repeat the main looping trajectory
  - Must be positive, nonzero
- **speed_factor**: (`float`, default=1.0) A speed multiplier that is applied to the main loop (but not the prefix or suffix)
  - Must be positive, nonzero

- **invert_direction**: (`bool`, default=False) Negate the whole trajectory (useful if actuators have different directionalities)
  - (`bool`): Negate all channels
  - (`list of ints`): Choose which channels to negate with a list of channel indices

```
cycle_fn = interp.get_cycle_function(
                num_reps=1,
                speed_factor=1.0,
                invert_direction=False)
```

- Same inputs as `get_interp_function()`, but returns a cycle function (returns the current cycle as a function of time)
- cycle_fn takes these values:
  - -2 = Prefix
  - -1 = Suffix
  - 0-N = Main loop index

CHAPTER

TWO

EXAMPLES

Several examples of how sorotraj is used in the Ctrl-P system as well as SoMo simulations are shown here. You can find them in the examples folder in the github repo.

2.1 Build One Trajectory

In this example, we will build one trajectory from a definition file, create an interpolator, and use that interpolator to generate a graph.

2.1.1 Trajectory Definition

Trajectories are defined in YAML files as either a series of waypoints, or simple periodic functions. Here we will look at a typical waypoint trajectory:

setpoint_traj_demo.yaml

```yaml
# A simple waypoint trajectory with prefix and suffix

# Top-level settings
settings:
    traj_type: 'direct'  # Types include: 'waveform', 'interp', 'direct'


# Place all of the type-specific settings in here
config:
    setpoints:
        # [time, finger1, finger2, n/c, n/c]
        main:
            - [0.0,   10, 12, 14,  16]
            - [1.0,    20, 0, 0,  0]
            - [2.0,   0, 20, 0,  0]
            - [3.0,     0, 0, 20, 0]
            - [4.0,     0, 0, 0, 20]
            - [5.0,    10, 12, 14, 16]

        prefix:
            - [0.000,   0, 0, 0,  0]
            - [1.0,    10, 12, 14,  16]
```

(continues on next page)

(continued from previous page)

```
        suffix:
            - [2.000,   10, 12, 14,  16]
            - [3.0,  0, 0, 0,  0]
```

All trajectories can have three component:

1. `prefix` - Runs once at the beginning of the trajectory
2. `main` - Can be looped over and over (ensure the first and last waypoint are equal)
3. `suffix` - Runs once at the end of the trajectory

Within each trajectory component, each waypoint is defined as a list:

- **arg #0** - Time (in seconds)
- **args #1-N** - Waypoint values for each signal

2.1.2 Trajectory Builder

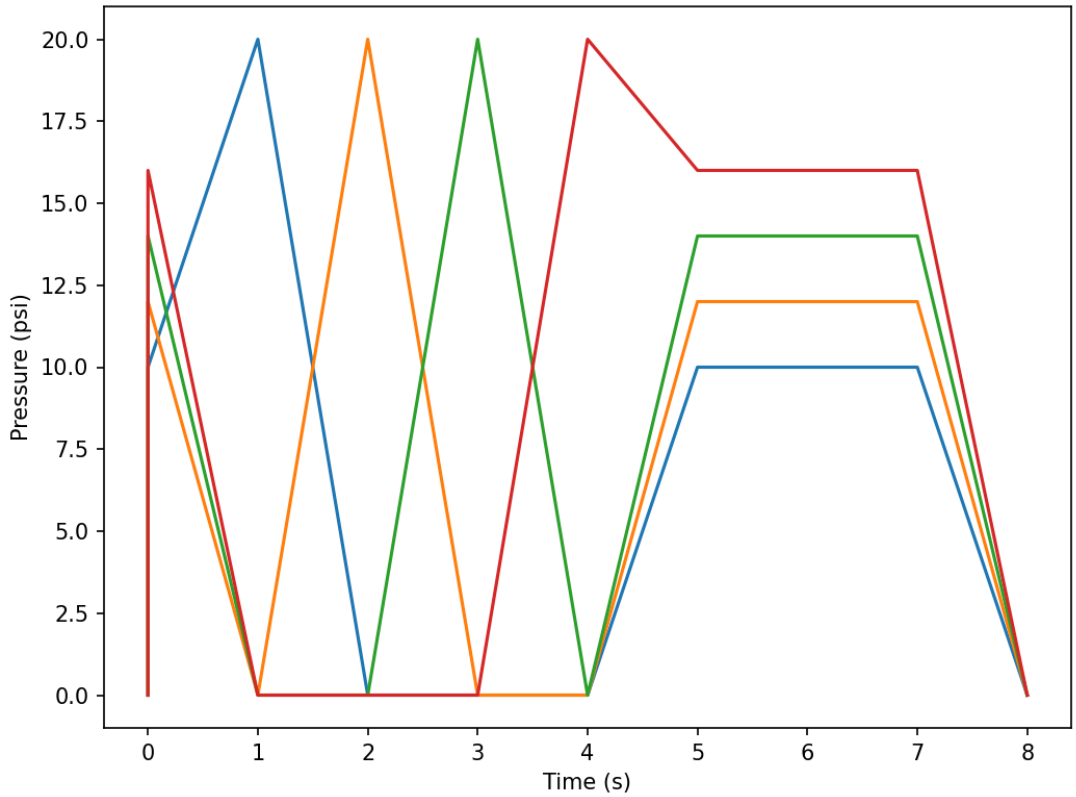
We need to build this trajectory before we can use it. To do this, we use a `TrajBuilder` object, and give it a file to load the definition from.

```
file_to_use = 'traj_setup/setpoint_traj_demo.yaml'
builder = sorotraj.TrajBuilder() # Make a traj-builder object
builder.load_traj_def(file_to_use) # Load the file
traj = builder.get_trajectory()
```

Note: We can also directly load a trajectory definition py passing a dictionary to the `TrajBuilder`: `builder.set_definition(def_dict)`

Now that we have the trajectory (`traj`), let's plot it:

```
builder.plot_traj()
```



2.1.3 Trajectory Interpolator

Now we want to create an interpolator so we can have a convenient way to get trajectory values given arbitrary values of time. We do this with an `Interpolator` object. Here, we want to get an interpolator function that loops the `main` trajectory component 2 times, with a speed factor of 1.0 (times as defined in the definition), and invert signals 1 and 3 (these are indices).

```
interp = sorotraj.Interpolator(traj)

# Get the actuation function for the specified run parameters
actuation_fn, final_time = interp.get_traj_function(
                num_reps=2,
                speed_factor=1.0,
                invert_direction=[1,3])
```

Now that we have an interpolation function we can input a time (or vector of times) and obtain the trajectory at arbirary times. This function is useful when performing simulations of soft systems.

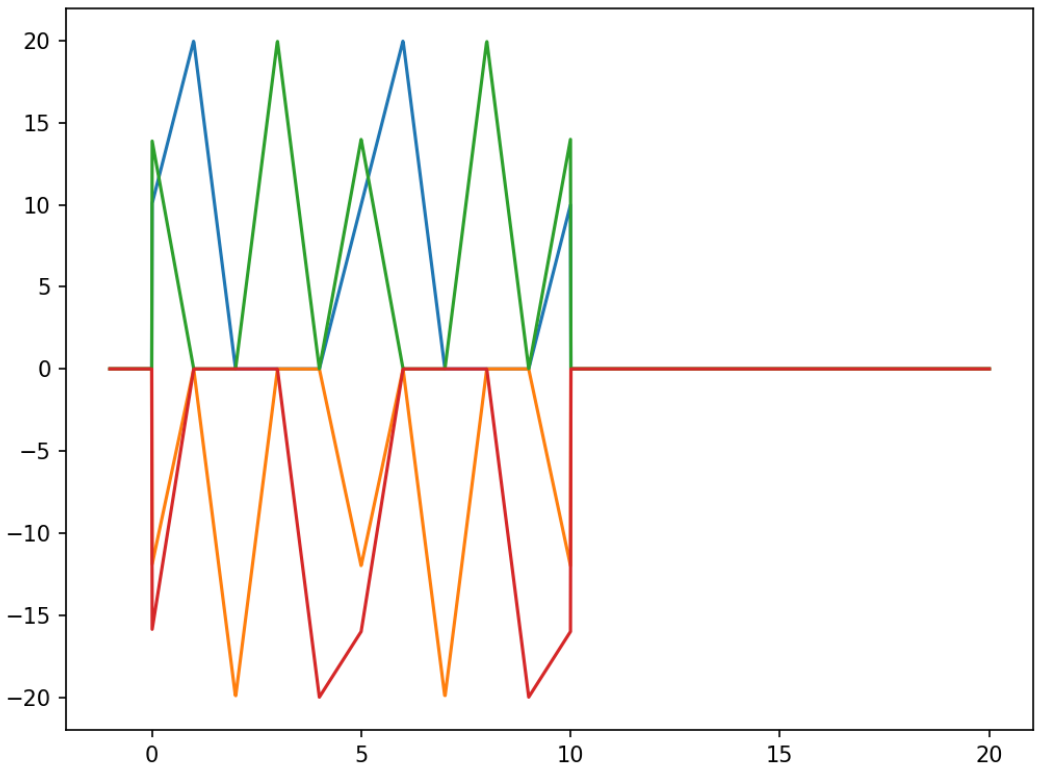
In this example, we go from -1 (before the start of the trajectory) to 20 sec. (well beyond the end of the trajectory).

```
times = np.linspace(-1,20,2000)
vals = actuation_fn(times)
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(8,6),dpi=150)
plt.plot(times, vals)
plt.show()
```



2.1.4 Full Code

Here's the whole example in one file:

build_one_trajectory.py

```
import sorotraj
import numpy as np
import matplotlib.pyplot as plt

#file_to_use = 'traj_setup/setpoint_traj_demo.yaml'      # Basic demo
#file_to_use = 'traj_setup/setpoint_traj_demo_err0.yaml' # duplicate time (will throw␣
↪exception)
#file_to_use = 'traj_setup/setpoint_traj_demo_err1.yaml' # non-monotonic time (will throw␣
↪exception)
#file_to_use = 'traj_setup/setpoint_traj_demo_0.yaml'    # empty prefix
file_to_use = 'traj_setup/setpoint_traj_demo_1.yaml'     # single line prefix
```

(continues on next page)

(continued from previous page)

```
#file_to_use = 'traj_setup/waveform_traj_demo.yaml'    # single prefix line

# Build the trajectory from the definition file
builder = sorotraj.TrajBuilder()
builder.load_traj_def(file_to_use)
traj = builder.get_trajectory()
for key in traj:
        print(key)
        print(traj[key])

# Plot the trajectory
builder.plot_traj(fig_kwargs={'figsize':(8,6),'dpi':150})

# Make an interpolator from the trajectory
interp = sorotraj.Interpolator(traj)

# Get the actuation function for the specified run parameters
actuation_fn, final_time = interp.get_traj_function(
                num_reps=2,
                speed_factor=1.0,
                invert_direction=[1,3])

print("Final Interpolation Time: %f"%(final_time))

# Get the cycle function for the specified run parameters
cycle_fn = interp.get_cycle_function(
                num_reps=2,
                speed_factor=1.0,
                invert_direction=[1,3])

# Plot the actuation function vs. time
times = np.linspace(-1,20,2000)
vals = actuation_fn(times)

plt.figure(figsize=(8,6),dpi=150)
plt.plot(times, vals)
plt.show()

abs_times = builder.get_absolute_times(num_reps=2, speed_factor=1.0)
abs_vals  = actuation_fn(abs_times)

print("Times")
print(abs_times)

traj_flat = builder.get_flattened_trajectory(num_reps=2, speed_factor=1.0, invert_
↪direction=[1,3])

print(traj_flat)
```

2.2 Build and Save Trajectories

build_save_trajectories.py

```
import sorotraj
import os

setup_location = 'traj_setup'
build_location = 'traj_built'

files_to_use = ['waveform_traj_demo','interp_setpoint','setpoint_traj_demo']

# Build a trajectory builder
traj = sorotraj.TrajBuilder()
for file in files_to_use:
        # Load, build, and save each trajectory
        traj.load_traj_def(os.path.join(setup_location,file))
        traj.save_traj(os.path.join(build_location,file))
```

2.3 Convert Trajectories

build_convert_trajectories.py

```
import sorotraj
import os

setup_location = 'traj_setup'
build_location = 'traj_built'

files_to_use = ['waveform_traj_demo','interp_setpoint','setpoint_traj_demo']

# Define a line-by-line conversion function to use
#   This example converts from orthogonal axes to differential actuation.
def linear_conversion(traj_line, weights):
    traj_length=len(traj_line)-1

    traj_line_new = [0]*(traj_length+1)
    traj_line_new[0]=traj_line[0] # Use the same time point

    for idx in range(int(traj_length/2)):
        idx_list = [2*idx+1, 2*idx+2]
        traj_line_new[idx_list[0]] = weights[0]*traj_line[idx_list[0]] + weights[1]*traj_
↪line[idx_list[1]]
        traj_line_new[idx_list[1]] = weights[0]*traj_line[idx_list[0]] - weights[1]*traj_
↪line[idx_list[1]]

    return traj_line_new


# Set up the specific version of the conversion function to use
weights = [1.0, 0.5]
```

(continues on next page)

(continued from previous page)

```
conversion_fun = lambda line: linear_conversion(line, weights)

# Test the conversion
traj_line_test = [0.00,  5,15,  5,15  ,-10,0,  -10,0]
print(traj_line_test)
print(conversion_fun(traj_line_test))

# Build the trajectories, convert them , and save them
traj = sorotraj.TrajBuilder()
for file in files_to_use:
    traj.load_traj_def(os.path.join(setup_location,file))
    traj.convert_traj(conversion_fun)
    traj.save_traj(os.path.join(build_location,file+'_convert'))

# Convert the definitions if possible
traj = sorotraj.TrajBuilder()
for file in files_to_use:
    traj.load_traj_def(os.path.join(setup_location,file))
    traj.convert_definition(conversion_fun)
    traj.save_definition(os.path.join(setup_location,file+'_convert'))
```

2.4 Advanced Examples

Getting a bit into the weeds, here are some examples that showcase some advanced functionality

2.4.1 Wrapped Interpolator

wrapped_interp.py

```
import sorotraj
import numpy as np
import matplotlib.pyplot as plt

file_to_use = 'traj_setup/setpoint_traj_demo.yaml'

# Build the trajectory from the definition file
builder = sorotraj.TrajBuilder()
builder.load_traj_def(file_to_use)
traj = builder.get_traj_components()

# Make a wrapped interpolator with the looping part
# of the trajectory
interp = sorotraj.interpolator.WrappedInterp1d(
                        traj['setpoints']['time'],
                        traj['setpoints']['values'],
                        axis=0)

interp_fun = interp.get_function()
```

(continues on next page)

(continued from previous page)

```
# Plot the values over a ridiculous range of times
times = np.linspace(-10,20,2000)
vals = interp_fun(times)

plt.plot(times, vals)
plt.show()
```

CHAPTER

THREE

API REFERENCE

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

For an explanation of how to use all of it together, see *Quickstart Guide*.

3.1 Trajectory Builder

Build trajectories from a definition (either from a yaml file or directly via python dictionary).

class sorotraj.build_traj.**TrajBuilder**(*verbose=False*, ***kwargs*)

Trajectory builder

verbose

Flag used to turn on verbose printing

Type bool

Examples

```
>>> def_file = 'examples/traj_setup/setpoint_traj_demo.yaml'
... builder = TrajBuilder()
... builder.load_traj_def(def_file)
... traj = builder.get_trajectory()
... out_file = 'examples/traj_built/setpoint_traj_demo.traj'
... builder.save_traj(out_file)
```

build_traj()

Build the current trajectory

Raises **RuntimeError** – If the trajectory definition has not been set

convert_definition(*conversion_fun*)

Convert a trajectory definition line-by-line using a conversion function.

Trajectory definition of type 'direct' and 'interp' can be converted, but waveform trajectory definitions cannot.

Parameters **conversion_fun** (*function*) – Conversion function taking in one waypoint (list) and returning waypoint (list)

Raises

- **RuntimeError** – If the trajectory definition is not set
- **RuntimeError** – If the trajectory type is incompatible (not direct or interp)

convert_traj(*conversion_fun*)

Convert a trajectory line-by-line using a conversion function

Parameters **conversion_fun** (*function*) – Conversion function taking in one trajectory line (list) and returning one line (list)

Raises **RuntimeError** – If the trajectory has not been built

get_absolute_times(*num_reps=1*, *speed_factor=1.0*, *merge_duplicates=False*, *offset_tol=0.001*)

Get a list of relevant time points starting from t=0 at the beginning of the prefix

Parameters

- **num_reps** (*int*) – Number of times to repeat the "main" trajectory segment
- **speed_factor** (*float*) – Speed multiplier (times are multiplied by inverse of this)
- **merge_duplicates** (*bool*) – Merge duplicate times (rather than applying a tiny offset)
- **offset_tol** (*float*) – The tolerance to use when fixing duplicate points. *This should be set to something very small relative to the timescales you are working with.*

Returns **times** – A list of key time points in the trajectory

Return type list

Raises

- **RuntimeError** – If the trajectory has not been built
- **ValueError** – If num_reps is less than 0, or if speed_factor is 0 or less

get_definition(*use_copy=False*)

Get the trajectory definition.

Parameters **use_copy** (*bool*) – Decide whether to pass the trajectory by referece. If True, the actual trajectory object is returned, otherwise a copy of the trajectory is returned.

Returns **trajectory_definition** – The trajectory definition

Return type dict

Raises **RuntimeError** – If the trajectory definition is not set set

get_flattened_trajectory(*num_reps=1*, *speed_factor=1.0*, *invert_direction=False*, ***kwargs*)

Get a flattened trajectory (simple list of waypoints) from start to finish including prefix, main, and suffix segments.

Parameters

- **num_reps** (*int*) – Number of times to repeat the "main" trajectory segment
- **speed_factor** (*float*) – Speed multiplier (times are multiplied by inverse of this)
- **invert_direction** (*Union[bool, list]*) – Invert the sign of the interpolated values. If True, all signs are flipped. If list, invert_direction is treated as a list of indices.
- ****kwargs** (*(Multiple)*) – Arguments to pass to the `get_absolute_times()` function

Returns **trajectory** – A simple list of waypoints for the trajectory from start to finish. *Each point is represented in the form * `[time, s0, s1, ..., sn]`, where s0-n are signals.

Return type list of lists

get_traj_components()

Get trajectory split into compoenents rather than in vector form

This generates a dictionary with the same trajectory components as a usual trajectory, but the values of each component are dictionaries with 'time' and 'values' rather than the usual list of lists.

Raises `RuntimeError` – If the trajectory has not been built

`get_trajectory`(*use_copy=False*)

Get the built trajectory.

Parameters `use_copy` (*bool*) – Decide whether to pass the trajectory by referece. If True, the actual trajectory object is returned, otherwise a copy of the trajectory is returned.

Returns **trajectory** – The full trajectory

Return type dict

Raises `RuntimeError` – If the trajectory has not been built

`load_traj_def`(*filename*)

Load a trajectory definition from a file.

Once lodaed, the trajectory definition is set, and the trajectory is built.

Parameters `filename` (*str*) – The file to load

Raises `ValueError` – If the filename is not of type 'str'

`plot_traj`(*fig_kwargs={}*, *plot_kwargs={}*)

Plot the current trajectory (assuming 1 rep of the main segment)

Parameters

- `fig_kwargs` (*Any*) – Keyword args to pass to the matplotlib's figure function
- `plot_kwargs` (*Any*) – Keyword args to pass to the matplotlib's plotting function

Raises `RuntimeError` – If the trajectory has not been built

`save_definition`(*filename*)

Save the trajectory definition to a file.

Parameters `filename` (*str*) – The file to save

Raises

- `ValueError` – If the filename is not of type 'str'
- `RuntimeError` – If the trajectory definition is not set

`save_traj`(*filename*)

Save the trajectory to a file.

Parameters `filename` (*str*) – The file to save

Raises

- `ValueError` – If the filename is not of type 'str'
- `RuntimeError` – If the trajectory has not been built

`set_definition`(*definition*)

Set the trajectory definition manually.

The trajectory definition is set, and the trajectory is rebuilt.

Parameters `definition` (*dict*) – The trajectory definition to set

Raises `ValueError` – If the trajectory definition is not of type 'dict'

3.2 Trajectory Interpolation

Use interpolators to obtain trajectory functions where you input a time (or array of times) and return the trajectory at that timepoint (or timepoints).

3.2.1 Interpolator Class

This is the primary way to create an interpolation function. This handles many edge-cases to ensure behavior is the same as the real-life Ctrl-P control system.

class sorotraj.interpolator.**Interpolator**(*trajectory*)

Trajectory interpolator

trajectory

Trajectory to interpolate

Type dict

Examples

```
>>> interp = sorotraj.Interpolator(traj)
... actuation_fn = interp.get_interp_function(
...     num_reps=1,
...     speed_factor=1.0,
...     invert_direction=False)
... interp.get_final_time()
8.0
```

get_cycle_function(*num_reps=1*, *speed_factor=1.0*, *invert_direction=False*, *as_list=None*)

Get a function to return the current cycle number given time as an input

Parameters

- **num_reps** (*int*) – Number of times to repeat the "main" trajectory segment
- **speed_factor** (*float*) – Speed multiplier (times are multiplied by inverse of this)
- **invert_direction** (*Union[bool, list]*) – Invert the sign of the interpolated values. If True, all signs are flipped. If list, invert_direction is treated as a list of indices.

Returns

- **cycle_function** (*function*) – The cycle function
- **final_time** (*float*) – The end time of the trajectory

get_final_time()

Get the final time of the most-recent interpolator

(This function exists for backward compatibillity. In the future, obtain the final time from the "get_traj_function" instead.)

get_interp_function(*num_reps=1*, *speed_factor=1.0*, *invert_direction=False*, ***kwargs*)

Get a trajectory interpolation function with the specified parameters

(This function exists for backward compatibillity. In the future, use "get_traj_function" instead.)

Parameters

- **num_reps** (*int*) – Number of times to repeat the "main" trajectory segment
- **speed_factor** (*float*) – Speed multiplier (times are multiplied by inverse of this)
- **invert_direction** (*Union[bool, list]*) – Invert the sign of the interpolated values. If True, all signs are flipped. If list, invert_direction is treated as a list of indices.

Returns The trajectory interpolation function

Return type traj_function

Raises **ValueError** – If num_reps is less than 0, or if speed_factor is 0 or less

get_traj_function(*num_reps=1*, *speed_factor=1.0*, *invert_direction=False*)

Get a trajectory interpolation function with the specified parameters

Parameters

- **num_reps** (*int*) – Number of times to repeat the "main" trajectory segment
- **speed_factor** (*float*) – Speed multiplier (times are multiplied by inverse of this)
- **invert_direction** (*Union[bool, list]*) – Invert the sign of the interpolated values. If True, all signs are flipped. If list, invert_direction is treated as a list of indices.

Returns

- **traj_function** (*function*) – The trajectory interpolation function
- **final_time** (*float*) – The end time of the trajectory

3.2.2 Custom Back-End Interpolators

Several custom interpolation classes are used under the hood to make the functions behave like the physical control system. Below you can find documentation for these classes

class sorotraj.interpolator.**TrajectoryInterpolator**(*traj_unpacked*, *num_reps=1*, *speed_factor=1.0*, *invert_direction=False*, *fill_value=None*)

A trajectory interpolator based on specified parameters

Parameters

- **traj_unpacked** (*dict*) – Unpacked trajectory object (dict where keys are trajectory components with fields "time" and ""values")
- **num_reps** (*int, optional*) – Number of times to repeat the "main" trajectory segment
- **speed_factor** (*float, optional*) – Speed multiplier (times are multiplied by inverse of this)
- **invert_direction** (*Union[bool, list], optional*) – Invert the sign of the interpolated values. If True, all signs are flipped. If list, invert_direction is treated as a list of indices.
- **fill_value** (*Union[list, np.ndarray], optional*) – Default value of signals (only used when prefix and main are empty in the trajectory)

Raises **ValueError** – If all trajectory components are empty

get_final_time()

Get the final time of the trajectory

Returns **final_time** – The final time

Return type float

get_traj_function()
Get the trajectory function

Returns **traj_function** – The trajectory interpolation meta-function.

Return type function

traj_function(*x0*)
The trajectory interpolation function

Parameters **x0** (*Union[float, list, np.ndarray]*) –

Returns **output** – The trajectory at the given time point(s). If **x0** is a float, output is a 1D np.ndarray. If **x0** is a list or np.ndarray, output is a 2D np.ndarray.

Return type np.ndarray

Raises

- **ValueError** – If input is not a 1D array-like object
- **RuntimeError** – If the length of the output does not equal the length of the input

class sorotraj.interpolator.**WrappedInterp1d**(*x*, *y*, ***kwargs*)
Create a wrapping 1D interpolator

Parameters

- **x** (*dict*) – x points to use in interpolation
- **y** (*int, optional*) – Values to use for interpolation
- ****kwargs** (*optional*) – kwargs to pass to scipy.interpolate.interp1d().

get_function()
Get the wrapped interpolation function

Returns **wrapped_interp1d** – The wrapped interpolator function

Return type function

max_wrap(*x0*)
Calculate wrapped x values when x is greater than the wrapping bounds

Parameters **x0** (*np.ndarray*) – Values of x

Returns **wrapped_x0** – Values of x wrapped.

Return type np.ndarray

min_wrap(*x0*)
Calculate wrapped x values when x is less than the wrapping bounds

Parameters **x0** (*np.ndarray*) – Values of x

Returns **wrapped_x0** – Values of x wrapped.

Return type np.ndarray

wrapped_interp1d(*x0*)
The wrapped interp1d function. Input x0, return interpolated cyclic values

Parameters **x0** (*Union[float, list, np.ndarray]*) – Values of x where you want to interpolate

Returns **output** – The interpolated values of y at the given time point(s)

Return type np.ndarray

Raises `ValueError` – If the input is not 1D

`sorotraj.interpolator.`**`interp1d_patched`**(*x*, *y*, ***kwargs*)

Get a 1D interpolation function where single-length input data are handled

When the length of x and y is greater than 1, interp1d is used. When the legnth of x and y is 1, use the value of y for all values of x0.

Parameters **x0** (*Union[float, list, np.ndarray]*) – Values of x where you want to interpolate

Returns **patched_interp1d** – The patched interp1d function (same way the regular interp1d works)

Return type function

CHAPTER

FOUR

CONTRIBUTING

Contributing Checklist

- only through a new branch and reviewed PR (no pushes to master!)
- always bump the version of your branch by increasing the version number listed in setup.py

CHAPTER

FIVE

INDEX

CHAPTER

SIX

QUICK INSTALL

```
pip install sorotraj
```

CHAPTER

SEVEN

EXPLORE THE EXAMPLES

Check out the *Examples*, or run any of the files in the examples folder. The "*Build One Trajectory*" example is a great place to start!

CHAPTER

EIGHT

LINKS

- **Documentation:** Read the Docs
- **pip install:** View on PyPi
- **Source code:** Github

CHAPTER

NINE

CONTACT

If you have questions, or if you've done something interesting with this package, get in touch with Clark Teeple, or the Harvard Microrobotics Lab!

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

CHAPTER

TEN

USED IN…

Sorotraj has enabled several published works:

- [Graule *et al.*, 2021]
- [Teeple *et al.*, 2021]
- [Teeple *et al.*, 2021]

10.1 References

BIBLIOGRAPHY

[graule2020somo] Moritz A. Graule, Clark B Teeple, Thomas P McCarthy, Randall C St. Louis, Grace R Kim, and Robert J Wood. Somo: fast and accurate simulation of continuum robots in complex environments. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021.

[teeple2021active] Clark B Teeple, Grace R Kim, Moritz A. Graule, and Robert J Wood. An active palm enhances dexterity for soft robotic in-hand manipulation. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume, 11790–11796. 2021. doi:10.1109/ICRA48506.2021.9562049.

[teeple2021arrangement] Clark B Teeple, Randall C. St. Louis, Moritz A. Graule, and Robert J Wood. The role of digit arrangement in soft robotic in-hand manipulation. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021.

PYTHON MODULE INDEX

S

sorotraj.build_traj, 15
sorotraj.interpolator, 19

INDEX

B

build_traj() (*sorotraj.build_traj.TrajBuilder method*), 15

C

convert_definition() (*sorotraj.build_traj.TrajBuilder method*), 15
convert_traj() (*sorotraj.build_traj.TrajBuilder method*), 15

G

get_absolute_times() (*sorotraj.build_traj.TrajBuilder method*), 16
get_cycle_function() (*sorotraj.interpolator.Interpolator method*), 18
get_definition() (*sorotraj.build_traj.TrajBuilder method*), 16
get_final_time() (*sorotraj.interpolator.Interpolator method*), 18
get_final_time() (*sorotraj.interpolator.TrajectoryInterpolator method*), 19
get_flattened_trajectory() (*sorotraj.build_traj.TrajBuilder method*), 16
get_function() (*sorotraj.interpolator.WrappedInterp1d method*), 20
get_interp_function() (*sorotraj.interpolator.Interpolator method*), 18
get_traj_components() (*sorotraj.build_traj.TrajBuilder method*), 16
get_traj_function() (*sorotraj.interpolator.Interpolator method*), 19
get_traj_function() (*sorotraj.interpolator.TrajectoryInterpolator method*), 19
get_trajectory() (*sorotraj.build_traj.TrajBuilder method*), 17

I

interp1d_patched() (*in module sorotraj.interpolator*), 21
Interpolator (*class in sorotraj.interpolator*), 18

L

load_traj_def() (*sorotraj.build_traj.TrajBuilder method*), 17

M

max_wrap() (*sorotraj.interpolator.WrappedInterp1d method*), 20
min_wrap() (*sorotraj.interpolator.WrappedInterp1d method*), 20
module
    sorotraj.build_traj, 15
    sorotraj.interpolator, 19

P

plot_traj() (*sorotraj.build_traj.TrajBuilder method*), 17

S

save_definition() (*sorotraj.build_traj.TrajBuilder method*), 17
save_traj() (*sorotraj.build_traj.TrajBuilder method*), 17
set_definition() (*sorotraj.build_traj.TrajBuilder method*), 17
sorotraj.build_traj
    module, 15
sorotraj.interpolator
    module, 19

T

traj_function() (*sorotraj.interpolator.TrajectoryInterpolator method*), 20
TrajBuilder (*class in sorotraj.build_traj*), 15
trajectory (*sorotraj.interpolator.Interpolator attribute*), 18
TrajectoryInterpolator (*class in sorotraj.interpolator*), 19

V

`verbose` (*sorotraj.build_traj.TrajBuilder attribute*), 15

W

`wrapped_interp1d()` (*sorotraj.interpolator.WrappedInterp1d method*), 20
`WrappedInterp1d` (*class in sorotraj.interpolator*), 20