trep Documentation

Release 1.0.3

Elliot Johnson

October 03, 2017

Contents

Release v1.0.3

Date October 03, 2017

Trep is a Python module for modeling rigid body mechanical systems in generalized coordinates. It provides tools for calculating continuous and discrete dynamics (based on a midpoint variational integrator), and the first and second derivatives of both. Tools for trajectory optimization and other basic optimal control methods are also available.

You can find detailed installation instructions on our website. Many examples are included with the source code (browse online).

The *API Reference* has detailed documentation for each part of `trep`. We have also put together a detailed *Tutorials* that gives an idea of the capabilities and organization of `trep` by stepping through several example problems.

If you have any questions or suggestions, please head over to our project page.

CHAPTER 1

trep - Core Components

The central component of *trep* is the *System* class. A *System* is a collection of coordinate frames, forces, potential energies, and constraints that describe a mechanical system in generalized coordinates. The *System* is capable of calculating the continuous dynamics and the first and second derivatives of the dynamics.

System - A Mechanical System

The *System* object is the central component for modeling a mechanical system, and usually the first *trep* object you create. It contains the entire definition of the system, including all coordinate frames, configuration variables, constraints, etc.

System is responsible for calculating continuous dynamics and derivative, but it is also used for the underlying Lagrangian calculations in any discrete dynamics calculations.

A *System* has a single inertial coordinate frame, accessible through the `world_frame` attribute. Every other coordinate frame added to the system will be descended from this coordinate frame.

class `trep.`**System**
Create a new empty mechanical system.

- *Derivative Testing*
- *Structure Updates*

System Components

`System.`**`world_frame`**
The root spatial frame of the system. The world frame will always exist and cannot be changed other than adding child frames.

(read only)

`System.`**`frames`**
`System.`**`configs`**
`System.`**`dyn_configs`**
`System.`**`kin_configs`**
`System.`**`potentials`**
`System.`**`forces`**
`System.`**`inputs`**
`System.`**`constraints`**
Tuples of all the components in the system. **The order of components in these tuples defines their order throughout** *trep*. Any vector of configuration values will be the same order as *System.configs*, any vector of constraint forces will be the same order as *System.constraints*, etc.

For example, any array of numbers representing a configuration will be ordered according to *System.configs*. An array of constraint forces will correspond to the order of *System.constraints*.

configs is guaranteed to be ordered as the concatenation of *dyn_configs* and *kin_configs*:

```
System.configs = System.dyn_configs + System.kin_configs
```

The tuples are read only, but the components in them may be modified.

`System.`**`masses`**
A tuple of all the *Frame* objects in the system with non-zero mass properties.

(read only)

`System.`**`nQ`**
Number of configuration variables in the system. Equivalent to `len(system.configs)`.

(read only)

`System.`**`nQd`**
Number of dynamic configuration variables in the system. Equivalent to `len(system.dyn_configs)`.

(read only)

`System.`**`nQk`**
Number of kinematic configuration variables in the system. Equivalent to `len(system.kin_configs)`.

(read only)

`System.`**`nu`**
Number of inputs in the system. Equivalent to `len(system.inputs)`.

(read only)

`System.`**`nc`**
Number of constraints in the system. Equivalent to `len(system.constraints)`.

(read only)

Finding Specific Components

System.**get_frame**(*identifier*)
System.**get_config**(*identifier*)
System.**get_potential**(*identifier*)
System.**get_constraint**(*identifier*)
System.**get_force**(*identifier*)
System.**get_input**(*identifier*)

Find a specific component in the system. The identifier can be:

•Integer: Returns the component at that position in the corresponding tuple.

system.get_frame(i) = system.frames[i]

If the index is invalid, an `IndexError` exception is raised.

•String: Returns the component with the matching name:

system.get_config('theta').name == 'theta'

If no object has a matching name, a `KeyError` exception is raised.

•Object: Return the identifier unmodified:

```
system.get_force(force) == force
```

If the object is the incorrect type, a `TypeError` exception is raised:

```
>>> config = system.configs[0]
>>> system.get_frame(config)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
   File "/usr/local/lib/python2.7/dist-packages/trep/system.py", line 106, in␣
↪get_frame
    return self._get_object(identifier, Frame, self.frames)
  File "/usr/local/lib/python2.7/dist-packages/trep/system.py", line 509, in _
↪get_object
    raise TypeError()
TypeError
```

Importing and Exporting Frame Definitions

System.**import_frames**(*children*)

Adds children to this system's world frame using a special frame definition. See *Frame.import_frames()* for details.

System.**export_frames**(*system_name='system'*, *frames_name='frames'*, *tab_size=4*)

Create python source code to define this system's frames. The code is returned as a string.

System State

A *System* is a stateful object that has a current time, configuration, velocity, and input at all times. When working directly with a *System*, you are a responsible for setting the state.

System.**t**
: Current time of the system.

The other state information is actually spread out among each component in the system. For example, each current configuration value is stored in *Config.q*. These can be modified directly through each component, but the following attributes are usually more convenient for reading and writing multiple values at once.

System.**q**
System.**dq**
System.**ddq**
: The value, velocity, and acceleration of the complete configuration.

System.**qd**
System.**dqd**
System.**ddqd**
: The value, velocity, and acceleration of the dynamic configuration.

System.**qk**
System.**dqk**
System.**ddqk**
: The value, velocity, and acceleration of the kinematic configuration.

System.**u**
: The current values of the force inputs.

Reading the each attribute will return a numpy array of the current values.

The values can be set with three different methods:

- A dictionary that maps names to values:

```
>>> system.q = { 'x' : 1.0, 'theta' : 0.1}
```

  Any variables that are not named will be unchanged.

- A array-like list of numbers:

```
>>> system.q = [1.0, 0.1, 0.0, 2.3]
>>> system.q = np.array([1.0, 0.1, 0.0, 2.3])
```

  If the size of the array and the number of variables doesn't match, the shorter of the two will be used:

```
>>> system.nQ
4

# This only sets the first 2 configuration variables!
>>> system.q = [0.2, 5.0]
# This ignores the last 2 values!
>>> system.q = [0.5, 0, 1.1, 2.4, 9.0]
```

- A single number:

```
>>> system.q = 0
```

  This will set the entire configuration to the number.

System.**set_state**(*q=None*, *dq=None*, *u=None*, *ddqk=None*, *t=None*)
: Set the current state of the system, not including the "output" ddqd. The types of values accepted are the same as described above.

Constraints

System.**satisfy_constraints**(*tolerance=1e-10*, *verbose=False*, *keep_kinematic=False*, *constant_q_list=None*)

Modify the current configuration to satisfy the system constraints. Letting q_0 be the system's current configuration, this performs the optimization:

$$q = \arg\min_q |q - q_0|^2 \quad \text{s.t.} \quad h(q) = 0$$

The new configuration will be set in the system and returned. If the optimization fails, a `StandardError` exception is raised.

Setting *verbose* to `True` will make the optimization print out information to the console while it is running.

Setting *keep_kinematic* to `True` will modify above optimization to optimize over q_d instead of q; thus all kinematic configuration variables will be kept constant. Passing a list (or tuple) of configurations to *constant_q_list* will define an arbitrary set of configurations to hold constant during optimization. Internally this method uses *System.get_config()* thus, any valid *identifier* mentioned in *Finding Specific Components* section will work.

System.**minimize_potential_energy**(*tolerance=1e-10*, *verbose=False*, *keep_kinematic=False*, *constant_q_list=None*)

Modify the current configuration to satisfy the system constraints while also minimizing potential energy. The system velocity is set to zero, and thus the kinetic energy is zero. Therefore, the actual optimization that is solved is given by

$$q = \arg\min_q \left(-L(q, \dot{q} = 0)\right) \quad \text{s.t.} \quad h(q) = 0$$

The new configuration will be set in the system and returned. If the optimization fails, a `StandardError` exception is raised. This function may be useful for finding nearby equilibrium points that satisfy the system constraints.

Setting *verbose* to `True` will make the optimization print out information to the console while it is running.

Setting *keep_kinematic* to `True` will modify above optimization to optimize over q_d instead of q; thus all kinematic configuration variables will be kept constant. Passing a list (or tuple) of configurations to *constant_q_list* will define an arbitrary set of configurations to hold constant during optimization. Internally this method uses *System.get_config()* thus, any valid *identifier* mentioned in *Finding Specific Components* section will work.

Lagrangian Calculations

System.**total_energy**()

Calculate the total energy in the current state.

System.**L**()
System.**L_dq**(*q1*)
System.**L_dqdq**(*q1*, *q2*)
System.**L_dqdqdq**(*q1*, *q2*, *q3*)
System.**L_ddq**(*dq1*)
System.**L_ddqdq**(*dq1*, *q2*)
System.**L_ddqdqdq**(*dq1*, *q2*, *q3*)
System.**L_ddqdqdqdq**(*dq1*, *q2*, *q3*, *q4*)
System.**L_ddqddq**(*dq1*, *dq2*)
System.**L_ddqddqdq**(*dq1*, *dq2*, *q3*)

System.**L_ddqddqdqdq**(*dq1*, *dq2*, *q3*, *q4*)

Calculate the Lagrangian or it's derivatives at the current state. When calculating the Lagrangian derivatives, you must specify a variable to take the derivative with respect to.

Calculating $\frac{\partial L}{\partial q_0}$:

```
>>> q0 = system.configs[0]
>>> system.L_dq(q0)
```

Calculating $\frac{\partial^2 L}{\partial q_0 \partial \dot{q}_1}$:

```
>>> q0 = system.configs[0]
>>> q1 = system.configs[1]
>>> system.L_ddqdq(q1, q0)
```

Calculating $\frac{\partial^2 L}{\partial \dot{q}_0 \partial \dot{q}_1}$:

```
>>> q0 = system.configs[0]
>>> q1 = system.configs[1]
>>> system.L_ddqddq(q1, q0)

# Mixed partials always commute, so we could also do:
>>> system.L_ddqddq(q0, q1)
```

Calculate an entire derivative $\frac{\partial L}{\partial q}$:

```
>>> [system.L_dq(q) for q in system.configs]
```

Dynamics

System.**f**(*q=None*)

Calculate the dynamics at the current state, $\ddot{q}_d = f(q, \dot{q}, \ddot{q}_k, u, t)$.

This calculates the second derivative of the dynamic configuration variables. The results are also written to *Config.ddq*.

If q is None, the entire vector $\ddot{q}_d$ is returned. The array is in the same order as *System.dyn_configs*.

If q is specified, it must be a dynamic configuration variable, and it's second derivative will be returned. This could also be accessed as q.ddq.

Once the dynamics are calculated, the results are saved until the system's state changes, so repeated calls will not keep repeating work.

System.**f_dq**(*q=None*, *q1=None*)
System.**f_ddq**(*q=None*, *dq1=None*)
System.**f_dddk**(*q=None*, *k1=None*)
System.**f_du**(*q=None*, *u1=None*)

Calculate the first derivative of the dynamics with respect to the configuration, the configuration velocity, the kinematic acceleration, or the force inputs.

If both parameters are None, the entire first derivative is returned as a numpy array with the derivatives across the rows.

If any parameters are specified, they must be appropriate objects, and the function will return the specific information requested.

Calculating $\frac{\partial f}{\partial q_2}$:

```
>>> dq = system.configs[2]
>>> system.f_dq(q1=dq)
array([-0.076, -0.018, -0.03 , ...,  0.337, -0.098,  0.562])
```

Calculating $\frac{\partial f}{\partial \dot{q}}$:

```
>>> system.f_ddq()
array([[-0.001, -0.   , -0.   , ...,  0.001, -0.001,  0.   ],
[-0.   , -0.002, -0.   , ..., -0.   , -0.   ,  0.   ],
[-0.   , -0.   , -0.   , ...,  0.   , -0.   ,  0.   ],
...,
[-0.   , -0.001, -0.   , ..., -0.003, -0.   ,  0.   ],
[ 0.001, -0.   ,  0.   , ..., -0.005,  0.008, -0.   ],
[-0.001, -0.   , -0.   , ...,  0.008, -0.027,  0.   ]])
>>> system.f_ddq().shape
(22, 23)
```

Calculating $\frac{\partial \ddot{q}_4}{\partial \ddot{q}_{k0}}$:

```
>>> qk = system.kin_configs[0]
>>> q = system.dyn_configs[4]
>>> system.f_dddk(q, qk)
-0.31036045452513322
```

The first call to any of these functions will calculate and cache the entire first derivative. Once calculated, subsequent calls will not recalculate the derivatives until the system's state is changed.

System.**f_dqdq**(*q=None*, *q1=None*, *q2=None*)
System.**f_ddqdq**(*q=None*, *dq1=None*, *q2=None*)
System.**f_ddqddq**(*q=None*, *dq1=None*, *dq2=None*)
System.**f_dddkdq**(*q=None*, *k1=None*, *q2=None*)
System.**f_dudq**(*q=None*, *u1=None*, *q2=None*)
System.**f_duddq**(*q=None*, *u1=None*, *dq2=None*)
System.**f_dudu**(*q=None*, *u1=None*, *u2=None*)

Calculate second derivatives of the dynamics.

If no parameters specified, the entire second derivative is returned as a `numpy` array. The returned arrays are indexed with the two derivatives variables as the first two dimensions and the dynamic acceleration as the last dimension. In other words, calling:

```
system.f_dqdq(q, q1, q2)
```

is equivalent to:

```
result = system.f_dqdq()
result[q1.index, q2.index, q.index]
```

If any parameters are specified, they must be appropriate objects, and the function will return the specific information requested. For example:

```
system.f_dqdq(q2=q2)
```

is equivalent to:

```
result = system.f_dqdq()
result[:, q2.index, :]
```

The second derivatives are indexed opposite from the first derivatives because they are generally multiplied by an adjoint variable in practice. For example, the quantity:

$$z^T \frac{\partial^2 f}{\partial q \partial \dot{q}}$$

can be calculated as:

```
numpy.inner(system.f_dqddq(), z)
```

without having to do a transpose or specify a specific axis.

The first call to any of these functions will calculate and cache the entire second derivative. Once calculated, subsequent calls will not recalculate the derivatives until the system's state is changed.

Constraint Forces

System.**lambda_**(*constraint=None*)

System.**lambda_dq**(*constraint=None*, *q1=None*)
System.**lambda_ddq**(*constraint=None*, *dq1=None*)
System.**lambda_dddk**(*constraint=None*, *k1=None*)
System.**lambda_du**(*constraint=None*, *u1=None*)

System.**lambda_dqdq**(*constraint=None*, *q1=None*, *q2=None*)
System.**lambda_ddqdq**(*constraint=None*, *dq1=None*, *q2=None*)
System.**lambda_ddqddq**(*constraint=None*, *dq1=None*, *dq2=None*)
System.**lambda_dddkdq**(*constraint=None*, *k1=None*, *q2=None*)
System.**lambda_dudq**(*constraint=None*, *u1=None*, *q2=None*)
System.**lambda_duddq**(*constraint=None*, *u1=None*, *dq2=None*)
System.**lambda_dudu**(*constraint=None*, *u1=None*, *u2=None*)

These are functions for calculating the value of the constraint force vector λ and its derivatives. They have the same behavior and index orders as the *dynamics* functions.

Derivative Testing

System.**test_derivative_dq**(*func*, *func_dq*, *delta=1e-6*, *tolerance=1e-7*, *verbose=False*, *test_name='<unnamed>'*)
System.**test_derivative_ddq**(*func*, *func_ddq*, *delta=1e-6*, *tolerance=1e-7*, *verbose=False*, *test_name='<unnamed>'*)

Test the derivative of a function with respect to a configuration variable's time derivative and its numerical approximation.

func -> Callable taking no arguments and returning float or np.array

func_ddq -> Callable taking one configuration variable argument and returning a float or np.array.

delta -> perturbation to the current configuration to calculate the numeric approximation.

tolerance -> acceptable difference between the approximation and exact value. (|exact - approx| <= tolerance)

verbose -> Boolean indicating if a message should be printed for failures.

name -> String identifier to print out when reporting messages when verbose is true.

Returns False if any tests fail and True otherwise.

Structure Updates

Whenever a system is significantly modified, an internal function is called to make sure everything is consistent and resize the arrays used for calculating values as needed. You can register a function to be called after the system has been made consistent using the `add_structure_changed_func()`. This is useful if you building your own component that needs to be updated whenever the system's structure changed (See `Damping` for an example).

`System.`**`add_structure_changed_func`**(*function*)
Register a function to call whenever the system structure changes. This includes adding and removing frames, configuration variables, constraints, potentials, and forces.

Since every addition to the system triggers a structure update, building a large system can cause a long delay. In these cases, it is useful to stop the structure updates until the system is fully constructed and then perform the update once.

Warning: Be sure to remove all holds before performing any calculations with system.

`System.`**`hold_structure_changes`**`()`
Prevent the system from calling `System._update_structure()` (mostly). This can be called multiple times, but *`resume_structure_changes()`* must be called an equal number of times.

`System.`**`resume_structure_changes`**`()`
Stop preventing the system from calling `System._update_structure()`. The structure will only be updated once every hold has been removed, so calling this does not guarantee that the structure will be immediately update.

`Frame` – Coordinate Frame

The basic geometry of a mechanical system is defined by tree of coordinate frames in *`trep`*. The root of the tree is the fixed *`System.world_frame`*. Every other coordinate frame is defined by a coordinate transformation relative to its parent. The coordinate transformations are either fixed or parameterized by a single configuration variable. For example, a rotational joint is modeled as a rotation transformation where the angle is controlled by a configuration variable.

Coordinate frames also define the masses in the system. For every mass, a coordinate frame must be placed with the origin at the center of mass and the axes aligned with the principle axes of the rotational inertia. Coordinate frames can also be mass-less, or have a mass but no rotational inertia to model point masses.

A `Frame` object can calculate its global position and body velocity, and their derivatives.

Frame Transformation Types

There are a fixed set of coordinate transformations to define each frame:

Constant	Description
`trep.`**`RX`**	Rotation about the parent's X axis.
`trep.`**`RY`**	Rotation about the parent's Y axis.
`trep.`**`RZ`**	Rotation about the parent's Z axis.
`trep.`**`TX`**	Translation about the parent's X axis.
`trep.`**`TY`**	Translation about the parent's Y axis.
`trep.`**`TZ`**	Translation about the parent's Z axis.
`trep.`**`CONST_SE3`**	Constant SE(3) transformation from parent.
`trep.`**`WORLD`**	Unique World Frame.

The first six transformations can either be parameterized by a fixed constant or a configuration variable. The *`CONST_SE3`* transformation can only be fixed.

The *`WORLD`* transformation is reserved for the system's *`world_frame`*.

Defining the Frames

New coordinate frames can be directly defined using the *`Frame`* constructor. For example, this will create a simple pendulum:

```
>>> import trep
>>> system = trep.System()
>>> frame1 = trep.Frame(system.world_frame, trep.RX, "theta")
>>> frame2 = trep.Frame(frame1, trep.TZ, -1, mass=4)
```

This gets tedious and makes it difficult to see the mechanical structure, so *`trep`* provides an alternate method to declare frames using *`Frame.import_frames()`* and a few extra functions:

trep.**tx**(*param*, *name=None*, *kinematic=False*, *mass=0.0*)
trep.**ty**(*param*, *name=None*, *kinematic=False*, *mass=0.0*)
trep.**tz**(*param*, *name=None*, *kinematic=False*, *mass=0.0*)
trep.**rx**(*param*, *name=None*, *kinematic=False*, *mass=0.0*)
trep.**ry**(*param*, *name=None*, *kinematic=False*, *mass=0.0*)
trep.**rz**(*param*, *name=None*, *kinematic=False*, *mass=0.0*)
trep.**const_se3**(*se3*, *name=None*, *kinematic=False*, *mass=0.0*)
trep.**const_txyz**(*xyz*, *name*, *kinematic*, *mass*)

The parameters are the same as for creating new *Frame* objects directly, except the parent and transform type are gone. The transform type is implied by the function name. The parent will be implied by how we use the definition.

Frame.import_frames() expects a list of these definitions. For each definition, the frame will create a new frame and add it to it's children.

For example, suppose we had a frame with 6 children:

```
>>> child1 = Frame(parent, trep.TX, 'q1')
>>> child2 = Frame(parent, trep.TY, 'q2', name='favorite_child')
>>> child3 = Frame(parent, trep.TZ, 'q3')
>>> child4 = Frame(parent, trep.RX, 'q4')
>>> child5 = Frame(parent, trep.TX, 'q5')
```

Using *Frame.import_frames()*, this becomes:

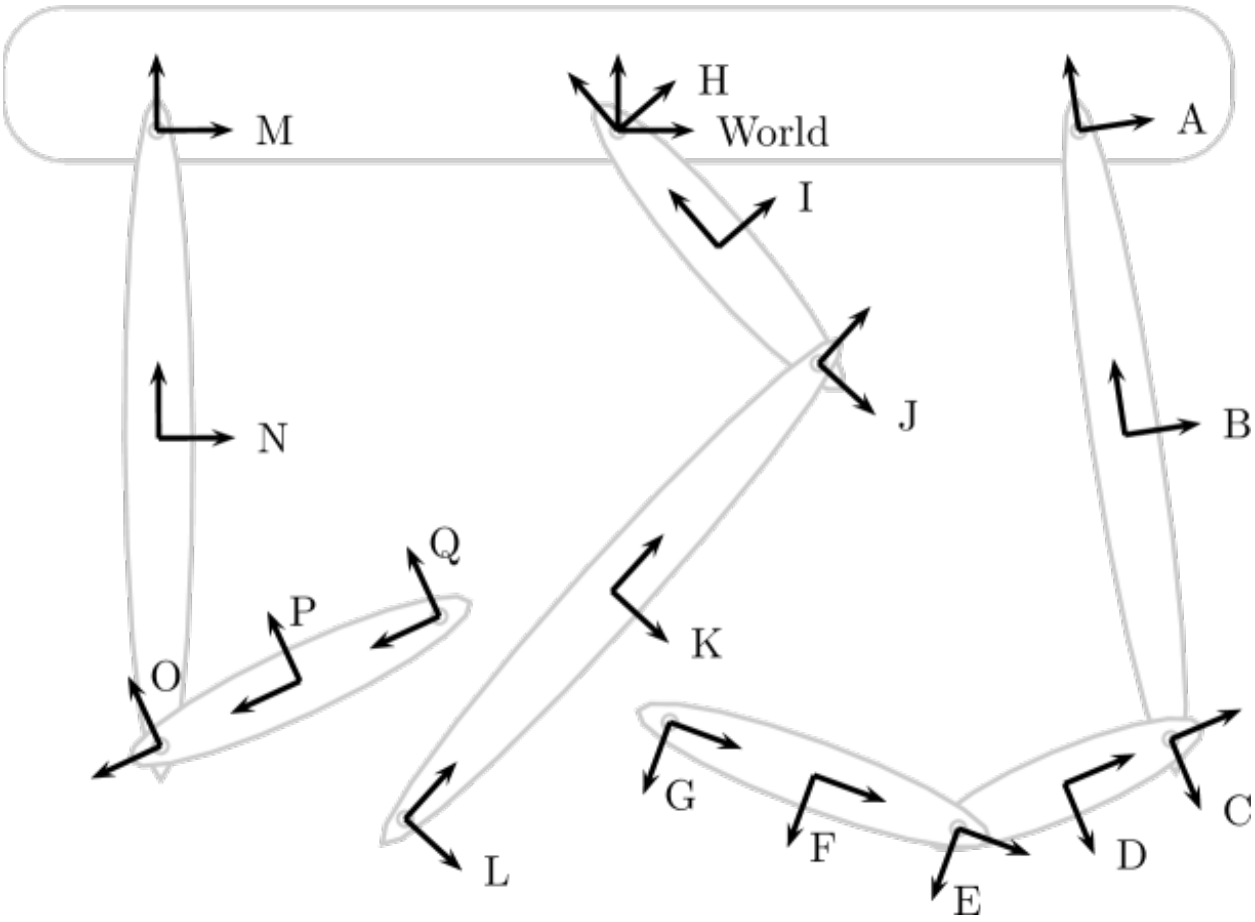
```
>>> children = [
...     trep.tx('q1'),
...     trep.ty('q2', name='favorite_child'),
...     trep.tz('q3'),
...     trep.rx('q4'),
...     trep.tx('q5')
...     ]
>>> parent.import_frames(children)
```

We can also add children to these children. If *Frame.import_frames()* finds a list after a frame definition, then it will call the new call the new child's `import_frames()` method with the new list.

For example, the *pendulum* we created earlier, can be defined as:

```
>>> import trep
>>> from trep import rx, tz
>>> children = [
...     rx('theta'), [
...             tz(-1, mass=4)
...     ]]
>>> system.world_frame.import_frames(children)
```

Since *Frame.import_frames()* works recursively, we can describe arbitrarily complex trees. Consider this more complicated example:



In the above image, the system is entirely 2D, and we are looking at the *x*-*z* plane of the *World* frame. Thus the arrows of each labeled coordinate frame are showing the positive *x* and positive *z* axes of that frame. The corresponding frame definition is:

```
import trep
from trep import tx, ty, tz, rx, ry, rz

system = trep.System()
frames = [
    ry('H', name='H'), [
        tz(-0.5, name='I', mass=1),
        tz(-1), [
            ry('J', name='J'), [
                tz(-1, name='K', mass=1),
                tz(-2, name='L')]]],
    tx(-1.5), [
        ry('M', name='M'), [
            tz(-1, name='N', mass=1),
            tz(-2), [
                ry('O', name='O'), [
                    tz(-0.5, name='P', mass=1),
                    tz(-1.0, name='Q')]]]],
    tx(1.5), [
        ry('A', name='A'), [
            tz(-1, name='B', mass=1),
            tz(-2), [
                ry('C', name='C'), [
```

```
                    tz(-0.375, name='D', mass=1),
                    tz(-0.75), [
                        ry('E', name='E'), [
                            tz(-0.5, name='F', mass=1),
                            tz(-1.0, name='G')
                            ]
                        ]
                    ]
                ]
            ]
        ]
    ]
system.import_frames(frames)
```

This is much more concise than defining the frames directly. It is also easier to see the structure of the system by taking advantage of how most Python editors will indent the nested lists. Trep also provides the function *Frame.tree_view()* for creating a visual representation of a system's tree structure. For the above system we can view the tree with the following command:

```
>>> print system.get_frame('World').tree_view()
<Frame 'World'>
   <Frame 'H' RY(H)>
      <Frame 'I' TZ(-0.5) 1.000000>
      <Frame 'None' TZ(-1.0) 0.000000>
         <Frame 'J' RY(J)>
            <Frame 'K' TZ(-1.0) 1.000000>
            <Frame 'L' TZ(-2.0) 0.000000>
   <Frame 'None' TX(-1.5) 0.000000>
      <Frame 'M' RY(M)>
         <Frame 'N' TZ(-1.0) 1.000000>
         <Frame 'None' TZ(-2.0) 0.000000>
            <Frame 'O' RY(O)>
               <Frame 'P' TZ(-0.5) 1.000000>
               <Frame 'Q' TZ(-1.0) 0.000000>
   <Frame 'None' TX(1.5) 0.000000>
      <Frame 'A' RY(A)>
         <Frame 'B' TZ(-1.0) 1.000000>
         <Frame 'None' TZ(-2.0) 0.000000>
            <Frame 'C' RY(C)>
               <Frame 'D' TZ(-0.375) 1.000000>
               <Frame 'None' TZ(-0.75) 0.000000>
                  <Frame 'E' RY(E)>
                     <Frame 'F' TZ(-0.5) 1.000000>
                     <Frame 'G' TZ(-1.0) 0.000000>
```

A convenience function is also provided to create constant SE(3) transformations from an angle and an axis.

`trep.`**`rotation_matrix`**(*theta*, *axis*)
Build a 4x4 SE3 matrix corresponding to a rotation of theta radians around axis.

Frame Objects

class `trep.`**`Frame`**(*parent*, *transform*, *param*[, *name=None*, *kinematic=False*, *mass=0.0*])
Create a new coordinate frame attached to *parent*. *transform* must be one of the *transformation constants* and defines how the new frame is related to the parent frame.

param is either a number or a string. If *param* is an number, the frame is fixed relative to the parent.

If *param* is a string, the frame's coordinate transformation is controlled by a configuration variable. A new configuration variable will be created using *param* as the name. By default, the new configuration variable will be dynamic. If *kinematic* is `True`, it will be kinematic.

If *transform* is `CONST_SE`, *param* must be a 4x4 matrix that defines the Frame's constant SE(3) transformation relative to *parent*.

name is an optional name for the frame.

mass defines the inertial properties of the frame. If *mass* is a single number, it is the frame's linear mass and the rotational inertia will be zero.

mass can also be a list of 4 numbers that define the Frame's *mass*, *Ixx*, *Iyy*, and *Izz* inertial properties.

`Frame.`**`uses_config`**`(`*q*`)`

Parameters **q** (*Config*) – A configuration variable in the system.

Determine if this coordinate frame depends on the configuration variable *q*.

When a frame does not depend on a configuration variable, the derivatives of its position and velocity will always be zero. You can usually improve the performance of new constraints, potentials, and forces by checking for this and avoiding unnecessary calculations.

`Frame.`**`system`**
The *System* that the frame belongs to.

(read only)

`Frame.`**`config`**
The *Config* that parameterizes the frame's transformation. This will be `None` for fixed transformations.

(read only)

`Frame.`**`parent`**
The parent frame of this frame. This is always `None` for the *System.world_frame*, and always a valid *Frame* otherwise.

(read only)

`Frame.`**`children`**
A tuple of the frame's child frames.

(read only)

`Frame.`**`flatten_tree`**`()`
Create a list of the frame and its entire sub-tree. There is no guarantee on the ordering other than it won't change as long as no frames are added to the system.

`Frame.`**`tree_view`**`(`*indent=0*`)`
Return a string that visually describes this frame and it's descendants.

Importing and Exporting Frames

`Frame.`**`import_frames`**`(`*children*`)`
Import a tree of frames from a tree description. See *Defining the Frames*. The tree will be added to this frame's children.

`Frame.`**`export_frames`**`(`*tabs=0*, *tab_size=4*`)`
Create python source code to define this frame and it's sub-tree. The code is returned as a string.

Transform Information

Frame.**transform_type**
: Transformation type of the coordinate frame. This will be one of the constants described in *Frame Transformation Types*.

  (read only)

Frame.**transform_value**
: Current value of the frame's transformation parameters. This will either be the fixed transformation parameter or the value of the frame's configuration variable.

Frame.**set_SE3**(*Rx=(1, 0, 0)*, *Ry=(0, 1, 0)*, *Rz=(0, 0, 1)*, *p=(0, 0, 0)*)
: Set the SE3 transformation for a const_SE3 frame.

Inertial Properties

Frame.**mass**
Frame.**Ixx**
Frame.**Iyy**
Frame.**Izz**

Frame.**set_mass**(*mass*, *Ixx=0.0*, *Iyy=0.0*, *Izz=0.0*)
: The coordinate frame can have mass at its origin and rotational inertia about each axis.

Local Frame Transformations

Frame.**lg**()
Frame.**lg_dq**()
Frame.**lg_dqdq**()
Frame.**lg_dqdqdq**()
Frame.**lg_dqdqdqdq**()
: These functions calculate the coordinate transformation to the frame from its parent in SE(3) and the derivatives of coordinate transformation with respect to the frame's configuration variable. If the frame is fixed, the derivatives will be zero. The returned values are 4x4 `numpy` arrays.

Frame.**lg_inv**()
Frame.**lg_inv_dq**()
Frame.**lg_inv_dqdq**()
Frame.**lg_inv_dqdqdq**()
Frame.**lg_inv_dqdqdqdq**()
: These functions calculate the inverse coordinate transformation to the frame from its parent (the transformation from the frame to its parent.) and its derivatives.

Global Frame Transformations

Frame.**g**()
Frame.**g_dq**(*q1*)
Frame.**g_dqdq**(*q1*, *q2*)
Frame.**g_dqdqdq**(*q1*, *q2*, *q3*)
Frame.**g_dqdqdqdq**(*q1*, *q2*, *q3*, *q4*)
: These functions calculate the global coordinate transformation in SE(3) for the frame (i.e, the coordinate transformation from the world frame to this frame) and its derivatives with respect to arbitrary configuration variables. The returned values are 4x4 `numpy` arrays.

Frame.**g_inv**()
Frame.**g_inv_dq**(*q1*)
Frame.**g_inv_dqdq**(*q1*, *q2*)
These functions calculate the inverse of the global coordinate transformation in SE(3) for the frame (i.e, the coordinate transformation from this frame to the world frame) and its derivatives with respect to arbitrary configuration variables. The returned values are 4x4 numpy arrays.

Frame.**p**()
Frame.**p_dq**(*q1*)
Frame.**p_dqdq**(*q1*, *q2*)
Frame.**p_dqdqdq**(*q1*, *q2*, *q3*)
Frame.**p_dqdqdqdq**(*q1*, *q2*, *q3*, *q4*)
These functions calculate the global position of the coordinate frame in R^3 for the frame (i.e, the origin's location with respect to the world frame to this frame) and its derivatives with respect to arbitrary configuration variables. The returned values are 4x4 numpy arrays.

Body Velocity Calculations

Frame.**twist_hat**()

Frame.**vb**()
Calculate the twist and the body velocity of the coordinate frame in se(3). The returned values are 4x4 numpy arrays.

Frame.**vb_dq**(*q1*)
Frame.**vb_ddq**(*dq1*)
Calculate first derivative of the body velocity with respect to the value or velocity of a configuration variable. The returned values are 4x4 numpy arrays.

Frame.**vb_dqdq**(*q1*, *q2*)
Frame.**vb_dqdqdq**(*q1*, *q2*, *q3*)
Calculate second derivative of the body velocity with respect to the values of configuration variables. The returned values are 4x4 numpy arrays.

Frame.**vb_ddqdq**(*dq1*, *q2*)
Frame.**vb_ddqdqdq**(*dq1*, *q2*, *q3*)
Frame.**vb_ddqdqdqdq**(*dq1*, *q2*, *q3*, *q4*)
Calculate derivative of the body velocity with respect to the velocity of *dq1* and the values of the other configuration variables. The returned values are 4x4 numpy arrays.

Config – Configuration Variables

class trep.**Config**(*system*[, *name=None*, *kinematic=False*])

Parameters

- **system** – An instance of *System* to add the variable to.
- **name** – A string that uniquely identifies the configuration variable.
- **kinematic** – True to define a kinematic configuration variable.

An *Config* instance represents a configuration variable in the generalized coordinates, q, of a mechanical system. Configuration variables primarily parameterize rigid-body transformations between coordinate frames in the system, though sometimes special configuration variables are used only by constraints.

Configuration variables are created automatically by *Frame* and *Constraint* when needed, so you do not need to create them directly unless defining a new constraint type.

If a *name* is provided, it can be used to identify and retrieve configuration variables.

The current values and their derivatives of a configuration variable can be accessed directly (*q*, *dq*, *ddq*) for an individual variable, or through *System* to access all configuration variables at once (*System.q*, *System.dq*, *System.ddq*)

Warning: Currently *trep* does not enforce unique names for configuration variables. It is recommended to provide a unique name for every *Config* so they can be unambiguously retrieved by *System.get_config()*.

Dynamic and Kinematic Variables

A configuration variable can be dynamic or kinematic. Dynamic configuration variables, q_d, are traditional configuration variables. Their trajectory is determined by the system dynamics ($\ddot{q_d} = f(q(t), \dot{q}(t), u(t))$). Dynamic configuration variables must parameterize a rigid-body transformation, and there must be a coordinate frame with non-zero mass that depends on the dynamic variable (directly or indirectly).

Kinematic configuration variables, q_k, are considered perfectly controllable. Their second derivative is specified directly as an additional input to the system ($\ddot{q_k} = u_k(t)$). Kinematic configuration variables can parameterize any rigid body transformation in a system, regardless of whether or not a frame with non-zero mass depends directly or indirectly on the variable. Additionally, kinematic configuration variables that do not parameterize a transformation can be defined by constraint functions. For example, the `Distance` constraint uses a kinematic configuration variable to maintain a time-varying distance between two coordinate frames.

Config Objects

`Config.`**`q`**
The current value of the configuration variable.

`Config.`**`dq`**
The 1st time derivative (velocity) of the configuration variable (i.e, $\dot{q}$).

`Config.`**`ddq`**
The 2nd time derivative (acceleration) of the configuration variable (i.e, $\ddot{q}$).

`Config.`**`kinematic`**
Boolean indicating if this configuration variable is dynamic or kinematic.

(read only)

`Config.`**`system`**
The *System* that this configuration variable belongs to.

(read only)

`Config.`**`frame`**
The *Frame* that depends on this configuration variable for its transformation, or `None` if it is not used by a coordinate frame (e.g, a kinematic configuration variable in a constraint).

(read only)

`Config.`**`name`**
The name of this configuration variable or `None`.

`Config.`**`index`**
Index of the configuration variable in *`System.configs`*.

For dynamic configuration variables, this is also the index of the variable in *`System.dyn_configs`*.

(read only)

`Config.`**`k_index`**
For kinematic configuration variables, this is the index of this variable in *`System.kin_configs`*.

(read-only)

`Config.`**`masses`**
A tuple of all *`coordinate frames`* with non-zero masses that depend (directly or indirectly) on this configuration variable.

(read-only)

`Input` – Input Variables

class `trep.`**`Input`**(*system*[, *name=None*])

Parameters

- **`system`** – An instance of *`System`* to add the variable to.
- **`name`** – A string that uniquely identifies the input variable.

An *`Input`* instance represents a single variable in the input vector, u, of a mechanical system. Inputs are used by *`Force`* to apply non-conservative forcing to the system (e.g, a torque or body wrench)

Input variables are created automatically by implementations of *`Force`* when needed, so you do not need to create them directly unless defining a new force type. In that case, they should be created using `Force._create_input()`.

If a *name* is provided, it can be used to identify and retrieve input variables.

The current value of a single input variable can be accessed directly (*`u`*) or through *`System`* to access all input variables at once (*`System.u`*).

Warning: Currently *`trep`* does not enforce unique names for input variables. It is recommended to provide a unique name for every *`Input`* so they can be unambiguously retrieved by *`System.get_input()`*.

Input Objects

`Input.`**`u`**
The current value of the input.

`Input.`**`system`**
The system that owns this input variable.

(read only)

`Input.`**`force`**
The force that uses this input variable.

(read only)

`Input.`**`name`**
The name of this input variable or `None`.

`Input.`**`index`**
The index of the input in *`System.inputs`*.

`Force` – Base Class for Forces

class `trep.`**`Force`**(*system*[, *name=None*])

Parameters

- **`system`** (*`System`*) – An instance of *`System`* to add the force to.
- **`name`** – A string that uniquely identifies the force.

This is the base class for all forces in a *`System`*. It should never be created directly. Forces are created by instantiating a specific type of force..

See *trep.forces - Forces* for the built-in types of forces. Additional forces can be added through either the Python or C-API.

Forces are used to include non-conservative and control forces in a mechanical system. Forces must be expressed in the generalized coordinates of the system. Conservative forces like gravity or spring-like potentials should be implemented using *`Potential`* instead.

Force Objects

`Force.`**`system`**
The *`System`* that this force belongs to.

(read-only)

`Force.`**`name`**
The name of this force or `None`.

`Force.`**`f`**(*q*)

Parameters **`q`** (*`Config`*) – Configuration variable

Calculate the force on configuration variable q at the current state of the system.

Table 1.1: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	?
1st Derivative	?
2nd Derivative	?
Discrete Dynamics	?
1st Derivative	?
2nd Derivative	?

`Force.`**`f_dq`**(*q*, *q1*)

Parameters

- **q** (*Config*) – Configuration variable
- **q1** (*Config*) – Configuration variable

Calculate the derivative of the force on configuration variable q with respect to the value of $q1$.

Table 1.2: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	?
1st Derivative	?
2nd Derivative	?
Discrete Dynamics	?
1st Derivative	?
2nd Derivative	?

`Force.`**`f_ddq`**(*q*, *dq1*)

Parameters

- **q** (*Config*) – Configuration variable
- **dq1** (*Config*) – Configuration variable

Calculate the derivative of the force on configuration variable q with respect to the velocity of $dq1$.

Table 1.3: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	?
1st Derivative	?
2nd Derivative	?
Discrete Dynamics	?
1st Derivative	?
2nd Derivative	?

`Force.`**`f_du`**(*q*, *u1*)

Parameters

- **q** (*Config*) – Configuration variable
- **u1** (*Input*) – Input variable

Calculate the derivative of the force on configuration variable q with respect to the value of $u1$.

Table 1.4: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	?
1st Derivative	?
2nd Derivative	?
Discrete Dynamics	?
1st Derivative	?
2nd Derivative	?

`Force.`**`f_dqdq`**(*q*, *q1*, *q2*)

Parameters

- **q** (*Config*) – Configuration variable
- **q1** (*Config*) – Configuration variable
- **q2** (*Config*) – Configuration variable

Calculate the second derivative of the force on configuration variable *q* with respect to the value of *q1* and the value of *q2*.

Table 1.5: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	?
1st Derivative	?
2nd Derivative	?
Discrete Dynamics	?
1st Derivative	?
2nd Derivative	?

`Force.`**`f_ddqdq`**(*q*, *dq1*, *q2*)

Parameters

- **q** (*Config*) – Configuration variable
- **dq1** (*Config*) – Configuration variable
- **q2** (*Config*) – Configuration variable

Calculate the second derivative of the force on configuration variable *q* with respect to the velocity of *dq1* and the value of *q2*.

Table 1.6: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	?
1st Derivative	?
2nd Derivative	?
Discrete Dynamics	?
1st Derivative	?
2nd Derivative	?

`Force.`**`f_ddqddq`**(*q*, *dq1*, *dq2*)

Parameters

- **q** (*Config*) – Configuration variable
- **dq1** (*Config*) – Configuration variable
- **dq2** (*Config*) – Configuration variable

Calculate the second derivative of the force on configuration variable *q* with respect to the velocity of *dq1* and the velocity of *q2*.

Table 1.7: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	?
1st Derivative	?
2nd Derivative	?
Discrete Dynamics	?
1st Derivative	?
2nd Derivative	?

`Force.f_dudq`(*q*, *u1*, *q2*)

Parameters

- **q** (*Config*) – Configuration variable
- **u1** (*Input*) – Input variable
- **q2** (*Config*) – Configuration variable

Calculate the second derivative of the force on configuration variable *q* with respect to the value of *u1* and the value of *q2*.

Table 1.8: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	?
1st Derivative	?
2nd Derivative	?
Discrete Dynamics	?
1st Derivative	?
2nd Derivative	?

`Force.f_duddq`(*q*, *u1*, *dq2*)

Parameters

- **q** (*Config*) – Configuration variable
- **u1** (*Input*) – Input variable
- **dq2** (*Config*) – Configuration variable

Calculate the second derivative of the force on configuration variable *q* with respect to the value of *u1* and the velocity of *q2*.

Table 1.9: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	?
1st Derivative	?
2nd Derivative	?
Discrete Dynamics	?
1st Derivative	?
2nd Derivative	?

`Force.f_dudu`(*q*, *u1*, *u2*)

Parameters

- **q** (*Config*) – Configuration variable
- **u1** (*Input*) – Input variable
- **u2** (*Input*) – Input variable

Calculate the second derivative of the force on configuration variable *q* with respect to the value of *u1* and the value of *u2*.

Table 1.10: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	?
1st Derivative	?
2nd Derivative	?
Discrete Dynamics	?
1st Derivative	?
2nd Derivative	?

Verifying Derivatives of the Force

It is important that the derivatives of `f()` are correct. The easiest way to check their correctness is to approximate each derivative using numeric differentiation. These methods are provided to perform this test. The derivatives are only compared at the current configuration of the system. For improved coverage, try running each test several times at different configurations.

`Force.`**validate_h_dq**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Force.`**validate_h_ddq**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Force.`**validate_h_du**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Force.`**validate_h_dqdq**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Force.`**validate_h_ddqdq**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Force.`**validate_h_ddqddq**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Force.`**validate_h_dudq**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Force.`**validate_h_duddq**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Force.`**validate_h_dudu**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)

Parameters

- **delta** – Amount to add to each configuration, velocity, or input.
- **tolerance** – Acceptable difference between the calculated and approximate derivatives
- **verbose** – Boolean to print error and result messages.

Return type Boolean indicating if all tests passed

Check the derivatives against the approximate numeric derivative calculated from one less derivative (ie, approximate *f_dq()* from *f()* and *f_dudq()* from *f_du()*).

See *System.test_derivative_dq()*, *System.test_derivative_ddq()*, and `System.test_derivative_du()` for details of the approximation and comparison.

Visualization

`Force.`**opengl_draw**()
Draw a representation of this force in the current OpenGL context. The OpenGL coordinate frame will be in the

System's root coordinate frame.

This function is called by the automatic visualization tools. The default implementation does nothing.

`Potential` – Base Class for Potential Energies

class `trep.`**`Potential`**(*system*[, *name=None*])

Parameters

- **`system`** (*`System`*) – An instance of *`System`* to add the potential to.
- **`name`** – A string that uniquely identifies the potential energy.

This is the base class for all potential energies in a *`System`*. It should never be created directly. Potential energies are created by instantiating a specific type of potential energy.

See *trep.potentials - Potential Energies* for the built-in types of potential energy. Additional potentials can be added through either the Python or C-API.

Potential energies represent conservative forces in a mechanical system like gravity and springs. Implementing these forces as potentials energies instead of generalized forces will result in improved simulations with better energetic and momentum conserving properties.

Potential Objects

`Potential.`**`system`**
The *`System`* that this potential belongs to.

(read-only)

`Potential.`**`name`**
The name of this potential energy or `None`.

`Potential.`**`V`**`()`

Return type `Float`

Return the value of this potential energy at the system's current state. This function should be implemented by derived Potentials.

Table 1.11: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	n
1st Derivative	n
2nd Derivative	n
Discrete Dynamics	n
1st Derivative	n
2nd Derivative	n

Note: This actual potential value is not used in discrete or continuous time dynamics/derivatives, so you do not need to implement it unless you need it for your own calculations. However, implementing it allows one to compare the derivative *`V_dq()`* with a numeric approximation based on *`V()`* to help debug your potential.

`Potential.`**`V_dq`**(*q1*)

Parameters **q1** (*Config*) – Derivative variable

Return type `Float`

Return the derivative of V with respect to *q1*.

Table 1.12: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	Y
1st Derivative	Y
2nd Derivative	Y
Discrete Dynamics	Y
1st Derivative	Y
2nd Derivative	Y

`Potential.`**`V_dqdq`**(*q1*, *q2*)

Parameters

- **q1** (*Config*) – Derivative variable
- **q2** (*Config*) – Derivative variable

Return type `Float`

Return the second derivative of V with respect to *q1* and *q2*.

Table 1.13: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	n
1st Derivative	Y
2nd Derivative	Y
Discrete Dynamics	Y
1st Derivative	Y
2nd Derivative	Y

`Potential.`**`V_dqdqdq`**(*q1*, *q2*, *q3*)

Parameters

- **q1** (*Config*) – Derivative variable
- **q2** (*Config*) – Derivative variable
- **q3** (*Config*) – Derivative variable

Return type `Float`

Return the third derivative of V with respect to *q1*, *q2*, and *q3*.

Table 1.14: **Required for Calculations**

Desired Calculation	Required
Continuous Dynamics	n
1st Derivative	n
2nd Derivative	Y
Discrete Dynamics	n
1st Derivative	n
2nd Derivative	Y

Verifying Derivatives of the Potential

It is important that the derivatives of `V()` are correct. The easiest way to check their correctness is to approximate each derivative using numeric differentiation. These methods are provided to perform this test. The derivatives are only compared at the current configuration of the system. For improved coverage, try running each test several times at different configurations.

`Potential.`**`validate_V_dq`**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Potential.`**`validate_V_dqdq`**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Potential.`**`validate_V_dqdqdq`**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)

Parameters

- **`delta`** – Amount to add to each configuration
- **`tolerance`** – Acceptable difference between the calculated and approximate derivatives
- **`verbose`** – Boolean to print error and result messages.

Return type Boolean indicating if all tests passed

Check the derivatives against the approximate numeric derivative calculated from one less derivative (i.e,, approximate *`V_dq()`* from *`V()`* and *`V_dqdq()`* from *`V_dq()`*).

See *`System.test_derivative_dq()`* for details of the approximation and comparison.

Visualization

`Potential.`**`opengl_draw`**`()`
Draw a representation of this potential energy in the current OpenGL context. The OpenGL coordinate frame will be in the System's root coordinate frame.

This function is called by the automatic visualization tools. The default implementation does nothing.

`Constraint` – Base Class for Holonomic Constraints

class `trep.`**`Constraint`**(*system*[, *name=None*, *tolerance=1e-10*])

Parameters

- **`system`** (*`System`*) – An instance of *`System`* to add the constraint to.
- **`name`** – A string that uniquely identifies the constraint.
- **`tolerance`** (`Float`) – Tolerance to consider the constraint satisfied.

This is the base class for all holonomic constraints in a *System*. It should never be created directly. Constraints are created by instantiating a specific type of constraint.

See *trep.constraints - Holonomic Constraints* for the built-in types of constraints. Additional constraints can be added through either the Python or C-API.

Holonomic constraints restrict the allowable configurations of a mechanical system. Every constraint has an associated constraint function $h(q) : Q \to R$. A configuration q is acceptable if and only if $h(q) = 0$.

Constraint Objects

`Constraint.`**`system`**
The *System* that this constraint belongs to.

(read-only)

`Constraint.`**`name`**
The name of this constraint or `None`.

`Constraint.`**`index`**
The index of the constraint in *System.constraints*. This is also the index of the constraint's force in any values of λ or its derivatives used through *trep*.

(read-only)

`Constraint.`**`tolerance`**
The constraint should be considered satisfied if $|h(q)| < tolerance$. This is primarly used by the variational integrator when it finds the next configuration, or by *System.satisfy_constraints()*.

`Constraint.`**`h`**`()`

Return type `Float`

Return the value of the constraint function at the system's current state. This function should be implemented by derived Constraints.

`Constraint.`**`h_dq`**`(`*q1*`)`

Parameters **q1** (*Config*) – Derivative variable

Return type `Float`

Return the derivative of h with respect to *q1*.

`Constraint.`**`h_dqdq`**`(`*q1*, *q2*`)`

Parameters

- **q1** (*Config*) – Derivative variable
- **q2** (*Config*) – Derivative variable

Return type `Float`

Return the second derivative of h with respect to *q1* and *q2*.

`Constraint.`**`h_dqdqdq`**`(`*q1*, *q2*, *q3*`)`

Parameters

- **q1** (*Config*) – Derivative variable
- **q2** (*Config*) – Derivative variable
- **q3** (*Config*) – Derivative variable

Return type `Float`

Return the third derivative of h with respect to *q1*, *q2*, and *q3*.

`Constraint.`**`h_dqdqdqdq`**(*q1*, *q2*, *q3*, *q4*)

Parameters

- **q1** (*Config*) – Derivative variable
- **q2** (*Config*) – Derivative variable
- **q3** (*Config*) – Derivative variable
- **q4** (*Config*) – Derivative variable

Return type `Float`

Return the fourth derivative of h with respect to *q1*, *q2*, *q3*, and *q4*.

Verifying Derivatives of the Constraint

It is important that the derivatives of `h()` are correct. The easiest way to check their correctness is to approximate each derivative using numeric differentiation. These methods are provided to perform this test. The derivatives are only compared at the current configuration of the system. For improved coverage, try running each test several times at different configurations.

`Constraint.`**`validate_h_dq`**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Constraint.`**`validate_h_dqdq`**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Constraint.`**`validate_h_dqdqdq`**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)
`Constraint.`**`validate_h_dqdqdqdq`**(*delta=1e-6*, *tolerance=1e-6*, *verbose=False*)

Parameters

- **delta** – Amount to add to each configuration
- **tolerance** – Acceptable difference between the calculated and approximate derivatives
- **verbose** – Boolean to print error and result messages.

Return type Boolean indicating if all tests passed

Check the derivatives against the approximate numeric derivative calculated from one less derivative (ie, approximate *h_dq()* from *h()* and *h_dqdq()* from *h_dq()*).

See *System.test_derivative_dq()* for details of the approximation and comparison.

Visualization

`Constraint.`**`opengl_draw`**()

Draw a representation of this constraint in the current OpenGL context. The OpenGL coordinate frame will be in the System's root coordinate frame.

This function is called by the automatic visualization tools. The default implementation does nothing.

`MidpointVI` - Midpoint Variational Integrator

The *MidpointVI* class implements a variational integrator using a midpoint quadrature. The integrator works with any system, including those with constraints, forces, and kinematic configuration variables. The integrator implements full first and second derivatives of the discrete dynamics.

For information on variational integrators (including relevant notation used in this manual), see *Discrete Dynamics and Variational Integrators*.

The Midpoint Variational Integrator is defined by the approximations for L_d and $f^{\pm}$:

$$L_2(q_1, q_2, t_1, t_2) = (t_2 - t_1)L\left(\frac{q_1 + q_2}{2}, \frac{q_2 - q_1}{t_2 - t_1}\right)$$
$$f_2^-(q_{k-1}, q_k, t_{k-1}, t_k) = (t_k - t_{k-1})f\left(\frac{q_{k-1} + q_k}{2}, \frac{q_k - q_{k-1}}{t_k - t_{k-1}}, u_{k-1}, \frac{t_k - t_{k-1}}{2}\right)$$
$$f_d^+(q_{k-1}, q_k, t_{k-1}, t_k) = 0$$

where L and f are the continuous Lagrangian and generalized forcing on the system.

MidpointVI Objects

class `trep.`**`MidpointVI`**(*system*, *tolerance=1e-10*, *num_threads=None*)

Create a new empty mechanical system. *system* is a valid `System` object that will be simulation.

***tolerance* sets the desired tolerance of the root solver when** solving the DEL equation to advance the integrator.

`MidpointVI` makes use of multithreading to speed up the calculations. *num_threads* sets the number of threads used by this integrator. If *num_threads* is `None`, the integrator will use the number of available processors reported by Python's `multiprocessing` module.

`MidpointVI.`**`system`**

The `System` that the integrator simulates.

`MidpointVI.`**`nq`**

Number of configuration variables in the system.

`MidpointVI.`**`nd`**

Number of dynamic configuration variables in the system.

`MidpointVI.`**`nk`**

Number of kinematic configuration variables in the system.

`MidpointVI.`**`nu`**

Number of input variables in the system.

`MidpointVI.`**`nc`**

Number of constraints in the system.

Integrator State

In continuous dynamics the *System* only calculates the second derivative, leaving the actual simulation to be done by separate numerical integrator. The discrete variational integrator, on the other hand, performs the actual simulation by finding the next state. Because of this, *MidpointVI* objects store two discrete states: one for the previous time step t_1, and one for the new/current time step t_2.

Also unlike the continuous *System*, these variables are rarely set directly. They are typically modified by the initialization and stepping methods.

MidpointVI.**t1**
MidpointVI.**q1**
MidpointVI.**p1**
: State variables for the previous time. *q1* is the entire configuration (dynamic and kinematic). *p1* is discrete momemtum, which only has a dynamic component, not a kinematic component.

MidpointVI.**t2**
MidpointVI.**q2**
MidpointVI.**p2**
: State variables for the current/new time.

MidpointVI.**u1**
: The input vector at t_1.

MidpointVI.**v2**
: The finite-differenced velocity of the kinematic configuration variables at t_2 i.e. $v_2 = \frac{q_{k2} - q_{k1}}{t_2 - t_1}$. If $t_2 = t_1$ the built-in Python constant `None` is returned.

MidpointVI.**lambda1**
: The constraint force vector at t_1. These are the constraint forces used for the system to move from t_1 to t_2.

Initialization

MidpointVI.**initialize_from_state**(*t1*, *q1*, *p1*, *lambda1=None*)
: Initialize the integrator from a known state (time, configuration, and momentum). This prepares the integrator to start simulating from time t_1.

  lambda1 can optionally specify the initial constraint vector. This serves at the initial guess for the simulation's root solving algorithm. If you have a simulation that you are trying to re-simulate, but from a different starting time (e.g, you saved a forward simulation and now want to move backwards and calculate the linearization at each time step for a LQR problem.), it is a good idea to save *lambda1* during the simulation and set it here. Otherwise, the root solver can find a slightly different solution which eventually diverges. If not specified, it defaults to the zero vector.

MidpointVI.**initialize_from_configs**(*t0*, *q0*, *t1*, *q1*, *lambda1=None*)
: Initialize the integrator from two consecutive time and configuration pairs. This calculates p_1 from the two pairs and initializes the integrator with the state (t1, q1, p1).

  lambda1 is optional. See *initialize_from_state()*.

Dynamics

MidpointVI.**step**(*t2*, *u1=tuple()*, *k2=tuple()*, *max_iterations=200*, *lambda1_hint=None*, *q2_hint=None*)
: Step the integrator forward to time t2 . This advances the time and solves the DEL equation. The current state will become the previous state (ie, $t_2 \Rightarrow t_1$, $q_2 \Rightarrow q_1$, $p_2 \Rightarrow p_1$). The solution will be saved as the new state, available through *t2*, *q2*, and *p2*. `lambda` will be updated with the new constraint force, and *u1* will be updated with the value of *u1*.

lambda1 and *q2* can be specified to seed the root solving algorthm. If they are `None`, the previous values will be used.

The method returns the number of root solver iterations needed to find the solution.

Raises a `ConvergenceError` exception if the root solver cannot find a solution after *max_iterations*.

`MidpointVI.`**`calc_f`**`()`
Evaluate the DEL equation at the current states. For dynamically consistent states, this should be zero. Otherwise it is the remainder of the DEL.

The value is returned as a `numpy` array.

Derivatives of q_2

`MidpointVI.`**`q2_dq1`**`(`*q=None*, *q1=None*`)`
`MidpointVI.`**`q2_dp1`**`(`*q=None*, *p1=None*`)`
`MidpointVI.`**`q2_du1`**`(`*q=None*, *u1=None*`)`
`MidpointVI.`**`q2_dk2`**`(`*q=None*, *k2=None*`)`
Calculate the first derivatives of q_2 with respect to the previous configuration, the previous momentum, the input forces, or the kinematic inputs.

If both the parameters are `None`, the entire derivative is returned as a `numpy` array with derivatives across the rows.

If any parameters are specified, they must be appropriate objects. The function will return the specific information requested. For example, *q2_dq1()* will calculate the derivative of the new value of q with respect to the previous value of *q1*.

Calculating the derivative of q_2 with respect to the *i*-th configuration variable $\frac{\partial q_2}{\partial q_1^i}$:

```
>>> q1 = system.configs[i]
>>> mvi.q2_dq1(q1=q1)
array([ 0.133, -0.017,  0.026, ..., -0.103, -0.017,  0.511])
```

Equivalently:

```
>>> mvi.q2_dq1()[:,i]
array([ 0.133, -0.017,  0.026, ..., -0.103, -0.017,  0.511])
```

Calculating the derivative of the *j*-th new configuration with respect to the *i*-th kinematic input:

```
>>> q = system.configs[j]
>>> k2 = system.kin_configs[i]
>>> mvi.q2_dk2(q, k2)
0.023027007157071705

# Or...
>>> mvi.q2_dk2()[j,i]
0.023027007157071705

# Or...
>>> mvi.q2_dk2()[q.index, k2.k_index]
0.023027007157071705
```

`MidpointVI.`**`q2_dq1dq1`**`(`*q=None*, *q1_1=None*, *q1_2=None*`)`
`MidpointVI.`**`q2_dq1dp1`**`(`*q=None*, *q1=None*, *p1=None*`)`
`MidpointVI.`**`q2_dq1du1`**`(`*q=None*, *q1=None*, *u1=None*`)`

MidpointVI.**q2_dq1dk2**(*q=None*, *q1=None*, *k2=None*)
MidpointVI.**q2_dp1dp1**(*q=None*, *p1_1=None*, *p1_2=None*)
MidpointVI.**q2_dp1du1**(*q=None*, *p1=None*, *u1=None*)
MidpointVI.**q2_dp1dk2**(*q=None*, *p1=None*, *k2=None*)
MidpointVI.**q2_du1du1**(*q=None*, *u1_1=None*, *u1_2=None*)
MidpointVI.**q2_du1dk2**(*q=None*, *u1=None*, *k2=None*)
MidpointVI.**q2_dk2dk2**(*q=None*, *k2_1=None*, *k2_2=None*)

Calculate second derivatives of the new configuration.

If no parameters specified, the entire second derivative is returned as a `numpy` array. The returned arrays are indexed with the two derivatives variables as the first two dimensions and the new state as the last dimension.

To calculate the derivative of the *k*-th new configuration with respect to the *i*-th previous configuration and *j* previous momentum:

```
>>> q = system.configs[k]
>>> q1 = system.configs[i]
>>> p1 = system.configs[j]
>>> mvi.q2_dq1dp1(q, q1, p1)
6.4874251262289155e-06

# Or...
>>> result = mvi.q2_dq1dp1(q)
>>> result[i, j]
6.4874251262289155e-06
>>> result[q1.index, p1.index]
6.4874251262289155e-06

# Or....
>>> result = mvi.q2_dq1dp1()
>>> result[i, j, k]
6.4874251262289155e-06
>>> result[q1.index, p1.index, q.index]
6.4874251262289155e-06
```

Derivatives of p_2

MidpointVI.**p2_dq1**(*p=None*, *q1=None*)
MidpointVI.**p2_dp1**(*p=None*, *p1=None*)
MidpointVI.**p2_du1**(*p=None*, *u1=None*)
MidpointVI.**p2_dk2**(*p=None*, *k2=None*)
MidpointVI.**p2_dq1dq1**(*p=None*, *q1_1=None*, *q1_2=None*)
MidpointVI.**p2_dq1dp1**(*p=None*, *q1=None*, *p1=None*)
MidpointVI.**p2_dq1du1**(*p=None*, *q1=None*, *u1=None*)
MidpointVI.**p2_dq1dk2**(*p=None*, *q1=None*, *k2=None*)
MidpointVI.**p2_dp1dp1**(*p=None*, *p1_1=None*, *p1_2=None*)
MidpointVI.**p2_dp1du1**(*p=None*, *p1=None*, *u1=None*)
MidpointVI.**p2_dp1dk2**(*p=None*, *p1=None*, *k2=None*)
MidpointVI.**p2_du1du1**(*p=None*, *u1_1=None*, *u1_2=None*)
MidpointVI.**p2_du1dk2**(*p=None*, *u1=None*, *k2=None*)
MidpointVI.**p2_dk2dk2**(*p=None*, *k2_1=None*, *k2_2=None*)

Calculate the first and second derivatives of p_2. These follow the same conventions as the derivatives of q_2.

Derivatives of λ_1

MidpointVI.**lambda1_dq1**(*constraint=None*, *q1=None*)
MidpointVI.**lambda1_dp1**(*constraint=None*, *p1=None*)
MidpointVI.**lambda1_du1**(*constraint=None*, *u1=None*)
MidpointVI.**lambda1_dk2**(*constraint=None*, *k2=None*)

MidpointVI.**lambda1_dq1dq1**(*constraint=None*, *q1_1=None*, *q1_2=None*)
MidpointVI.**lambda1_dq1dp1**(*constraint=None*, *q1=None*, *p1=None*)
MidpointVI.**lambda1_dq1du1**(*constraint=None*, *q1=None*, *u1=None*)
MidpointVI.**lambda1_dq1dk2**(*constraint=None*, *q1=None*, *k2=None*)
MidpointVI.**lambda1_dp1dp1**(*constraint=None*, *p1_1=None*, *p1_2=None*)
MidpointVI.**lambda1_dp1du1**(*constraint=None*, *p1=None*, *u1=None*)
MidpointVI.**lambda1_dp1dk2**(*constraint=None*, *p1=None*, *k2=None*)
MidpointVI.**lambda1_du1du1**(*constraint=None*, *u1_1=None*, *u1_2=None*)
MidpointVI.**lambda1_du1dk2**(*constraint=None*, *u1=None*, *k2=None*)
MidpointVI.**lambda1_dk2dk2**(*constraint=None*, *k2_1=None*, *k2_2=None*)

Calculate the first and second derivatives of λ_1. These follow the same conventions as the derivatives of q_2. The constraint dimension is ordered according to *System.constraints*.

CHAPTER 2

trep.potentials - Potential Energies

Conservative forces are best modeled in *trep* as potential energies. Every type of potential is derived from *trep.Potential*.

These are the types of potential energy currently built in to *trep*.

Gravity – Basic Gravity

class trep.potentials.**Gravity**(*system*[, *gravity=(0.0, 0.0, -9.8)*, *name=None*])
(*Inherits from* Potential)

Parameters

- **system** (System) – An instance of System to add the gravity to.
- **gravity** (Sequence of three Float) – The gravity vector
- **name** – A string that uniquely identifies the gravity.

Gravity implements a basic constant acceleration gravity ($F = m\vec{g}$).

Table 2.1: **Implemented Calculations**

Calculation	Implemented
V	Y
V_dq	Y
V_dqdq	Y
V_dqdqdq	Y

Examples

Adding gravity to a system is as easy as declaring an instance of *Gravity*:

```
>>> system = build_custom_system()
>>> trep.potentials.Gravity(system)
<Gravity 0.000000 0.000000 -9.800000>
```

The `System` saves a reference to the new *Gravity*, so we do not have to save a refence to prevent it from being garbage collected.

The default gravity points in the negative *Z* direction. We can specify a new gravity vector when we add the gravity. For example, we can make gravity point in the positive *Y* direction:

```
>>> system = build_custom_system()
>>> trep.potentials.Gravity(system, (0, 9.8, 0))
<Gravity 0.000000 9.800000 0.000000>
```

Gravity Objects

`Gravity.`**`gravity`**
The gravity vector for this instance of gravity.

Visualization

`Gravity.`**`opengl_draw`**`()`
Gravity does not draw a visual representation.

`LinearSpring` – Linear spring between two points

LinearSpring creates a spring force between the origins of two coordinate frames in 3D space:

$$x = ||p_1 - p_2||$$
$$V(q) = -k(x - x_0)^2$$

where p_1 and p_2 are the origins of two coordinate frames, k is the spring stiffness and x_0 is the natural length of the spring.

Table 2.2: **Implemented Calculations**

Calculation	Implemented
V	Y
V_dq	Y
V_dqdq	Y
V_dqdqdq	Y

Warning: The current implementation will fail if p_1 equals p_2 and x_0 is nonzero because of a divide by zero problem.

If the two points are equal and x_0 is not zero, there should be a force. But since there is no vector between the two points, the direction of this force is undefined. When the natural length is zero, this problem can be corrected because the force also goes to zero.

Examples

We can create a simple 1D harmonic oscillator using *LinearSpring* with a frame that is free to translate:

```
import trep
from trep import tx,ty,tz,rx,ry,rz

# Create a sytem with one mass that moves in the x direction.
system = trep.System()
system.import_frames([tx('x', mass=1, name='block')])

trep.potentials.LinearSpring(system, 'World', 'block', k=20, x0=1)

# Remember to avoid x = 0 in simulation.
system.get_config('x').q = 0.5
```

The *LinearSpring* works between arbitrary frames, not just frames connected by a translation. Here, we create two pendulums and connect their masses with a spring:

```
import trep
from trep import tx,ty,tz,rx,ry,rz

system = trep.System()
system.import_frames([
    ry('theta1'), [
        tz(2, mass=1, name='pend1')
        ],
    tx(1), [
        ry('theta2'), [
            tz(2, mass=1, name='pend2')
            ]]
    ])

trep.potentials.LinearSpring(system, 'pend1', 'pend2', k=20, x0=0.9)
```

LinearSpring Objects

class `trep.potentials.`**`LinearSpring`**(*system*, *frame1*, *frame2*, *k*[, *x0=0.0*, *name=None*])

Create a new spring between *frame1* and *frame2*. The frames must already exist in the system.

`LinearSpring.`**`frame1`**

The coordinate frame at one end of the spring.

(read only)

`LinearSpring.`**`frame1`**

The coordinate frame at the other end of the spring.

(read only)

`LinearSpring.`**`x0`**

The natural length of the spring.

`LinearSpring.`**`k`**

The spring constant of the spring.

`ConfigSpring` – Linear Spring acting on a configuration variable

Unlike the *`LinearSpring`* which creates a spring force between two points in 3D space, a *`ConfigSpring`* implements a spring force directly on a generalized coordinate:

$$F_q = -k(q - q_0)$$

where k is the spring stiffness and q_0 is the neutral "length" of the spring.

Table 2.3: **Implemented Calculations**

Calculation	Implemented
V	Y
V_dq	Y
V_dqdq	Y
V_dqdqdq	Y

Examples

We can create a simple 1D harmonic oscillator using *`ConfigSpring`* on a translational configuration variable. The same oscillator could be created using a *`LinearSpring`* as well.

```
import trep
from trep import tx,ty,tz,rx,ry,rz

# Create a sytem with one mass that moves in the x direction.
system = trep.System()
system.import_frames([tx('x', mass=1, name='block')])

trep.potentials.ConfigSpring(system, 'x', k=20, q0=0)
```

A *`ConfigSpring`* can be used to create a torsional spring on a rotational configuration variable. Here we create a pendulum with a length of 2 and mass of 1. Instead of gravity, the pendulum is moved by a single torsional spring.

```
import trep
from trep import tx,ty,tz,rx,ry,rz

system = trep.System()
system.import_frames([
    ry("theta"), [
        tz(2, mass=1)
        ]])

trep.potentials.ConfigSpring(system, 'theta', k=20, q0=0.7)
```

ConfigSpring Objects

class `trep.potentials.`**`ConfigSpring`**(*system*, *config*, *k*[, *q0=0.0*, *name=None*])

Create a new spring acting on the specified configuration variable. The configuration variable must already exist in the system.

`ConfigSpring.`**`config`**
The configuration variable that spring depends and acts on.

(read only)

`ConfigSpring.`**`q0`**
The "neutral length" of the spring. When `self.config.q == q0`, the force is zero.

`ConfigSpring.`**`k`**
The spring constant of the spring.

`NonlinearConfigSpring` – Nonlinear spring acting on a configuration variable

Like *`ConfigSpring`* which creates a spring force directly on a generalized coordinate, a *`NonlinearConfigSpring`* implements a force, F_q on a generalized coordinate which can be a function of configuration:

$$F_q = -f(m \cdot q + b)$$

where f is a spline function provided by the user, m is a linear scaling factor, and b is an offset value.

Table 2.4: **Implemented Calculations**

Calculation	Implemented
V	N
V_dq	Y
V_dqdq	Y
V_dqdqdq	Y

NonlinearConfigSpring Objects

class `trep.potentials.`**`NonlinearConfigSpring`**(*system*, *config*, *spline*[, *m=1.0*, *b=0.0*, *name=None*])
Create a new nonlinear spring acting on the specified configuration variable. The configuration variable must already exist in the system.

`NonlinearConfigSpring.`**`config`**
The configuration variable that spring depends and acts on.

(read only)

`NonlinearConfigSpring.`**`spline`**
A `Spline` object relating the configuration to force. See the *`trep.Spline`* documentation to create a spline object.

`NonlinearConfigSpring.`**`m`**
A linear scaling factor on the configuration.

`NonlinearConfigSpring.`**`b`**
An offset factor on the scaled configuration.

CHAPTER 3

trep.forces - Forces

Non-conservative forces are modeled in trep by deriving from the `Force` type. These types of forces include damping and control forces/torques.

These are the types of forces currently built in to *trep*.

`Damping` – Damping on Configuration Variables

Damping implements a damping `Force` on the generalized coordinates of a system:

$$F_{q_i} = -c_i \, \dot{q}_i$$

where the damping constant c_i is a positive real number.

One instance of *Damping* defines damping parameters for every configuration variable in the system. You can specify values for specific configuration variables and have a default value for the other variables.

Table 3.1: **Implemented Calculations**

Calculation	Implemented
f	Y
f_dq	Y
f_ddq	Y
f_du	Y
f_dqdq	Y
f_ddqdq	Y
f_ddqddq	Y
f_dudq	Y
f_duddq	Y
f_dudu	Y

Examples

`damped-pendulum.py`, `extensor-tendon-model.py`, `forced-pendulum-inverse-dynamics.py`, `initial-conditions.py`, `pccd.py`, `pend-on-cart-optimization.py`, `puppet-basic.py`, `puppet-continuous-moving.py`, `puppet-moving.py`, `pyconfig-spring.py`, `pypccd.py`, `radial.py`

class `trep.forces.`**`Damping`**(*system*, *default=0.0*, *coefficients={}*, *name=None*)

Create a new damping force for the system. *default* is the default damping coefficient.

Damping coefficients for specific configuration variables can be specified with *coefficients*. This should be a dictionary mapping configuration variables (or their names or index) to the damping coefficient:

```
trep.forces.Damping(system, 0.1, {'theta' : 1.0})
```

`Damping.`**`default`**

The default damping coefficient for configuration variable.

`Damping.`**`get_damping_coefficient`**(*config*)

Return the damping coefficient for the specified configuration variable. If the configuration variable does not have a set value, the default value is returned.

`Damping.`**`set_damping_coefficient`**(*config*, *coeff*)

Set the damping coefficient for a specific configuration variable. If *coeff* is `None`, the specific coefficient for that variable will be deleted and the default value will be used.

`LinearDamper` – Linear damper between two points

`LinearDamper` creates a damping force between the origins of two coordinate frames in 3D space:

We can derive a mapping from the damper force to a force in the generalized coordinates by looking at the work done by the damper. The virtual work done by a damper is simply the damper force multiplied by the change in the damper's length (distance):

$$x = ||p_1 - p_2||$$
$$f = -c\dot{x}$$
$$F_{q_i} = f\frac{\partial x}{\partial q_i}$$

where p_1 and p_2 are the origins of two coordinate frames and c is the damper coefficient.

Table 3.2: **Implemented Calculations**

Calculation	Implemented
f	Y
f_dq	Y
f_ddq	Y
f_du	Y
f_dqdq	Y
f_ddqdq	Y
f_ddqddq	Y
f_dudq	Y
f_duddq	Y
f_dudu	Y

Examples

`dual_pendulums.py`

class `trep.forces.`**`LinearDamper`**(*system*, *frame1*, *frame2*, *c*[, *name=None*])
Create a new damper between *frame1* and *frame2*. The frames must already exist in the system.

`LinearDamper.`**`frame1`**
The coordinate frame at one end of the damper.

(read only)

`LinearDamper.`**`frame1`**
The coordinate frame at the other end of the damper.

(read only)

`LinearDamper.`**`c`**
The damping coefficient of the damper.

`ConfigForce` – Apply forces to a configuration variable.

`ConfigForce` creates an input variable that applies a force directly to a configuration variable:

$$F_{q_i} = u(t)$$

where the $u(t)$ is a `Input` variable.

Table 3.3: **Implemented Calculations**

Calculation	Implemented
f	Y
f_dq	Y
f_ddq	Y
f_du	Y
f_dqdq	Y
f_ddqdq	Y
f_ddqddq	Y
f_dudq	Y
f_duddq	Y
f_dudu	Y

Examples

`forced-pendulum-inverse-dynamics.py`, `initial-conditions.py`, `pend-on-cart-optimization.py`

class `trep.forces.`**`ConfigForce`**(*system*, *config*, *finput*, *name=None*)

Create a new input to apply a force on a configuration variable.

config should be an existing configuration variable (a name, index, or object).

finput should be a string to name the new input variable.

`ConfigForce.`**`finput`**

The input variable (`Input`) that controls this force.

`ConfigForce.`**`config`**

The configuration variable (`Config`) that this force is applied to.

`BodyWrench` – Apply a body wrench to a frame.

`BodyWrench` applies a fixed or variable wrench to a coordinate frame. The wrench is expressed in the coordinates of the frame:

$$F_{q_i} = \left(g^{-1} \frac{\partial g}{\partial q_i} \right)^{\check{}} \cdot f$$

where g_1 is the coordinate frame the wrench is applied to and f is the wrench. The wench is a vector of six numbers that defined the forces applied to the x, y, and z axes and torques about the x, y, and z axes. In trep, each component of the wrench can be a fixed real number or an input variable (*`trep.Input`*).

Table 3.4: **Implemented Calculations**

Calculation	Implemented
f	Y
f_dq	Y
f_ddq	Y
f_du	Y
f_dqdq	Y
f_ddqdq	Y
f_ddqddq	Y
f_dudq	Y
f_duddq	Y
f_dudu	Y

BodyWrench(system, frame, wrench=tuple(), name=None):
Create a new body wrench force to apply to *frame*.

wrench is a mixed tuple of 6 real numbers and strings. Components that are real numbers will be constant values for the wrench, while components that are strings will be controlled by input variables. An instance of `Input` will be created for each string, with the string defining the input's name.

`BodyWrench.`**wrench**
A mixed tuple of numbers and inputs that define the wrench.

(read-only)

`BodyWrench.`**wrench_val**
A tuple of the current numeric values of the wrench.

`BodyWrench.`**frame**
The frame that this wrench is applied to.

(read-only)

`HybridWrench` – Apply a hybrid wrench to a frame.

`HybridWrench` applies a fixed or variable wrench to a coordinate frame. The wrench is expressed in the world coordinate frame and applied to the origin of the coordinate frame.

The wench is a vector of six numbers that defined the forces applied to the x, y, and z axes and torques about the x, y, and z axes. In trep, each component of the wrench can be a fixed real number or an input variable (*`trep.Input`*).

Table 3.5: **Implemented Calculations**

Calculation	Implemented
f	Y
f_dq	Y
f_ddq	Y
f_du	Y
f_dqdq	Y
f_ddqdq	Y
f_ddqddq	Y
f_dudq	Y
f_duddq	Y
f_dudu	Y

HybridWrench(system, frame, wrench=tuple(), name=None):
Create a new hybrid wrench force to apply to *frame*.

wrench is a mixed tuple of 6 real numbers and strings. Components that are real numbers will be constant values for the wrench, while components that are strings will be controlled by input variables. An instance of `Input` will be created for each string, with the string defining the input's name.

`HybridWrench.`**wrench**
A mixed tuple of numbers and inputs that define the wrench.

(read-only)

`HybridWrench.`**wrench_val**
A tuple of the current numeric values of the wrench.

`HybridWrench.`**frame**
The frame that this wrench is applied to.

(read-only)

`SpatialWrench` – Apply a spatial wrench to a frame.

`SpatialWrench` applies a fixed or variable wrench to a coordinate frame. The wrench is expressed in the world coordinate frame and applied to the location of the world origin in the coordinate frame.

$$F_{q_i} = \left(\frac{\partial g}{\partial q_i} g^{-1}\right)^{\check{}} \cdot f$$

where g_1 is the coordinate frame the wrench is applied to and f is the wrench. The wench is a vector of six numbers that defined the forces applied to the x, y, and z axes and torques about the x, y, and z axes. In trep, each component of the wrench can be a fixed real number or an input variable (*`trep.Input`*).

Table 3.6: **Implemented Calculations**

Calculation	Implemented
f	Y
f_dq	Y
f_ddq	Y
f_du	Y
f_dqdq	Y
f_ddqdq	Y
f_ddqddq	Y
f_dudq	Y
f_duddq	Y
f_dudu	Y

SpatialWrench(system, frame, wrench=tuple(), name=None):
Create a new spatial wrench force to apply to *frame*.

wrench is a mixed tuple of 6 real numbers and strings. Components that are real numbers will be constant values for the wrench, while components that are strings will be controlled by input variables. An instance of `Input` will be created for each string, with the string defining the input's name.

`SpatialWrench.`**wrench**
A mixed tuple of numbers and inputs that define the wrench.

(read-only)

`SpatialWrench.`**wrench_val**
A tuple of the current numeric values of the wrench.

`SpatialWrench.`**frame**
The frame that this wrench is applied to.

(read-only)

CHAPTER 4

trep.constraints - Holonomic Constraints

This module contains the built in types of holonomic constraints.

These are the types of constraints currently built in to *trep*. Additional constraints can be added through either the Python or C-API.

Distance - Maintain a specific distance between points

The *Distance* constraint maintains a specific distance between the origins of two coordinate frames. The distance can be a fixed number or controlled by a kinematic configuration variable.

The constraint equation is:

$$d^2 = (p_1 - p_2)^T (p_1 - p_2)$$

where p_1 and p_2 are the origins of two coordinate frames and d is the desired distance between them.

Warning: This constraint is undefined when $d = 0$. If you want to constraint two points to be coincident, you can use the `PointToPoint` constraint instead.

Examples

`puppet-basic.py`, `puppet-continuous-moving.py`, `puppet-moving.py`

class `trep.constraints.`**`Distance`**(*system*, *frame1*, *frame2*, *distance*, *name=None*)

Create a new constraint to maintain the distance between *frame1* and *frame2*.

frame1 and *frame2* should be existing coordinate frames in *system*. They can be the `Frame` objects themselves or their names.

distance can be a real number or a string. If it is a string, the constraint will create a new kinematic configuration variable with that name.

Distance.**config**
This is the kinematic configuration variable that controls the distance, or None for a fixed distance constraint.

(read-only)

Distance.**frame1**
Distance.**frame2**
These are the two Frame objects being constrained.

(read-only)

Distance.**distance**
This is the *desired* constraint between the two coordinate frames. This is either the fixed value or the value of the configuration variable. If you set the distance, the appropriate value will be updated.

Distance.**get_actual_distance**()
Calculate the current distance between the two coordinate frames. If the constraint is currently satisfied, this is equal to *distance*.

If you have the system in a configuration you like, you can use this to set the correct distance:

```
>>> constraint.distance = constraint.get_actual_distance()
```

PointToPoint - Constrain the origins of two frames together

The PointToPoint constraints are designed to constrain the origin of one frame to the origin of another frame (maintaining zero distance between the frames).

The constraint equation is:

$$h(q) = (p_1 - p_2)[i]$$

where p_1 and p_2 are the origin points of the two frames being constrained, and i is the basis component of the error vector.

Note: Defining a PointToPoint constraint creates multiple one-dimensional constraints for each axis being used. For example, if a system is defined in 3D, the PointToPoint3D method creates 3 PointToPoint constraints, one for each axis.

Examples

scissor.py

---

class trep.constraints.**PointToPoint3D**(*system*, *frame1*, *frame2*, *name=None*)

Parameters

- **frame1** (Frame) – First frame to constrain
- **frame2** (Frame) – Second frame to constrain

Create a new set of 3 constraints to force the origin of *frame1* to the origin of *frame2*. The system must be defined in 3D, otherwise, for 2D systems, use *PointToPoint2D*.

class trep.constraints.**PointToPoint2D**(*system*, *plane*, *frame1*, *frame2*, *name=None*)

Parameters

- **plane** (string) – 2D plane of system, ie. 'xy', 'xz', or 'yz'
- **frame1** (Frame) – First frame to constrain
- **frame2** (Frame) – Second frame to constrain

Create a new set of 2 constraints to force the origin of *frame1* to the origin of *frame2*. The system must be defined in 2D, otherwise, for 3D systems, use *PointToPoint3D*.

class trep.constraints.**PointToPoint1D**(*system*, *axis*, *frame1*, *frame2*, *name=None*)

Parameters

- **axis** (string) – 1D axis of system, ie. 'x', 'y', or 'z'
- **frame1** (Frame) – First frame to constrain
- **frame2** (Frame) – Second frame to constrain

Create a new constraint to force the origin of *frame1* to the origin of *frame2*. The system must be defined in 1D, otherwise, for 2D or 3D systems, use *PointToPoint2D* or *PointToPoint3D*.

PointToPoint3D.**get_actual_distance**()
PointToPoint2D.**get_actual_distance**()
PointToPoint1D.**get_actual_distance**()

Calculate the current distance between the two coordinate frames. If the constraint is currently satisfied, this is equal to 0.

PointOnPlane - Constraint a point to a plane

The *PointOnPlane* constraint constraints a point (defined by the origin of a coordinate frame) to lie in a plane (defined by the origin of another coordinate frame and normal vector).

The constraint equation is:

$$h(q) = (p_1 - p_2) \cdot (g_1 \cdot norm)$$

where g_1 and p_1 are the transformation of the frame defining the plane and its origin, p_2 is the point being constrained, and $norm$ is the normal vector of the plane expressed in the local coordinates of g_1.

By defining two of *PointOnPlane* constraints with normal plane, you can constrain a point to a line. With three constraints, you can constraint a point to another point.

Examples

pccd.py, pypccd.py, radial.py

---

class trep.constraints.**PointOnPlane**(*system*, *plane_frame*, *plane_normal*, *point_frame*, *name=None*)

Create a new constraint to force the origin of *point_frame* to lie in a plane. The plane is coincident with the origin of *plane_frame* and normal to the vector *plane_normal*. The normal is expresed in the coordinates of *plane_frame*.

PointOnPlane.**plane_frame**

This is the coordinate frame the defines the plane.

(read-only)

PointOnPlane.**normal**

This is the normal of the plane expressed in *plane_frame* coordinates.

PointOnPlane.**point_frame**
This is the coordinate frame that defines the point to be constrained.

CHAPTER 5

`trep.discopt` - Discrete Optimal Control

This module provides tools for using discrete optimal control techniques.

Discrete LQ Problems

The *`trep.discopt`* module provides functions for solving time-varying discrete LQ problems.

The Linear Quadratic Regulator (LQR) Problem

The LQR problem is to find the input for a linear system that minimizes a quadratic cost. The optimal input turns out to be a feedback law that is independent of the system's initial condition. Because of this, the LQR problem is a useful tool to automatically calculate a stabilizing feedback controller for a dynamic system. For nonlinear systems, the LQR problem is solved for the linearization of the system about a trajectory to get a locally stabilizing controller.

Problem Statement: Given a discrete linear system Find the control input $u(k)$ that minimizes a quadratic cost:

$$V(x(k_0), u(\cdot), k_0) = \sum_{k=k_0}^{k_f-1} \left[x^T(k)Q(k)x(k) + u^T(k)R(k)u(k)\right] + x^T(k_f)Q(k_f)x(k_f)$$

where

$$R(k) = R^T(k) \geq 0 \; \forall \; k \in \{k_0 \ldots (k_f - 1)\} \tag{5.1}$$
$$Q(k) = Q^T(k) \geq 0 \; \forall \; k \in \{k_0 \ldots (k_f - 1)\} \tag{5.2}$$
$$x(k_0) \text{ is known} \tag{5.3}$$
$$x(k+1) = A(k)x(k) + B(k)u(k) \tag{5.4}$$

Solution: The optimal control $u^*(k)$ and optimal cost $V^*(x(k_0), k_0)$ are

$$u^*(k) = -\mathcal{K}(k)x(k) \tag{5.5}$$
$$V^*(x(k_0), k_0) = x^T(k_0)P(k_0)x(k_0) \tag{5.6}$$

where

$$\mathcal{K}(k) = \Gamma^{-1}(k)B^T(k)P(k+1)A(k)$$
$$\Gamma(k) = R(k) + B^T(k)P(k+1)B(k)$$

and $P(k+1)$ is a symmetric time varying matrix satisfying a discrete Ricatti-like equation:

$$P(k_f) = Q(k_f) \tag{5.7}$$
$$P(k) = Q(k) + A^T(k)P(k+1)A(k) - \mathcal{K}^T(k)\Gamma(k)\mathcal{K}(k) \tag{5.8}$$

trep.discopt.**solve_tv_lqr**(*A*, *B*, *Q*, *R*)

Parameters

- **A** (*Sequence of N numpy arrays, shape (nX, nX)*) – Linear system dynamics
- **B** (*Sequence of N numpy arrays, shape (nX, nU)*) – Linear system input matrix
- **Q** (*Function Q(k) returning numpy array, shape (nX, nX)*) – Quadratic State Cost
- **R** (*Function R(k) returning numpy array, shape (nU, nU)*) – Quadratic Input Cost

Return type named tuple (K, P)

This function solve the time-varying discrete LQR problem for the linear system *A*, *B* and costs *Q* and *R*.

A is a sequence of the linear system dynamics, `A[k]`.

B is a sequence of the linear system's input matrix, `B[k]`.

Q is a function `Q(k)` that returns the state cost matrix at time *k*. For example, if $Q(k) = \mathcal{I}$:

```
Q = lambda k: numpy.eye(nX)
```

R is a function `Q(k)` that returns the state cost matrix at time *k*. For example, if the cost matrices are stored in an array *r_costs*:

```
R = lambda k: r_costs[k]
```

The function returns the optimal feedback law $\mathcal{K}(\|)$ and the solution to the discrete Ricatti equation at k=0, $P(0)$. *K* is a sequence of N numpy arrays of shape (nU,nX). *P* is a single (nX, nX) numpy array.

The Linear Quadratic (LQ) Problem

The LQ problem is to find the input for a linear system that minimizes a cost with linear and quadratic terms. In trep, the LQ problem is a sub-problem for discrete trajectory optimization that is used to calculate the descent direction at each iteration.

Problem Statement: Find the control input $u(k)$ that minimizes the cost:

$$V(x(k_0), u(\cdot), k_0) = \sum_{k=k_0}^{k_f-1} \left[2 \begin{bmatrix} q(k) \\ r(k) \end{bmatrix}^T \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} + \begin{bmatrix} x(k) \\ u(k) \end{bmatrix}^T \begin{bmatrix} Q(k) & S(k) \\ S^T(k) & R(k) \end{bmatrix} \begin{bmatrix} x(k) \\ u(k) \end{bmatrix} \right]$$
$$+2q^T(k_f)x(k_f) + x^T(k_f)Q(k_f)x(k_f)$$

where

$$\begin{aligned} R(k) = R^T(k) > 0 \; \forall \; k \in \{k_0 \dots (k_f - 1)\} \\ Q(k) = Q^T(k) \geq 0 \; \forall \; k \in \{k_0 \dots k_f\} \\ x(k_0) \text{ is known.} \\ x(k+1) = A(k)x(k) + B(k)u(k) \end{aligned}$$

Solution: The optimal control $u^*(k)$ and optimal cost $V^*(x(k_0), k_0)$ are:

$$\begin{aligned} u^*(k) &= -\mathcal{K}(k)x(k) - C(k) \\ V^*(x(k_0), k_0) &= x^T(k_0)P(k_0)x(k_0) + 2b^T(k_0)x(k_0) + c(k_0) \end{aligned}$$

where:

$$\begin{aligned} K(k) &= \Gamma^{-1}(k)\left[B^T(k)P(k+1)A(k) + S^T(k)\right] \\ C(k) &= \Gamma^{-1}(k)\left[B^T(k)b(k+1) + r(k)\right] \\ \Gamma(k) &= \left[R(k) + B^T(k)P(k+1)B(k)\right] \end{aligned}$$

and $P(k)$, $b(k)$, and $c(k)$ are solutions to backwards difference equations:

$$\begin{aligned} P(k_f) &= Q(k_f) \\ P(k) &= Q(k) + A^T(k)P(k+1)A(k) - \mathcal{K}^T(k)\Gamma(k)\mathcal{K}(k) \end{aligned}$$

$$\begin{aligned} b(k_f) &= q(k_f) \\ b(k) &= \left[A^T(k) - \mathcal{K}^T(k)B^T(k)\right]b(k+1) + q(k) - \mathcal{K}^T(k)r(k) \end{aligned}$$

$$\begin{aligned} c(k_f) &= 0 \\ c(k) &= c(k+1) - C(k)^T\Gamma(k)C(k) \end{aligned}$$

trep.discopt.**solve_tv_lq**(*A*, *B*, *q*, *r*, *Q*, *S*, *R*)

Parameters

- **A** (*Sequence of N numpy arrays, shape (nX, nX)*) – Linear system dynamics
- **B** (*Sequence of N numpy arrays, shape (nX, nU)*) – Linear system input matrix
- **q** (*Sequence of N numpy arrays, shape (nX)*) – Linear State Cost
- **r** (*Sequence of N numpy arrays, shape (nU)*) – Linear Input Cost
- **Q** (*Function Q(k) returning numpy array, shape (nX, nX)*) – Quadratic State Cost
- **S** (*Function S(k) returning numpy array, shape (nX, nU)*) – Quadratic Cross Term Cost
- **R** (*Function R(k) returning numpy array, shape (nU, nU)*) – Quadratic Input Cost

Return type named tuple (K, C, P, *b*)

This function solve the time-varying discrete LQ problem for the linear system *A*, *B*.

A[k] is a sequence of the linear system dynamics, $A(k)$.

B[k] is a sequence of the linear system's input matrix, $B(k)$.

q[k] is a sequence of the linear state cost, $q(k)$.

r[k] is a sequence of the linear input cost, $r(k)$.

Q(k) is a function that returns the quadratic state cost matrix *at time k*. For example, if $Q(k) = \mathcal{I}$:

```
Q = lambda k: numpy.eye(nX)
```

S(k) is a function that returns the quadratic cross term cost *matrix at time k*.

R(k) is a function that returns the state cost matrix at time *k*. For example, if the cost matrices are stored in an array *r_costs*:

```
R = lambda k: r_costs[k]
```

The function returns the optimal feedback law $\mathcal{K}(\|)$, the affine input term $C(k)$, and the last solution to two of the difference equations, $P(0)$ and $b(0)$.

K is a sequence of N numpy arrays of shape (nU,nX).

C is a sequence of N numpy arrays of shape (nU).

P is a single (nX, nX) numpy array.

b is a single (nX) numpy array.

`DSystem` - Discrete System wrapper for Midpoint Variational Integrators

DSystem objects represent a `MidpointVI` variational integrators as first order discrete systems of the form $X(k+1) = f(X(k), U(k), k)$. This representation is used by *DOptimizer* for discrete trajectory optimization. *DSystem* also provides methods for automatically calculating linearizations and feedback controllers along trajectories.

The discrete state consists of the variational integrator's full configuration, the dynamic momentum, and the kinematic velocity:

$$X(k) = \begin{bmatrix} q_d(k) \\ q_k(k) \\ p(k) \\ v_k(k) \end{bmatrix}$$

The configuration and momentum are the same as in `MidpointVI`. The kinematic velocity is calculated as:

$$v_k(k) = \frac{q_k(k) - q_k(k-1)}{t(k) - t(k-1)}$$

The discrete input consists of the variational integrator's force inputs and the future state of the kinematic configurations:

$$U(k) = \begin{bmatrix} u(k) \\ \rho(k) \end{bmatrix}$$

where the kinematic inputs are denoted by ρ thoughout this code (i.e, $q_k(k+1) = \rho(k)$). Additionally, the state input *U* is always capitalized to distinguish it from the force input *u* which is always lower case.

DSystem provides methods for converting between trajectories for the discrete system and trajectories for the variational integrator.

Examples

`pend-on-cart-optimization.py`

DSystem Objects

class `trep.discopt.`**`DSystem`**(*varint*, *t*)

Parameters

- **`varint`** (`MidpointVI` instance) – The variational integrator being represented.
- **`t`** (*numpy array of floats, shape (N)*) – An array of times

Create a discrete system wrapper for the variational integrator *varint* and the time *t*. The time *t* is the array $t(k)$ that maps a discrete time index to a time. It should have the same length *N* as trajectories used with the system.

`DSystem.`**`nX`**
Number of states to the discrete system.

int

`DSystem.`**`nU`**
Number of inputs to the discrete system.

int

`DSystem.`**`varint`**
The variational integrator wrapped by this instance.

`MidpointVI`

`DSystem.`**`system`**
The mechanical system modeled by the variational integrator.

`System`

`DSystem.`**`time`**
The time of the discrete steps.

numpy array, shape (N)

`trep.discopt.`**`xk`**
Current state of the system.

numpy array, shape (nX)

`trep.discopt.`**`uk`**
Current input of the system.

numpy array, shape (nU)

trep.discopt.**k**
Current discrete time of the system.

int

DSystem.**kf**()

Return type int

Return the last available state that the system can be set to. This is one less than `len(self.time)`.

State and Trajectory Manipulation

DSystem.**build_state**([*Q=None*, *p=None*, *v=None*])

Parameters

- **Q** (*numpy array, shape (nQ)*) – A configuration vector
- **p** (*numpy array, shape (nd)*) – A momentum vector
- **v** (*numpy array, shape (nk)*) – A kinematic velocity vector

Build a state vector *Xk* from components. Unspecified components are set to zero.

DSystem.**build_input**([*u=None*, *rho=None*])

Parameters

- **u** (*numpy array, shape (nu)*) – An input force vector
- **rho** (*numpy array, shape (nk)*) – A kinematic input vector

Type numpy array, shape (nU)

Build an input vector *Uk* from components. Unspecified components are set to zero.

DSystem.**build_trajectory**([*Q=None*, *p=None*, *v=None*, *u=None*, *rho=None*])

Parameters

- **Q** (*numpy array, shape (N, nQ)*) – A configuration trajectory
- **p** (*numpy array, shape (N, nd)*) – A momentum trajectory
- **v** (*numpy array, shape (N, nk)*) – A velocity trajectory
- **u** (*numpy array, shape (N-1, nu)*) – An input force trajectory
- **rho** (*numpy array, shape (N-1, nk)*) – A kinematic input trajectory

Return type named tuple of (*X*, *U*)

Combine component trajectories into a state and input trajectories. The state length is the same as the time base, the input length is one less than the time base. Unspecified components are set to zero:

```
>>> dsys.build_trajectory() # Create a zero state and input trajectory
```

DSystem.**split_state**([*X=None*])

Parameters **X** (*numpy array, shape (nX)*) – A state vector for the system

Return type named tuple of (*Q*, *p*, *v*)

Split a state vector into its configuration, moementum, and kinematic velocity parts. If *X* is `None`, returns zero arrays for each component.

DSystem.**split_input**([*U=None*])

Parameters **U** (*numpy array, shape (nU)*) – An input vector for the system

Return type named tuple of (*u*, *rho*)

Split a state input vector *U* into its force and kinematic input parts, (*u*, *rho*). If *U* is `None`, returns zero arrays of the appropriate size.

DSystem.**split_trajectory**([*X=None*, *U=None*])

Parameters

- **X** (*numpy array, shape (N, nX)*) – A state trajectory
- **U** (*numpy array, shape (N-1, nU)*) – An input trajectory

Return type named tuple of (*Q*, *p*, *v*, *u*, *rho*)

Split the state trajectory (*X*, *U*) into its *Q*, *p*, *v*, *u*, *rho* components. If *X* or *U* are `None`, the corresponding components are arrays of zero.

DSystem.**convert_trajectory**(*dsys_a*, *X*, *U*)

Parameters

- **dsys_a** (*DSystem*) – Another discrete system
- **X** (*numpy array, shape (N, nX)*) – A state trajectory for *dsys_a*
- **U** (*numpy array, shape (N, nU)*) – An input trajectory for *dsys_a*

Return type trajectory for this system, named tuple (*X*, *U*)

Convert the trajectory (*X*, *U*) for *dsys_a* into a trajectory for this system. This reorders the trajectory components according to the configuration and input variable names and drops components that aren't in this system. Variables in this system that are not in *dsys_a* are replaced with zero.

Note: The returned path may not be a valid trajectory for this system in the sense that $x(k+1) = f(x(k), u(k), k)$. This function only reorders the information.

DSystem.**save_state_trajectory**(*filename*[, *X=None*, *U=None*])

Parameters

- **filename** (*string*) – Location to save the trajectory
- **X** (*numpy array, shape (N, nX)*) – A state trajectory
- **U** (*numpy array, shape (N-1, nU)*) – An input trajectory

Save a trajectory to a file. This splits the trajectory with *split_trajectory()* and saves the results with `trep.save_traejctory()`. If *X* or *U* are not specified, they are replaced with zero arrays.

DSystem.**load_state_trajectory**(*filename*)

Parameters **filename** (*string*) – Location of saved trajectory

Return type named tuple of (X, U)

Load a trajectory from a file that was stored with *save_state_trajectory()* or `trep.save_trajectory()`.

If the file does not contain complete information for the system (e.g, it was saved for a different system with different states, or the inputs were not saved), the missing components will be filled with zeros.

Dynamics

DSystem.**set**(*self*, *xk*, *uk*, *k*[, *xk_hint=None*, *lambda_hint=None*])
Set the current state, input, and time of the discrete system.

If *xk_hint* and *lambda_hint* are provided, these are used to provide hints to hints to `MidpointVI.step()`. If the solution is known (for example, if you are calculating the linearization about a known trajectory) this can result in faster performance by reducing the number of root solver iterations in the variational integrator.

DSystem.**step**(*self*, *uk*[, *xk_hint=None*, *lambda_hint=None*])
Advance the system to the next discrete time using the given input *uk*.

This is equivalent to calling `self.set(self.f(), uk, self.k+1)`.

If *xk_hint* and *lambda_hint* are provided, these are used to provide hints to hints to `MidpointVI.step()`. If the solution is known (for example, if you are calculating the linearization about a known trajectory) this can result in faster performance by reducing the number of root solver iterations in the variational integrator.

DSystem.**f**()

Return type numpy array, shape (*nX*)

Get the next state of the system, $x(k+1)$.

First Derivatives

DSystem.**fdx**()

Return type numpy array, shape (*nX*, *nX*)

DSystem.**fdu**()

Return type numpy array, shape (*nX*, *nU*)

These functions return first derivatives of the system dynamics $f()$ as numpy arrays with the derivatives across the rows.

Second Derivatives

DSystem.**fdxdx**(*z*)

Parameters **z** (*numpy array, shape (nX)*) – adjoint vector

Return type numpy array, shape (*nU*, *nU*)

DSystem.**fdxdu**(*z*)

Parameters **z** (*numpy array, shape (nX)*) – adjoint vector

Return type numpy array, shape (*nX*, *nU*)

DSystem.**fdudu**(*z*)

Parameters **z** (*numpy array, shape (nX)*) – adjoint vector

Return type numpy array, shape (*nU*, *nU*)

These functions return the product of the 1D array z and the second derivative of f. For example:

$$z^T \frac{\partial^2 f}{\partial u \partial u}$$

Linearization and Feedback Controllers

DSystem.**linearize_trajectory**(*X*, *U*)

Return type named tuple (A, B)

Calculate the linearization of the system dynamics about a trajectory. *X* and *U* do not have to be an exact trajectory of the system.

Returns the linearization in a named tuple `(A, B)`.

DSystem.**project**(*bX*, *bU*[, *Kproj=None*])

Rtyple named tuple (X, U)

Project *bX* and *bU* into a nearby trajectory for the system using a linear feedback law:

```
X[0] = bX[0]
U[k] = bU[k] - Kproj * (X[k] - bU[k])
X[k+1] = f(X[k], U[k], k)
```

If no feedback law is specified, one will be created by *calc_feedback_controller()* along *bX* and *bU*. This is typically a **bad idea** if *bX* and *bU* are not very close to an actual trajectory for the system.

Returns the projected trajectory in a named tuple `(X, U)`.

DSystem.**dproject**(*A*, *B*, *bdX*, *bdU*, *K*)

Rtyple named tuple (dX, dU)

Project *bdX* and *bdU* into the tangent trajectory space of the system. *A* and *B* are the linearization of the system about the trajectory. *K* is a stabilizing feedback controller.

Returns the projected tangent trajectory `(dX, dU)`.

DSystem.**calc_feedback_controller**(*X*, *U*[, *Q=None*, *R=None*, *return_linearization=False*])

Return type K or named tuple (K, A, B)

Calculate a stabilizing feedback controller for the system about a trajectory *X* and *U*. The feedback law is calculated by solving the discrete LQR problem for the linearization of the system about *X* and *U*.

X and *U* do not have to be an exact trajectory of the system, but if they are not close, the controller is unlikely to be effective.

If the LQR weights *Q* and *R* are not specified, identity matrices are used.

If *return_linearization* is `False`, the return value is the feedback control law, *K*.

If *return_linearization* is `True`, the method returns the linearization as well in a named tuple: `(K, A, B)`.

Checking the Derivatives

DSystem.**check_fdx**(*xk*, *uk*, *k*[, *delta=1e-5*])
DSystem.**check_fdu**(*xk*, *uk*, *k*[, *delta=1e-5*])
DSystem.**check_fdxdx**(*xk*, *uk*, *k*[, *delta=1e-5*])
DSystem.**check_fdxdu**(*xk*, *uk*, *k*[, *delta=1e-5*])
DSystem.**check_fdudu**(*xk*, *uk*, *k*[, *delta=1e-5*])

Parameters

- **xk** (*numpy array, shape (nX)*) – A valid state of the system
- **uk** (*numpy array, shape (nU)*) – A valid input to the system

- **k** (*int*) – A valid discrete time index
- **delta** – The perturbation for approximating the derivative.

These functions check derivatives of the discrete state dynamics against numeric approximations generated from lower derivatives (e.g, fdx() from f(), and fdudu() from fdu()). A three point approximation is used:

```
approx_deriv = (f(x + delta) - f(x - delta)) / (2 * delta)
```

Each function returns a named tuple `(error, exact_norm, approx_norm)` where `error` is the norm of the difference between the exact and approximate derivative, `exact_norm` is the norm of the exact derivative, `approx_norm` is the norm of the approximate derivative.

DCost - Discrete Trajectory Cost

The *DCost* class defines the incremental and terminal costs of a trajectory during a discrete trajectory optimization. It is used in conjunction with *DSystem* and *DOptimizer*.

The discrete trajectory optimization finds a trajectory that minimizes a cost of the form:

$$h(\xi) = \sum_{k=0}^{k_f-1} \ell(x(k), u(k), k) + m(x(k_f))$$

DCost defines the costs $\ell(x, u, k)$ and $m(x)$ for a system and calculates their 1st and 2nd derivatives.

The current implementation defines a suitable cost for tracking a desired trajectory:

$$\ell(x, u, k) = \frac{1}{2}\left((x - x_d(k))^T Q (x - x_d(k)) + (x - u_d(k))^T R (u - u_d(k))\right)$$

$$m(x) = \frac{1}{2}(x - x_d(k_f))^T Q (x - x_d(k_f))$$

where $x_d(k)$ and $u_d(k)$ are the desired state and input trajectories and Q and R are positive definite matrices that define their weighting.

DCost Objects

class `trep.discopt.`**`DCost`**(*xd*, *ud*, *Q*, *R*)

Parameters

- **xd** (*numpy array, shape (N, nX)*) – The desired state trajectory
- **ud** (*numpy array, shape (N-1, nU)*) – The desired input trajectory
- **Q** (*numpy array, shape (nX, nX)*) – Cost weights for the states
- **R** (*numpy array, shape (nU, nU)*) – Cost weights for the inputs

Create a new cost object for the desired states *xd* weighted by *Q* and the desired inputs *ud* weighted by *R*.

`DCost.`**`Q`**
(numpy array, shape (nX, nX))

The weights of the states.

`DCost.`**`R`**
(numpy array, shape (nU, nU))

The weights of the inputs.

Costs

DCost.**l**(*xk*, *uk*, *k*)

Parameters

- **xk** (*numpy array, shape (nX)*) – Current state
- **uk** (*numpy array, shape (nU)*) – Current input
- **k** (*int*) – Current discrete time

Return type float

Calculate the incremental cost of *xk* and *uk* at discrete time *k*.

DCost.**m**(*xkf*)

Parameters **xkf** (*numpy array, shape (nX)*) – Final state

Return type float

Calculate the terminal cost of *xk*.

1st Derivatives

DCost.**l_dx**(*xk*, *uk*, *k*)

Parameters

- **xk** (*numpy array, shape (nX)*) – Current state
- **uk** (*numpy array, shape (nU)*) – Current input
- **k** (*int*) – Current discrete time

Return type numpy array, shape (*nX*)

Calculate the derivative of the incremental cost with respect to the state.

DCost.**l_du**(*xk*, *uk*, *k*)

Parameters

- **xk** (*numpy array, shape (nX)*) – Current state
- **uk** (*numpy array, shape (nU)*) – Current input
- **k** (*int*) – Current discrete time

Return type numpy array, shape (*nU*)

Calculate the derivative of the incremental cost with respect to the input.

DCost.**m_dx**(*xkf*)

Parameters **xkf** (*numpy array, shape (nX)*) – Current state

Return type numpy array, shape (*nX*)

Calculate the derivative of the terminal cost with respect to the final state.

2nd Derivatives

DCost.**l_dxdx**(*xk*, *uk*, *k*)

Parameters

- **xk** (*numpy array, shape (nX)*) – Current state
- **uk** (*numpy array, shape (nU)*) – Current input
- **k** (*int*) – Current discrete time

Return type numpy array, shape (*nX*, *nX*)

Calculate the second derivative of the incremental cost with respect to the state. For this implementation, this is always equal to *Q*.

DCost.**l_dudu**(*xk*, *uk*, *k*)

Parameters

- **xk** (*numpy array, shape (nX)*) – Current state
- **uk** (*numpy array, shape (nU)*) – Current input
- **k** (*int*) – Current discrete time

Return type numpy array, shape (*nU*, *nU*)

Calculate the second derivative of the incremental cost with respect to the inputs. For this implementation, this is always equal to *R*.

DCost.**l_dxdu**(*xk*, *uk*, *k*)

Parameters

- **xk** (*numpy array, shape (nX)*) – Current state
- **uk** (*numpy array, shape (nU)*) – Current input
- **k** (*int*) – Current discrete time

Return type numpy array, shape (*nX*, *nU*)

Calculate the second derivative of the incremental cost with respect to the state and inputs. For this implementation, this is always equal to zero.

DCost.**m_dxdx**(*xkf*)

Parameters **xkf** (*numpy array, shape (nX)*) – Current state

Return type numpy array, shape (*nX*, *nX*)

Calculate the second derivative of the terminal cost. For this implementation, this is always equal to *Q*.

DOptimizer - Discrete Trajectory Optimization

Discrete trajectory optimization is performed with `DOptimizer` objects. The optimizer finds a trajectory for a `DSystem` that minimizes a `DCost`.

The optimizer should work for arbitrary systems, not just ones based on variational integrators. You would need to define a compatible `DSystem` class that supports the right functionality.

Examples

`pend-on-cart-optimization.py`

DOptimizer Objects

class `trep.discopt.`**`DOptimizer`**(*dsys*, *cost*, *first_method_iterations=10*, *monitor=None*)

You should create a new optimizer for each new system or cost, but for a given combination you can optimize as many different trajectories as you want. The optimization is designed to mostly be used through the `optimize()` method.

You can use the `DOptimizerMonitor` class to monitor progress during optimizations. If *monitor* is `None`, the optimizer will use `DOptimizerDefaultMonitor` which prints useful information to the console.

`DOptimizer.`**`dsys`**

The `DSystem` that is optimized by this instance.

`DOptimizer.cost = cost`

The `DCost` that is optimized by this instance.

`DOptimizer.`**`optimize_ic`**

This is a Boolean indicating whether or not the optimizer should optimize the initial condition during an optimization.

Defaults to `False`.

> **Warning:** This is broken for constrained systems.

`DOptimizer.`**`monitor`**

The `DOptimizerMonitor` that this optimization reports progress to. The default is a new instance of `DOptimizerDefaultMonitor`.

`DOptimizer.`**`Qproj`**
`DOptimizer.`**`Rproj`**

These are the weights for an LQR problem used to generate a linear feedback controller for each trajectory. Each should be a function of k that returns an appropriately sized 2D `ndarray`.

The default values are identity matrices.

DOptimizer.**descent_tolerance**
A trajectory is considered a local minimizer if the norm of the cost derivative is less than this. The default value is 1e-6.

Cost Functions

DOptimizer.**calc_cost**(*X*, *U*)
Calculate the cost of a trajectory *X,U*.

DOptimizer.**calc_dcost**(*X*, *U*, *dX*, *dU*)
Calculate the derivative of the cost function evaluated at *X,U* in the direction of a tangent trajectory *dX,dU*.

It is important that *dX,dU* be an actual tangent trajectory of the system at *X,U* to get the correct cost. See *check_ddcost()* for an example where this is important.

DOptimizer.**calc_ddcost**(*X*, *U*, *dX*, *dU*, *Q*, *R*, *S*)
Calculate the second derivative of the cost function evaluated at *X,U* in the direction of a tangent trajectory *dX,dU*. The second order model parameters must be specified in *Q,R,S*. These can be obtained through *calc_newton_model()* or by *calc_descent_direction()* when *method*="newton".

It is important that *dX,dU* be an actual tangent trajectory of the system at *X,U* to get the correct cost. See *check_ddcost()* for an example where this is important.

Descent Directions

DOptimizer.**calc_steepest_model**()
Calculate a quadratic model to find a steepest descent direction:

$$Q = \mathcal{I} \quad R = \mathcal{I} \quad S = 0$$

DOptimizer.**calc_quasi_model**(*X*, *U*)
Calculate a quadratic model to find a quasi-newton descent direction. This uses the second derivative of the un-projected cost function.

$$Q = \frac{\partial^2 h}{\partial x \partial x} \quad R = \frac{\partial^2 h}{\partial u \partial u} \quad S = \frac{\partial^2 h}{\partial x \partial u}$$

This method does not use the second derivative of the system dynamics, so it tends to be as fast *calc_steepest_model()*, but usually converges much faster.

DOptimizer.**calc_newton_model**(*X*, *U*, *A*, *B*, *K*)
Calculate a quadratic model to find a newton descent direction. This is the true second derivative of the projected cost function:

$$\begin{aligned} Q(k_f) &= D^2 m(x(k_f)) \\ Q(k) &= \frac{\partial^2 \ell}{\partial x \partial x}(k) + z^T(k+1)\frac{\partial^2 f}{\partial x \partial x}(k) \\ S(k) &= \frac{\partial^2 \ell}{\partial x \partial u}(k) + z^T(k+1)\frac{\partial^2 f}{\partial x \partial u}(k) \\ R(k) &= \frac{\partial^2 \ell}{\partial u \partial u}(k) + z^T(k+1)\frac{\partial^2 f}{\partial u \partial u}(k) \end{aligned}$$

where:

$$\begin{aligned} z(k_f) &= Dm^T(x(k_f)) \\ z(k) &= \frac{\partial \ell}{\partial x}^T(k) - \mathcal{K}^T(k)\frac{\partial \ell}{\partial u}^T(k) + \left[\frac{\partial f}{\partial x}^T(k) - \mathcal{K}^T(k)\frac{\partial f}{\partial u}^T(i)\right] z(k+1) \end{aligned}$$

This method depends on the second derivative of the system's dynamics, so it can be significantly slower than other step methods. However, it converges extremely quickly near the minimizer.

DOptimizer.**calc_descent_direction**(*X*, *U*, *method='steepest'*)
Calculate the descent direction from the trajectory *X,U* using the specified method. Valid methods are:

- "steepest" - Use *calc_steepest_model()*
- "quasi" - Use *calc_quasi_model()*
- "newton" - Use *calc_newton_model()*

The method returns the named tuple `(Kproj, dX, dU, Q, R, S)`.

Armijo Line Search

DOptimizer.**armijo_beta**
DOptimizer.**armijo_alpha**
DOptimizer.**armijo_max_iterations**
Parameters for the Armijo line search performed at each step along the calculated descent direction.

armijo_beta should be between 0 and 1 (not inclusive). The default value is 0.7.

armijo_alpha should be between 0 (inclusive) and 1 (not inclusive). The default value is 0.00001.

armijo_max_iterations should be a positive integer. If the line search cannot satisfy the sufficient decrease criteria after this number of iterations, a `trep.ConvergenceError` is raised. The default value is 30.

DOptimizer.**armijo_simulate**(*bX*, *bU*, *Kproj*)
This is a sub-function for armijo search. It projects the trajectory bX,bU to a real trajectory like DSystem.project, but it also returns a partial trajectory if the simulation fails. It is not intended to be used directly.

DOptimizer.**armijo_search**(*X*, *U*, *Kproj*, *dX*, *dU*)
Perform an Armijo line search from the trajectory X,U along the tangent trajectory dX, dU. Returns the named tuple `(nX, nU, nCost)` or raises `trep.ConvergenceError` if the search doesn't terminate before taking the maximum number of iterations.

This method is used by *step()* once a descent direction has been found.

Optimizing a Trajectory

DOptimizer.**step**(*iteration*, *X*, *U*, *method='steepest'*)
Perform an optimization step using a particular method.

This finds a new trajectory nX, nU that has a lower cost than the trajectory X,U. Valid methods are defined in DOptimizer.calc_descent_direction().

If the specified method fails to find an acceptable descent direction, *step()* will try again with the method returned by *select_fallback_method()*.

iteration is an integer that is used by *select_fallback_method()* and passed to the *DOptimizerMonitor* when reporting the current step progress.

Returns the named tuple `(done, nX, nU, dcost0, cost1)` where:

- *done* is a Boolean that is `True` if the trajectory *X,U* cannot be improved (i.e, *X,U* is a local minimizer of the cost).
- *nX,nU* are the improved trajectory
- *dcost0* is the derivative of the cost at *X,U*.

•*cost1* is the cost of the improved trajectory.

DOptimizer.**optimize**(*X*, *U*, *max_steps=50*)
Iteratively optimize the trajectory *X,U* until a local minimizer is found or *max_steps* are taken. The descent direction method used at each step is determined by *select_method()*.

Returns the named tuple `(converged, X, U)` where:

•*converged* is a Boolean indicating if the optimization finished on a local minimizer.

•*X,U* is the improved trajectory.

DOptimizer.**first_method_iterations**
Number of steps to take using *first_method* before switching to *second_method* for the remaining steps. See *select_method()* for more information on controlling the step method.

Default: 10

DOptimizer.**first_method**
Descent method to use for the first iterations of the optimization.

Default: "quasi"

DOptimizer.**second_method**
Descent method to use for the optimzation after *first_method_iterations* iterations have been taken.

Default: "netwon"

DOptimizer.**select_method**(*iteration*)
Select a descent direction method for the specified iteration.

This is called by *optimize()* to choose a descent direction method for each step. The default implementation takes a pre-determined number (`lower_order_iterations`) of "quasi" steps and then switches to the "newton" method.

You can customize the method selection by inheriting *DOptimizer* and overriding this method.

DOptimizer.**select_fallback_method**(*iteration*, *current_method*)
When *step()* finds a bad descent direction (e.g, positive cost derivative), this method is called to figure out what descent direction it should try next.

Debugging Tools

DOptimizer.**descent_plot**(*X*, *U*, *method='steepest'*, *points=40*, *legend=True*)
Create a descent direction plot at X,U for the specified method.

This is a useful plot for examining the line search portion of an optimization step. It plots several values along the descent direction line. All values are plotted against the line search parameter $z \in \mathbb{R}$:

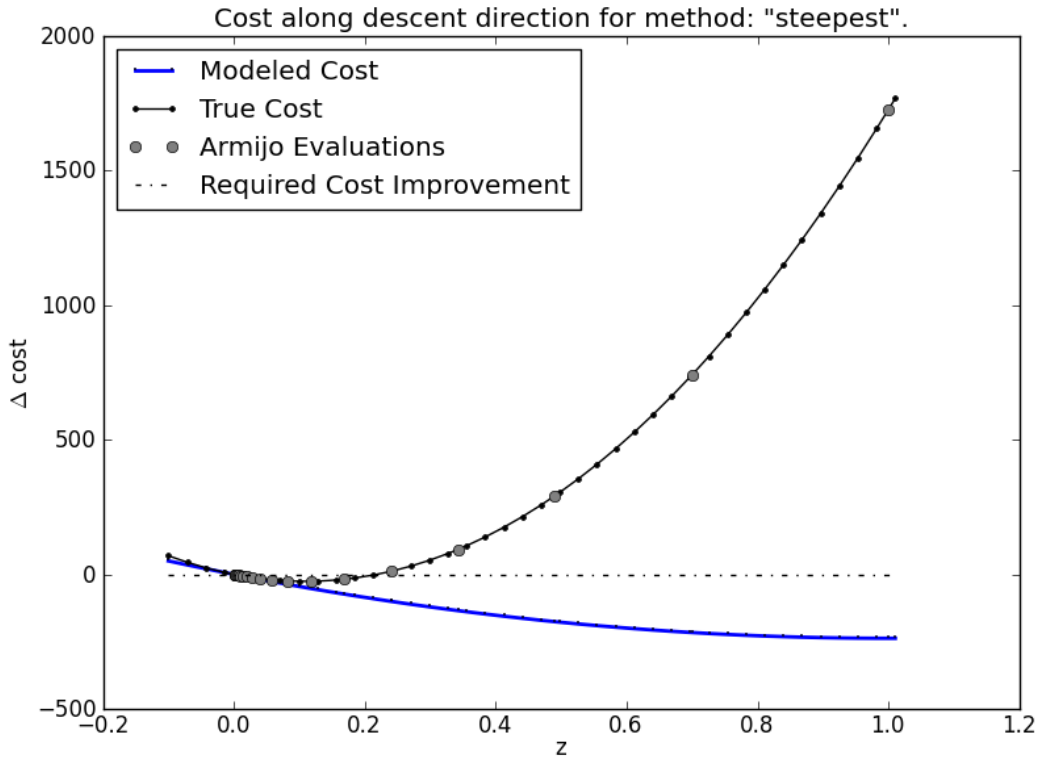
•The true cost $g(\xi + z\delta\xi) - g(\xi)$

•The modeled cost $Dg(\xi) \circ \delta\xi z + \frac{1}{2} q \circ (\delta\xi, \delta\xi) z^2$

•The sufficient decrease line: $g(\xi + z\delta\xi) < g(\xi) + \alpha z Dg(\xi) \circ \delta\xi$

•The armijo evaluation points: $z = \beta^m$

This is an example plot for a steepest descent plot during a particular optimization. This plot shows that the true cost increases much faster than the model predicts. As a result, 6 Armijo steps are required before the sufficient decrease condition is satisfied.



Example usage:

```
>>> import matplotlib.pyplot as pyplot
>>> optimizer.descent_plot(X, U, method='steepest', legend=True)
>>> pyplot.show()
```

`trep.discopt.`**`check_dcost`**(*X*, *U*, *method='steepest'*, *delta=1e-6*, *tolerance=1e-5*)
Check the calculated derivative of the cost function at X,U with a numeric approximation determined from the original cost function.

`DOptimizer.`**`check_ddcost`**(*X*, *U*, *method='steepest'*, *delta=1e-6*, *tolerance=1e-5*)
Check the second derivative of the cost function at X,U with a numeric approximation determined from the first derivative.

DOptimizerMonitor Objects

`DOptimizer` objects report optimization progress to their `monitor` object. The base implementation does nothing. The default monitor *`DOptimizerDefaultMonitor`* mainly prints

reports to the console. You can define your own monitor to gather more detailed information like saving each intermediate trajectory.

Note that if you do want to save any values, you should save copies. The optimizer might reuse the same variables in each step to optimize memory usage.

class `trep.discopt.`**`DOptimizerMonitor`**
This is the base class for Optimizer Monitors. It does absolutely nothing, so you can use this as your monitor if you want completely silent operation.

DOptimizerMonitor.**optimize_begin**(*X*, *U*)
Called when DOptimizer.optimize() is called with the initial trajectory.

DOptimizerMonitor.**optimize_end**(*converged*, *X*, *U*, *cost*)
Called before DOptimizer.optimize() returns with the results of the optimization.

DOptimizerMonitor.**step_begin**(*iteration*)
Called at the start of each DOptimize.step(). Note that step calls itself with the new method when one method fails, so this might be called multiple times with the same iteration.

All calls will be related to the same iteration until step_termination or step_completed are called.

DOptimizerMonitor.**step_info**(*method*, *cost*, *dcost*, *X*, *U*, *dX*, *dU*, *Kproj*)
Called after a descent direction has been calculated.

DOptimizerMonitor.**step_method_failure**(*method*, *cost*, *dcost*, *fallback_method*)
Called when a descent method results in a positive cost derivative.

DOptimizerMonitor.**step_termination**(*cost*, *dcost*)
Called if dcost satisfies the descent tolerance, indicating that the current trajectory is a local minimizer.

DOptimizerMonitor.**step_completed**(*method*, *cost*, *nX*, *nU*)
Called at the end of an optimization step with information about the new trajectory.

DOptimizerMonitor.**armijo_simulation_failure**(*armijo_iteration*, *nX*, *nU*, *bX*, *bU*)
Called when a simulation fails (usually an instability) during the evaluation of the cost in an armijo step. The Armijo search continues after this.

DOptimizerMonitor.**armijo_search_failure**(*X*, *U*, *dX*, *dU*, *cost0*, *dcost0*, *Kproj*)
Called when the Armijo search reaches the maximum number of iterations without satisfying the sufficient decrease criteria. The optimization cannot proceed after this.

DOptimizerMonitor.**armijo_evaluation**(*armijo_iteration*, *nX*, *nU*, *bX*, *bU*, *cost*, *max_cost*)
Called after each Armijo evaluation. The semi-trajectory bX,bU was successfully projected into the new trajectory nX,nU and its cost was measured. The search will continue if the cost is greater than the maximum cost.

DOptimizerDefaultMonitor Objects

This is the default monitor for *DOptimizer*. It prints out information to the console and records the cost and cost derivative history.

class trep.discopt.**DOptimizerDefaultMonitor**
This is the default DOptimizer Monitor. It mainly prints status updates to stdout and records the cost and dcost history so you can create convergence plots.

DOptimizerDefaultMonitor.**cost_history**
DOptimizerDefaultMonitor.**dcost_history**
Dictionaries mapping the iteration number to the cost and cost derivative, respectively. These are reset at the beginning of each optimization.

CHAPTER 6

trep.ros - ROS Tools

This module provides tools for using *trep* with the Robot Operating System (ROS). More information can be found on the python_trep wiki page for ROS.

URDF Import Tool

The *trep.ros* module provides several tools for working with the ROS environment. The URDF import function allows trep to interface directly with the Unified Robot Description Format (URDF). See the python_trep wiki page for more details on supported URDF tags.

trep.ros.**import_urdf**(*source*, *system=None*, *prefix=None*)

> **Parameters** **source** – string containing the URDF xml data
>
> **Return type** trep.system class

This function creates the trep.system class and fills in the frames and joints as defined in the URDF xml data.

source is a string containing the URDF data. May be loaded from the parameter server if using a launch file.

system is a previously defined trep.system class. If provided, the URDF will be imported into the existing system.

prefix is a string to be prefixed to all trep frames and configs created when importing the URDF. This is useful if importing the same URDF multiple times in a single system, ie. multiple robots.

The function returns the trep system defining the URDF.

trep.ros.**import_urdf_file**(*filename*, *system=None*, *prefix=None*)

> **Parameters** **filename** (*string*) – path to URDF
>
> **Return type** trep.system class

This function creates the trep.system class and fills in the frames and joints as defined in the URDF file.

filename is the complete path to the URDF file.

system is a previously defined `trep.system` class. If provided, the URDF will be imported into the existing system.

prefix is a string to be prefixed to all trep frames and configs created when importing the URDF. This is useful if importing the same URDF multiple times in a single system, ie. multiple robots.

The function returns the trep system defining the URDF.

`ROSMidpointVI` - ROS Midpoint Variational Integrator

The *`ROSMidpointVI`* class wraps the *`trep.MidpointVI`* class to implement a variational integrator with a few extra features for use in a ROS environment. This class is a superset of the *`trep.MidpointVI`* class, so see the *`trep.MidpointVI`* documentation for a full listing of the class description.

Initializing the *`ROSMidpointVI`* class creates a `rospy` publisher which will automatically publish all trep frames to the /tf topic when `step()` is called.

class `trep.ros.`**`ROSMidpointVI`**(*system*, *timestep*, *tolerance=1e-10*, *num_threads=None*)

Create a new empty mechanical system. *system* is a valid `System` object that will be simulation.

***timestep* is a different requirement for the *`ROSMidpointVI`* class. This** sets the fixed timestep for the system simulation. The value is in seconds, ie. 0.1.

***tolerance* sets the desired tolerance of the root solver when** solving the DEL equation to advance the integrator.

`MidpointVI` makes use of multithreading to speed up the calculations. *num_threads* sets the number of threads used by this integrator. If *num_threads* is `None`, the integrator will use the number of available processors reported by Python's `multiprocessing` module.

Simulation

`ROSMidpointVI.`**`step`**(*u1=tuple()*, *k2=tuple()*, *max_iterations=200*, *lambda1_hint=None*, *q2_hint=None*)

Step the integrator forward by one timestep . This advances the time and solves the DEL equation. The current state will become the previous state (ie, $t_2 \Rightarrow t_1$, $q_2 \Rightarrow q_1$, $p_2 \Rightarrow p_1$). The solution will be saved as the new state, available through `t2`, `q2`, and `p2`. `lambda` will be updated with the new constraint force, and `u1` will be updated with the value of *u1*. The frames are also published to the /tf topic.

lambda1 and *q2* can be specified to seed the root solving algorithm. If they are `None`, the previous values will be used.

The method returns the number of root solver iterations needed to find the solution.

Raises a `ConvergenceError` exception if the root solver cannot find a solution after *max_iterations*.

`ROSMidpointVI.`**`sleep`**()

Calls the `rospy.Rate.sleep()` method set to *timestep* of the *`ROSMidpointVI`* class. This method attempts to keep a loop at the specified frequency accounting for the time used by any operations during the loop.

Raises a `rospy.ROSInterruptException` exception if sleep is interrupted by shutdown.

CHAPTER 7

Misc. Components

`Spline` – Spline Objects

class `trep.`**`Spline`**(*points*)

Parameters **`points`** – A list of points defining the spline. See details.

The *`Spline`* class implements 1D spline interpolation between a set of points. *`Spline`* provides methods to evaluate points on the spline curve as well as its first and second derivatives. It can be used by new types of forces, potentials, and constraints to implement calculations.

The spline is defined by the list *points*. Each entry in *points* is a list of 2-4 numbers. The first two numbers are the *x* and *y* values of the points. The remaining two numbers, if provided and not `None`, are the first and second derivatives of the curve at that point. Any derivatives not provided are determined by the resulting interpolation.

For *N* points, The spline will comprise *N-1* polynomials of order 3-5, depending on how many derivatives were specified. *`Spline`* will choose the lowest order polynomials possible while still being able to satisfy the specified values. Specifying derivatives directly is much more effective than placing several points infinitesimally close to force the curve into a particular shape.

Spline Objects

`Spline.`**`x_points`**

Return type `numpy.ndarray`

List of the x points that define this spline.

(read-only)

`Spline.`**`y_points`**

Return type `numpy.ndarray`

List of the y points that define this spline.

(read-only)

Spline.**coefficients**

Return type numpy.ndarray

The coefficients of the interpolating polynomials.

(read-only)

Spline.**copy**()

Return type *Spline*

Create a new copy of this *Spline*.

Spline.**y**(*x*)

Return type Float

Evaluate the spline at *x*.

Spline.**dy**(*x*)

Return type Float

Evaluate the derivative of the spline at *x*.

Spline.**ddy**(*x*)

Return type Float

Evaluate the second derivative of the spline at *x*.

TapeMeasure – Measuring distances between frames

class trep.**TapeMeasure**(*system*, *frames*)

Parameters

- **system** – The *System* that the frames belong to.
- **frames** – A list of *Frame* objects or frame names.

A *TapeMeasure* object calculates the length of the line you get from playing "connect the dots" with the origins of a list of coordinate frames. *TapeMeasure* can calculate the length of the line and its derivatives with respect to configuration variables, and the velocity of the length ($\frac{dx}{dt}$) and its derivatives.

(figure here)

TapeMeasure can be used as the basis for new constraints, potentials, and forces, or used independently for your own calculations.

Length and Velocity Calculations

Let $(p_0, p_1, p_2 \ldots p_n)$ be the points at the origins of the frames specified by frames. The length, x is calculated as follows.

$$v_k = p_{k+1} - p_k$$
$$x_k = \sqrt{v_k^T v_k}$$
$$x = \sum_{k=0}^{n-1} x_k$$

The velocity is calculated by applying the chain rule to x:

$$\dot{x} = \sum_k \sum_i \frac{\partial x_k}{\partial q_i} \dot{q}_i$$

These calculations, and their derivatives, are optimized internally to take advantage of the fact that many of these terms are zero, significantly reducing the amount of calculation to do.

Warning: The derivatives of the length and velocity do not exist when the length of any part of the segment is zero. *TapeMeasure* does not check for this condition and will return `NaN` or cause a divide-by-zero error. Be careful to avoid these cases.

TapeMeasure Objects

`TapeMeasure.`**`system`**
The system that the *TapeMeasure* works in.

(read-only)

`TapeMeasure.`**`frames`**
A tuple of *Frame* objects that define the lines being measured.

(read-only)

`TapeMeasure.`**`length`**`()`

Return type `Float`

Calculate the total length of the line segments at the system's current configuration.

`TapeMeasure.`**`length_dq`**`(`*q1*`)`

Parameters **q1** (*Config*) – Derivative variable

Return type `Float`

Calculate the derivative of the length with respect to the value of *q1*.

`TapeMeasure.`**`length_dqdq`**`(`*q1*, *q2*`)`

Parameters

- **q1** (*Config*) – Derivative variable
- **q2** (*Config*) – Derivative variable

Return type `Float`

Calculate the second derivative of the length with respect to the value of *q1* and the value of *q2*.

`TapeMeasure.`**`length_dqdqdq`**`(`*q1*, *q2*, *q3*`)`

Parameters

- **q1** (*Config*) – Derivative variable
- **q2** (*Config*) – Derivative variable
- **q3** (*Config*) – Derivative variable

Return type `Float`

Calculate the third derivative of the length with respect to the value of *q1*, the value of *q2*, and the value of *q3*.

TapeMeasure.**velocity**()

Return type Float

TapeMeasure.**velocity_dq**(*q1*)

Parameters **q1** (*Config*) – Derivative variable

Return type Float

Calculate the derivative of the velocity with respect to the value of *q1*.

TapeMeasure.**velocity_ddq**(*dq1*)

Parameters **dq1** (*Config*) – Derivative variable

Return type Float

Calculate the derivative of the velocity with respect to the velocity of *q1*.

TapeMeasure.**velocity_dqdq**(*q1*, *q2*)

Parameters

- **q1** (*Config*) – Derivative variable
- **q2** (*Config*) – Derivative variable

Return type Float

Calculate the second derivative of the velocity with respect to the value of *q1* and the value of *q2*.

TapeMeasure.**velocity_dddqdq**(*dq1*, *dq2*)

Parameters

- **dq1** (*Config*) – Derivative variable
- **q2** (*Config*) – Derivative variable

Return type Float

Calculate the second derivative of the velocity with respect to the velocity of *q1* and the value of *q2*.

Visualization

TapeMeasure.**opengl_draw**(*width=1.0*, *color=(1.0*, *1.0*, *1.0)*)

Draw a representation of the line defined by the *TapeMeasure* with the specified width and color. The current OpenGL coordinate system should be the root coordinate frame of the *System*.

This method can be called by constraints, forces, and potentials that are based on the *TapeMeasure*.

Verifying Derivatives

TapeMeasure.**validate_length_dq**([*delta=1e-6*, *tolerance=1e-6*, *verbose=False*])
TapeMeasure.**validate_length_dqdq**([*delta=1e-6*, *tolerance=1e-6*, *verbose=False*])
TapeMeasure.**validate_length_dqdqdq**([*delta=1e-6*, *tolerance=1e-6*, *verbose=False*])
TapeMeasure.**validate_velocity_dq**([*delta=1e-6*, *tolerance=1e-6*, *verbose=False*])
TapeMeasure.**validate_velocity_ddq**([*delta=1e-6*, *tolerance=1e-6*, *verbose=False*])
TapeMeasure.**validate_velocity_dqdq**([*delta=1e-6*, *tolerance=1e-6*, *verbose=False*])

TapeMeasure.**validate_velocity_ddqdq**([*delta=1e-6*, *tolerance=1e-6*, *verbose=False*])

Unlike *Constraint*, *Potential*, and *Force*, *TapeMeasure* is used directly and all of the calculations are already implemented and verified. These functions are primarily used for testing during developement, but they might be useful for debugging if you are using a *TapeMeasure* and having trouble.

CHAPTER 8

Trep "Hello World"

This tutorial is meant to be an introduction to *trep* by walking the reader through extended examples. This tutorial will use a Python shell known as IPython for illustrating the organization of trep's components. It is recommended that IPython is downloaded and installed, although the tutorial is still useful without access to IPython.

Let's begin by creating the trep equivalent of "Hello World".

Import the trep module into python

Let us begin by importing the trep module into Python.

```
import trep
```

After importing trep if you type `trep` followed by pressing tab in IPython a list of all of the methods that are available for the trep module are displayed. As you can see that *trep.System* is listed, which instantiates an empty trep system. This is used in the next section.

```
>>> trep.
trep.CONST_SE3          trep.RZ                 trep.constraint         trep.ryXRT
trep.Config             trep.Spline             trep.constraints        trep.rz
trep.Constraint         trep.System             trep.finput             trep.save_
↪trajectory
trep.ConvergenceError   trep.TX                 trep.force              trep.spline
trep.Force              trep.TY                 trep.forces             trep.system
trep.Frame              trep.TZ                 trep.frame              trep.tapemeasure
trep.Input              trep.TapeMeasure        trep.load_trajectory    trep.tx
trep.MidpointVI         trep.WORLD              trep.midpointvi         trep.ty
trep.Potential          trep.config             trep.potential          trep.tz
trep.RX                 trep.const_se3          trep.potentials         trep.util
trep.RY                 trep.const_txyz         trep.rx                 trep.visual
```

Create a new instance of a trep system

Let us create a new instance of a trep system.

```
system = trep.System()
```

This creates a new instance of a trep system. Typing `system.` followed by pressing tab will list all the methods and properties of the system object.

```
>>> system.
system.L                              system.get_frame
system.L_ddq                          system.get_input
system.L_ddqddq                       system.get_potential
system.L_ddqddqdq                     system.hold_structure_changes
system.L_ddqddqdqdq                   system.import_frames
system.L_ddqdq                        system.inputs
system.L_ddqdqdq                      system.kin_configs
system.L_ddqdqdqdq                    system.lambda_
system.L_dq                           system.lambda_dddk
system.L_dqdq                         system.lambda_dddkdq
system.L_dqdqdq                       system.lambda_ddq
system.add_structure_changed_func     system.lambda_ddqddq
system.configs                        system.lambda_ddqdq
system.constraints                    system.lambda_dq
system.ddq                            system.lambda_dqdq
system.ddqd                           system.lambda_du
system.ddqk                           system.lambda_duddq
system.dq                             system.lambda_dudq
system.dqd                            system.lambda_dudu
system.dqk                            system.masses
system.dyn_configs                    system.nQ
system.export_frames                  system.nQd
system.f                              system.nQk
system.f_dddk                         system.nc
system.f_dddkdq                       system.nu
system.f_ddq                          system.potentials
system.f_ddqddq                       system.q
system.f_ddqdq                        system.qd
system.f_dq                           system.qk
system.f_dqdq                         system.resume_structure_changes
system.f_du                           system.satisfy_constraints
system.f_duddq                        system.set_state
system.f_dudq                         system.t
system.f_dudu                         system.test_derivative_ddq
system.forces                         system.test_derivative_dq
system.frames                         system.total_energy
system.get_config                     system.u
system.get_constraint                 system.world_frame
system.get_force
```

Some examples:

The current system time is

```
>>> system.t
0.0
```

The current inputs are (the system has no inputs)

```
>>> system.u
array([], dtype=float64)
```

The current generalized coordinates are (the system has no configuration)

```
>>> system.q
array([], dtype=float64)
```

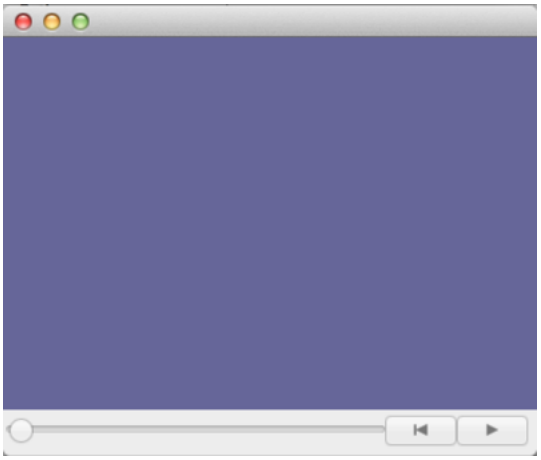
All of the properties and methods for the system object are documented in the *System* documentation. We will explore more of these properties and methods later in the tutorial.

Visualize the system with trep's visualization tools

Let us visualize the system.

```
trep.visual.visualize_3d([ trep.visual.VisualItem3D(system, [], []) ])
```

Below is what you should see.



This brings up the trep visualization of the mechanical system. Since this system has no components nothing is shown. The `VisualItem3D` method takes in three arguments: system object, time vector, and configuration vector. In this example the time vector and configuration vector are set to empty vectors. If these vectors were not empty the visualization would play a movie of the system going through the configurations at the given times.

trepHelloWorld.py code

Below is the entire script used in this section of the tutorial.

```
# trepHelloWorld.py

# Import the Trep module into python
import trep
# Create a new instance of a trep system
system = trep.System()
# Visualize the system with trep's visualization tools
trep.visual.visualize_3d([ trep.visual.VisualItem3D(system, [], []) ])
```

CHAPTER 9

Create a pendulum

Now let's create a trep model of a simple single-link pendulum, simulate a discrete trajectory, and then visualize the results.

Import necessary Python modules

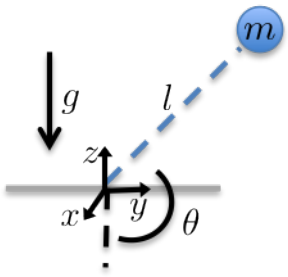
Let us begin by importing some standard modules including the trep module into python.

```
import math
from math import pi
import numpy as np
import trep
from trep import tx, ty, tz, rx, ry, rz
```

The line that should be noted here is highlighted above. `tx, ty, tz, rx, ry, rz` are trep methods that make it very easy to create new frames of reference for your system. They are used in the next section.

Build a pendulum system

Let us now build the pendulum system that corresponds to the figure below.



```
m = 1.0 # Mass of pendulum
l = 1.0 # Length of pendulum
q0 = 3./4.*pi # Initial configuration of pendulum
t0 = 0.0 # Initial time
tf = 5.0 # Final time
dt = 0.1 # Timestep

system = trep.System() # Initialize system

frames = [
    rx('theta', name="pendulumShoulder"), [
        tz(-l, name="pendulumArm", mass=m)]]
system.import_frames(frames) # Add frames
```

The first lines of code here define system parameters. Then you can see that a new trep system is created.

The frames of the system are defined using the methods that were imported above. The *Frame* documentation has a through explanation of how to create and use these frames. The first frame is created to rotate around its parent's (`system.world_frame`) X axis. It is defined by the configuration parameter `theta` and is named `pendulumShoulder`. The second frame is a translation of fixed amount (`-l`) along its parent's (`pendulumShoulder`) Z axis and is given a mass of `m`. The `system.import_frames` method is used to create the frames from the list of frame definations in `frames`.

Add forces to the system

Let us now add forces, potentials, and damping into the system.

```
trep.potentials.Gravity(system, (0, 0, -9.8)) # Add gravity
trep.forces.Damping(system, 1.0) # Add damping
trep.forces.ConfigForce(system, 'theta', 'theta-torque') # Add input
```

Gravity can work in any direction relative to the world frame. Here it is assigned to be parallel to the Z axis and have a value of $-9.8\ m/s^2$. Trep can handle other types of potentials as well. See the *Potential* documentation for information on the *trep.Potential* base class, and see the *trep.potentials* documentation for a list of the potentials that have been implemented.

Damping is applied to the entire system with the *trep.forces.Damping* method. Note that trep can also apply unique damping values to individual configurations or set default values for all configurations – see the *Damping* documentation.

An input is configured for the system by adding a configuration force with the *trep.forces.ConfigForce* method. Specifically this adds an input to the configuration variable `theta` (the configuration variable for the `pendulumShoulder` frame) with the name `theta-torque`. See *Forces* for a list of the available force types, and *trep.Force* for the documentation on the base class.

Create and initialize the variational integrator

Trep uses variational integrators to simulate the dynamics of mechanical systems.

```
mvi = trep.MidpointVI(system)
mvi.initialize_from_configs(t0, np.array([q0]), t0+dt, np.array([q0]))
```

Here a new variational integrator object `mvi` is created using our `system` instance of a *trep.System*. It is then initialized with a set of two time points and configurations using *trep.MidpointVI.initialize_from_configs*. The first two arguments are the current time and configuration and the next two are the next time and configuration. Trep calculates the discrete generalized momentum from these two pairs. You can also initialize the variational integrator with a single time, configuration, and momentum using the *trep.MidpointVI.initialize_from_state* method.

Here is a list of all the properties and methods of the variational integrator.

```
>>> mvi.
mvi.calc_f                  mvi.nk                  mvi.q2_dk2dk2
mvi.calc_p2                 mvi.nq                  mvi.q2_dp1
mvi.discrete_fm2            mvi.nu                  mvi.q2_dp1dk2
mvi.initialize_from_configs mvi.p1                  mvi.q2_dp1dp1
mvi.initialize_from_state   mvi.p2                  mvi.q2_dp1du1
mvi.lambda1                 mvi.p2_dk2              mvi.q2_dq1
mvi.lambda1_dk2             mvi.p2_dk2dk2           mvi.q2_dq1dk2
mvi.lambda1_dk2dk2          mvi.p2_dp1              mvi.q2_dq1dp1
mvi.lambda1_dp1             mvi.p2_dp1dk2           mvi.q2_dq1dq1
mvi.lambda1_dp1dk2          mvi.p2_dp1dp1           mvi.q2_dq1du1
mvi.lambda1_dp1dp1          mvi.p2_dp1du1           mvi.q2_du1
mvi.lambda1_dp1du1          mvi.p2_dq1              mvi.q2_du1dk2
mvi.lambda1_dq1             mvi.p2_dq1dk2           mvi.q2_du1du1
mvi.lambda1_dq1dk2          mvi.p2_dq1dp1           mvi.set_midpoint
mvi.lambda1_dq1dp1          mvi.p2_dq1dq1           mvi.step
mvi.lambda1_dq1dq1          mvi.p2_dq1du1           mvi.system
mvi.lambda1_dq1du1          mvi.p2_du1              mvi.t1
mvi.lambda1_du1             mvi.p2_du1dk2           mvi.t2
mvi.lambda1_du1dk2          mvi.p2_du1du1           mvi.tolerance
mvi.lambda1_du1du1          mvi.q1                  mvi.u1
mvi.nc                      mvi.q2                  mvi.v2
mvi.nd                      mvi.q2_dk2
```

Simulate the system forward

Let us now simulate this system forward in time.

```
T = [mvi.t1] # List to hold time values
Q = [mvi.q1] # List to hold configuration values
while mvi.t1 < tf:
    mvi.step(mvi.t2+dt, [0.0]) # Step the system forward by one time step
    T.append(mvi.t1)
    Q.append(mvi.q1)
```

The system is simulated forward in time using a simple while loop. First, two lists are initialized to hold all of the time values and configuration values for the simulation. Next, it enters a loop that says to continue until the variational integrator reaches the final time. In each interation of the loop the system is integrated forward by one time step. Then the new values are append to the storage vectors.

The variational integrator object has attributes for times, configurations, and discrete generalized momenta at both time points of the current integration (e.g. `mvi.t1`, `mvi.q1`, and `mvi.p1`). The *trep.MidpointVI.step* method integrates the system forward from the current time endpoint to the time given the in the first argument. The second argument specifies the input over that time period. You can see above that the input is set to zero.

Visualize the system in action

Let us use trep's visualization tools to see the system in action.

```
trep.visual.visualize_3d([ trep.visual.VisualItem3D(system, T, Q) ])
```

Finally let's create a visualization of the system being simulated. Only the system object, the list of times, and the list of configurations are needed to create the visualization.

The output that you should see on your screen is shown below.

Note: *This is an animated gif of captured screen shots. So it is slower and lower-quality than what you will see.*

pendulumSystem.py code

Below is the entire script used in this section of the tutorial.

```
# pendulumSystem.py

# Import necessary Python modules
import math
from math import pi
import numpy as np
import trep
from trep import tx, ty, tz, rx, ry, rz

# Build a pendulum system
m = 1.0 # Mass of pendulum
l = 1.0 # Length of pendulum
q0 = 3./4.*pi # Initial configuration of pendulum
t0 = 0.0 # Initial time
tf = 5.0 # Final time
dt = 0.1 # Timestep

system = trep.System() # Initialize system

frames = [
    rx('theta', name="pendulumShoulder"), [
        tz(-l, name="pendulumArm", mass=m)]]
system.import_frames(frames) # Add frames

# Add forces to the system
trep.potentials.Gravity(system, (0, 0, -9.8)) # Add gravity
trep.forces.Damping(system, 1.0) # Add damping
trep.forces.ConfigForce(system, 'theta', 'theta-torque') # Add input

# Create and initialize the variational integrator
mvi = trep.MidpointVI(system)
mvi.initialize_from_configs(t0, np.array([q0]), t0+dt, np.array([q0]))

# Simulate the system forward
T = [mvi.t1] # List to hold time values
Q = [mvi.q1] # List to hold configuration values
while mvi.t1 < tf:
    mvi.step(mvi.t2+dt, [0.0]) # Step the system forward by one time step
```

```
39      T.append(mvi.t1)
40      Q.append(mvi.q1)
41
42  # Visualize the system in action
43  trep.visual.visualize_3d([ trep.visual.VisualItem3D(system, T, Q) ])
```

CHAPTER 10

Design linear feedback controller

Let us now create a linear feedback controller that will stabilize the pendulum system that was created in the last section to its upright, unstable equilibrium.

Create pendulum system

Let us start by using the same code from the last section that creates a simple pendulum system with the addition of just a few lines.

```
# Import necessary python modules
import math
from math import pi
import numpy as np
from numpy import dot
import trep
import trep.discopt
from trep import tx, ty, tz, rx, ry, rz
import pylab

# Build a pendulum system
m = 1.0 # Mass of pendulum
l = 1.0 # Length of pendulum
q0 = 3./4.*pi # Initial configuration of pendulum
t0 = 0.0 # Initial time
tf = 5.0 # Final time
dt = 0.1 # Sampling time
qBar = pi # Desired configuration
Q = np.eye(2) # Cost weights for states
R = np.eye(1) # Cost weights for inputs

system = trep.System() # Initialize system

frames = [
```

```
    rx('theta', name="pendulumShoulder"), [
        tz(-l, name="pendulumArm", mass=m)]]
system.import_frames(frames) # Add frames

# Add forces to the system
trep.potentials.Gravity(system, (0, 0, -9.8)) # Add gravity
trep.forces.Damping(system, 1.0) # Add damping
trep.forces.ConfigForce(system, 'theta', 'theta-torque') # Add input

# Create and initialize the variational integrator
mvi = trep.MidpointVI(system)
mvi.initialize_from_configs(t0, np.array([q0]), t0+dt, np.array([q0]))
```

Three additional (highlighted above) modules are imported: `dot` from the `numpy` module, which we will use for matrix multiplication; `pylab` (part of the matplotlib project, which is used for plotting; and trep's discrete optimization module, *`trep.discopt`*.

In addition, a desired configuration value has been set to the variable `qBar`. The desired configuration is π, which is the pendulum's neutrally-stable, upright configuration. Also, correctly-dimensioned, identity state and input weighting matrices (`Q` and `R` respectively) have been created for the optimization of the linear feedback controller.

Create discrete system

Trep has a module that provides functionality for solving several linear, time-varying discrete linear-quadratic regulation problems (see *this page*).

```
TVec = np.arange(t0, tf+dt, dt) # Initialize discrete time vector
dsys = trep.discopt.DSystem(mvi, TVec) # Initialize discrete system
xBar = dsys.build_state(qBar) # Create desired state configuration
```

To do this, we use our system definition, and a prescribed time vector to create a *`trep.discopt.DSystem`*. This object is basically a wrapper for *`trep.System`* objects and *`trep.MidpointVI`* objects to represent the general nonlinear discrete systems as first order discrete nonlinear systems of the form $X(k+1) = f(X(k), U(k), k)$.

The states X and inputs U of a *`trep.discopt.DSystem`* should be noted. The state consists of the variational integrator's full configuration, the dynamic momentum, and the kinematic velocity. The full configuration consist of both dynamic states and kinematic states. The difference being that the dynamic states are governed by first-order differential equations and the kinematic states can be set directly with "kinematic" inputs. This is equivalent to saying you have "perfect" control over a dynamic configuration i.e. your system is capable of exerting unlimited force to drive the configuration to follow any arbitrary trajectory. The kinematic velocity is calculated as a finite difference of the kinematic configurations. The *`trep.discopt.DSystem`* class has a method, *`trep.discopt.DSystem.build_state`*, that will "build" this state vector from configuration, momentum, and kinematic velocity vectors. "Build" here means construct a state array of the correct dimension. Anything that is not specified is set to zero. This is used above to calculated a desired state array from the desired configuration value.

Using IPython you can display a list of all the properties and methods of the discrete system.

```
>>> dsys.
dsys.build_input               dsys.fdu                      dsys.save_state_
↪trajectory
dsys.build_state               dsys.fdudu                    dsys.set
dsys.build_trajectory          dsys.fdx                      dsys.split_input
dsys.calc_feedback_controller  dsys.fdxdu                    dsys.split_state
```

```
dsys.check_fdu                  dsys.fdxdx                  dsys.split_trajectory
dsys.check_fdudu                dsys.k                      dsys.step
dsys.check_fdx                  dsys.kf                     dsys.system
dsys.check_fdxdu                dsys.linearize_trajectory   dsys.time
dsys.check_fdxdx                dsys.load_state_trajectory  dsys.uk
dsys.convert_trajectory         dsys.nU                     dsys.varint
dsys.dproject                   dsys.nX                     dsys.xk
dsys.f                          dsys.project
```

Please refer to the *DSystem* documentation to learn more about this object.

Design linear feedback controller

Let us now design a linear feedback controller to stabilize the pendulum to the desired configuration.

```
Qd = np.zeros((len(TVec), dsys.system.nQ)) # Initialize desired configuration␣
↪trajectory
thetaIndex = dsys.system.get_config('theta').index # Find index of theta config␣
↪variable
for i,t in enumerate(TVec):
    Qd[i, thetaIndex] = qBar # Set desired configuration trajectory
    (Xd, Ud) = dsys.build_trajectory(Qd) # Set desired state and input trajectory

Qk = lambda k: Q # Create lambda function for state cost weights
Rk = lambda k: R # Create lambda function for input cost weights
KVec = dsys.calc_feedback_controller(Xd, Ud, Qk, Rk) # Solve for linear feedback␣
↪controller gain
KStabilize = KVec[0] # Use only use first value to approximate infinite-horizon␣
↪optimal controller gain

# Reset discrete system state
dsys.set(np.array([q0, 0.]), np.array([0.]), 0)
```

The `DSystem` class has a method (*trep.discopt.DSystem.calc_feedback_controller*) for calculating a stabilizing feedback controller for the system about state and input trajectories given both state and input weighting functions.

A trajectory of the desired configuration is constructed and then used with the *trep.discopt.DSystem.build_trajectory* method to create the desired state and input trajectories. The weighting functions are created with Python lambda functions that always output the state and input cost weights that were assigned to `Qk` and `Rk`. The `calc_feedback_controller` method returns the gain value K for each instance in time. To approximate the infinite-horizon optimal controller only the first value is used.

The discrete system must be reset to the initial state value because during the optimization the discrete system is integrated and thus set to the final time. Note, because the discrete system was created from the variational integrator, which was created from the system; setting the discrete system states also sets the configuration of variational integrator and the system. This can be checked by checking the values before and after running the `set` method, as shown below.

```
>>> dsys.xk
array([ 3.14159265,  0.        ])
```

```
>>> mvi.q1
array([ 3.14159265])
```

```
>>> system.q
array([ 3.14159265])
```

```
>>> dsys.set(np.array([q0, 0.]), np.array([0.]), 0)
```

```
>>> dsys.xk
array([ 2.35619449,  0.        ])
```

```
>>> mvi.q1
array([ 2.35619449])
```

```
>>> system.q
array([ 2.34019535])
```

```
>>> (mvi.q2 + mvi.q1)/2.0
array([ 2.34019535])
```

Note that the `system.q` returns the current configuration of the *system*. This is calculated using the midpoint rule and the endpoints of the variational integrator as seen above.

Simulate the system forward

As was done in the previous section of this tutorial the system is simulated forward with a while loop, which steps it forward one time step at a time.

```
T = [mvi.t1] # List to hold time values
Q = [mvi.q1] # List to hold configuration values
X = [dsys.xk] # List to hold state values
U = [] # List to hold input values
while mvi.t1 < tf-dt:
    x = dsys.xk # Grab current state
    xTilde = x - xBar # Compare to desired state
    u = -dot(KStabilize, xTilde) # Calculate input
    dsys.step(u) # Step the system forward by one time step
    T.append(mvi.t1) # Update lists
    Q.append(mvi.q1)
    X.append(x)
    U.append(u)
```

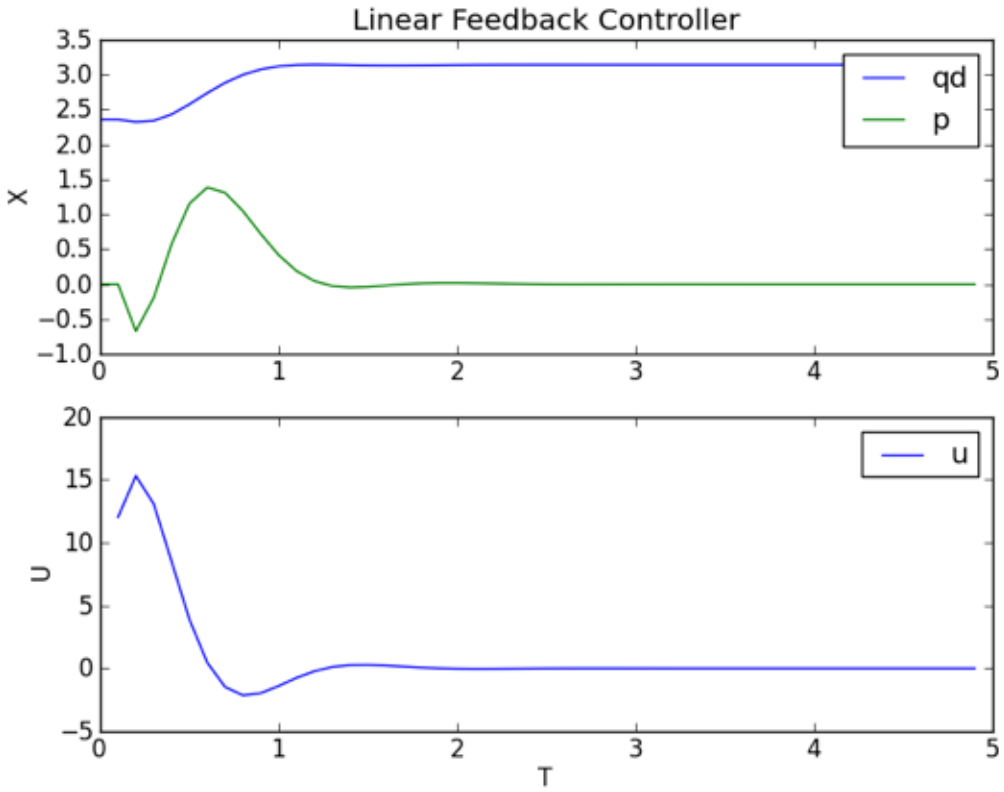
This time the system is stepped forward with the *trep.discopt.DSystem.step* method instead of the *trep.MidpointVI.step* method. This is done so that the discrete state gets updated and set as well as the system configurations and momenta. The input is calcuated with a standard negative feedback of the state error multiplied by the gain that was found previously. In addition, two more variables are created to store the state values and the input values throughout the simulation.

Visualize the system in action

The visualization is created in the exact way it was created in the last section. But this time also plotting state and input verse time.

```
trep.visual.visualize_3d([ trep.visual.VisualItem3D(system, T, Q) ])

# Plot results
ax1 = pylab.subplot(211)
pylab.plot(T, X)
pylab.title("Linear Feedback Controller")
pylab.ylabel("X")
pylab.legend(["qd","p"])
pylab.subplot(212, sharex=ax1)
pylab.plot(T[1:], U)
pylab.xlabel("T")
pylab.ylabel("U")
pylab.legend(["u"])
pylab.show()
```



From the plot you can see that it requires a large input to stablize the pendulum. What if the input torque is limited to a smaller value and we want to bring the pendulum to the upright configuration from any other configuration? By using a swing-up controller and then switching to this linear feedback controller we can accomplish this stabilization with a limited input from any configuration. We will demonstrate how this can be done with trep in the next two sections.

linearFeedbackController.py code

Below is the entire script used in this section of the tutorial.

```python
# linearFeedbackController.py

# Import necessary python modules
import math
from math import pi
import numpy as np
from numpy import dot
import trep
import trep.discopt
from trep import tx, ty, tz, rx, ry, rz
import pylab

# Build a pendulum system
m = 1.0 # Mass of pendulum
l = 1.0 # Length of pendulum
q0 = 3./4.*pi # Initial configuration of pendulum
t0 = 0.0 # Initial time
tf = 5.0 # Final time
dt = 0.1 # Sampling time
qBar = pi # Desired configuration
Q = np.eye(2) # Cost weights for states
R = np.eye(1) # Cost weights for inputs

system = trep.System() # Initialize system

frames = [
    rx('theta', name="pendulumShoulder"), [
        tz(-l, name="pendulumArm", mass=m)]]
system.import_frames(frames) # Add frames

# Add forces to the system
trep.potentials.Gravity(system, (0, 0, -9.8)) # Add gravity
trep.forces.Damping(system, 1.0) # Add damping
trep.forces.ConfigForce(system, 'theta', 'theta-torque') # Add input

# Create and initialize the variational integrator
mvi = trep.MidpointVI(system)
mvi.initialize_from_configs(t0, np.array([q0]), t0+dt, np.array([q0]))

# Create discrete system
TVec = np.arange(t0, tf+dt, dt) # Initialize discrete time vector
dsys = trep.discopt.DSystem(mvi, TVec) # Initialize discrete system
xBar = dsys.build_state(qBar) # Create desired state configuration

# Design linear feedback controller
Qd = np.zeros((len(TVec), dsys.system.nQ)) # Initialize desired configuration␣
↪trajectory
thetaIndex = dsys.system.get_config('theta').index # Find index of theta config␣
↪variable
for i,t in enumerate(TVec):
    Qd[i, thetaIndex] = qBar # Set desired configuration trajectory
    (Xd, Ud) = dsys.build_trajectory(Qd) # Set desired state and input trajectory

Qk = lambda k: Q # Create lambda function for state cost weights
Rk = lambda k: R # Create lambda function for input cost weights
KVec = dsys.calc_feedback_controller(Xd, Ud, Qk, Rk) # Solve for linear feedback␣
↪controller gain
KStabilize = KVec[0] # Use only use first value to approximate infinite-horizon␣
↪optimal controller gain
```

```
56
57 # Reset discrete system state
58 dsys.set(np.array([q0, 0.]), np.array([0.]), 0)
59
60 # Simulate the system forward
61 T = [mvi.t1] # List to hold time values
62 Q = [mvi.q1] # List to hold configuration values
63 X = [dsys.xk] # List to hold state values
64 U = [] # List to hold input values
65 while mvi.t1 < tf-dt:
66     x = dsys.xk # Grab current state
67     xTilde = x - xBar # Compare to desired state
68     u = -dot(KStabilize, xTilde) # Calculate input
69     dsys.step(u) # Step the system forward by one time step
70     T.append(mvi.t1) # Update lists
71     Q.append(mvi.q1)
72     X.append(x)
73     U.append(u)
74
75 # Visualize the system in action
76 trep.visual.visualize_3d([ trep.visual.VisualItem3D(system, T, Q) ])
77
78 # Plot results
79 ax1 = pylab.subplot(211)
80 pylab.plot(T, X)
81 pylab.title("Linear Feedback Controller")
82 pylab.ylabel("X")
83 pylab.legend(["qd","p"])
84 pylab.subplot(212, sharex=ax1)
85 pylab.plot(T[1:], U)
86 pylab.xlabel("T")
87 pylab.ylabel("U")
88 pylab.legend(["u"])
89 pylab.show()
```

CHAPTER 11

Design energy shaping swing-up controller

If the input torque is limited, the linear feedback controller may be unable to bring the pendulum to the upright configuration from an arbitrary initial condition. Let's use trep to help create a swing-up controller based on the energy of the system.

Create pendulum system

The code used to create the pendulum system is identical to the last section, except a new parameter `uMax` (highlighted below) has been added that sets the maximum absolute value of the input.

```
import math
from math import pi
import numpy as np
from numpy import dot
import trep
import trep.discopt
from trep import tx, ty, tz, rx, ry, rz
import pylab

# Build a pendulum system
m = 1.0 # Mass of pendulum
l = 1.0 # Length of pendulum
q0 = 0. # Initial configuration of pendulum
t0 = 0.0 # Initial time
tf = 10.0 # Final time
dt = 0.1 # Sampling time
qBar = pi # Desired configuration
Q = np.eye(2) # Cost weights for states
R = np.eye(1) # Cost weights for inputs
uMax = 5. # Max absolute input value

system = trep.System() # Initialize system
```

```
frames = [
    rx('theta', name="pendulumShoulder"), [
        tz(-l, name="pendulumArm", mass=m)]]
system.import_frames(frames) # Add frames

# Add forces to the system
trep.potentials.Gravity(system, (0, 0, -9.8)) # Add gravity
trep.forces.Damping(system, 1.0) # Add damping
trep.forces.ConfigForce(system, 'theta', 'theta-torque') # Add input

# Create and initialize the variational integrator
mvi = trep.MidpointVI(system)
mvi.initialize_from_configs(t0, np.array([q0]), t0+dt, np.array([q0]))

# Create discrete system
TVec = np.arange(t0, tf+dt, dt) # Initialize discrete time vector
dsys = trep.discopt.DSystem(mvi, TVec) # Initialize discrete system
xBar = dsys.build_state(qBar) # Create desired state configuration
```

Design energy shaping swing-up controller

It can be easily shown that a control law to stabilize a one-link pendulum to a reference energy is

$$u = -K\dot{\theta}(E - \bar{E})$$

where E is the current energy of the system, $\bar{E}$ is the reference energy, and K is any positive number. Therefore, the only thing that must be done to implement the energy contoller is calculate the reference energy and pick a positive gain value.

```
system.get_config('theta').q = qBar
eBar = system.total_energy()
KEnergy = 1

# Reset discrete system state
dsys.set(np.array([q0, 0.]), np.array([0.]), 0)
```

This is done by setting the system to the desired state and using the *trep.System.total_energy* method to get the desired energy level, which is called `eBar`. The gain is set to one using `KEnergy = 1`. Afterwards, the system must be reset to its initial condition.

Simulate the system forward

The system is simulated forward in the exact same way as in the last section.

```
T = [mvi.t1] # List to hold time values
Q = [mvi.q1] # List to hold configuration values
X = [dsys.xk] # List to hold state values
U = [] # List to hold input values
while mvi.t1 < tf-dt:
    x = dsys.xk # Get current state
    xDot = x[1] # Get equivilant of angular velocity
```

```
    e = system.total_energy() # Get current energy of the system
    eTilde = e - eBar # Get difference between desired energy and current energy
    # Calculate input
    if xDot == 0:
        u = np.array([uMax]) # Kick if necessary
    else:
        u = np.array([-xDot*KEnergy*eTilde])
    u = min(np.array([uMax]), max(np.array([-uMax]),u)) # Constrain input
    dsys.step(u) # Step the system forward by one time step
    T.append(mvi.t1) # Update lists
    Q.append(mvi.q1)
    X.append(x)
    U.append(u)
```

This time `u` is set to the energy shaping controller. Since the energy shaping controller depends on the angular velocity it needs a "kick" if the angular velocity is zero to get it going.

In addition the constraint on the input is included.

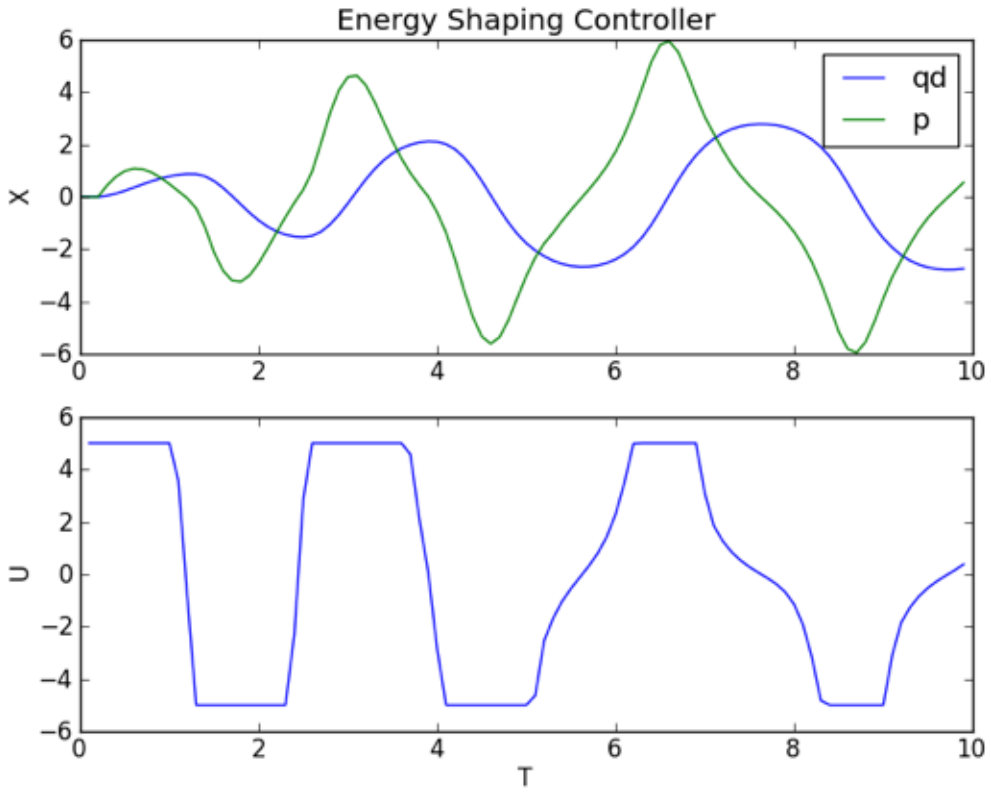
Visualize the system in action

The visualization is created in the exact way it was created in the previous sections.

```
trep.visual.visualize_3d([ trep.visual.VisualItem3D(system, T, Q) ])

# Plot results
ax1 = pylab.subplot(211)
pylab.plot(T, X)
pylab.title("Energy Shaping Controller")
pylab.ylabel("X")
pylab.legend(["qd","p"])
pylab.subplot(212, sharex=ax1)
pylab.plot(T[1:], U)
pylab.xlabel("T")
pylab.ylabel("U")
pylab.show()
```

Let's also plot the state and input verse time.



In the animation you can see that the pendulum swings-up and approaches the upright configuration. It does not quite get there because the damping term is not accounted for in the controller. However, it should be close enough that the linear controller designed in the last section can take over and stabilize to the upright configuration. Also, from the plot you can see that the input is limited from -5 to +5.

Complete code

Below is the entire script used in this section of the tutorial.

```
# enerygShapingSwingupController.py

# Import necessary python modules
import math
from math import pi
import numpy as np
from numpy import dot
import trep
import trep.discopt
from trep import tx, ty, tz, rx, ry, rz
import pylab

# Build a pendulum system
m = 1.0 # Mass of pendulum
l = 1.0 # Length of pendulum
q0 = 0. # Initial configuration of pendulum
t0 = 0.0 # Initial time
```

```
tf = 10.0 # Final time
dt = 0.1 # Sampling time
qBar = pi # Desired configuration
Q = np.eye(2) # Cost weights for states
R = np.eye(1) # Cost weights for inputs
uMax = 5. # Max absolute input value

system = trep.System() # Initialize system

frames = [
    rx('theta', name="pendulumShoulder"), [
        tz(-l, name="pendulumArm", mass=m)]]
system.import_frames(frames) # Add frames

# Add forces to the system
trep.potentials.Gravity(system, (0, 0, -9.8)) # Add gravity
trep.forces.Damping(system, 1.0) # Add damping
trep.forces.ConfigForce(system, 'theta', 'theta-torque') # Add input

# Create and initialize the variational integrator
mvi = trep.MidpointVI(system)
mvi.initialize_from_configs(t0, np.array([q0]), t0+dt, np.array([q0]))

# Create discrete system
TVec = np.arange(t0, tf+dt, dt) # Initialize discrete time vector
dsys = trep.discopt.DSystem(mvi, TVec) # Initialize discrete system
xBar = dsys.build_state(qBar) # Create desired state configuration

# Design linear feedback controller
Qd = np.zeros((len(TVec), dsys.system.nQ)) # Initialize desired configuration␣
↪trajectory
thetaIndex = dsys.system.get_config('theta').index # Find index of theta config␣
↪variable
for i,t in enumerate(TVec):
    Qd[i, thetaIndex] = qBar # Set desired configuration trajectory
    (Xd, Ud) = dsys.build_trajectory(Qd) # Set desired state and input trajectory

Qk = lambda k: Q # Create lambda function for state cost weights
Rk = lambda k: R # Create lambda function for input cost weights
KVec = dsys.calc_feedback_controller(Xd, Ud, Qk, Rk) # Solve for linear feedback␣
↪controller gain
KStabilize = KVec[0] # Use only use first value to approximate infinite-horizon␣
↪optimal controller gain

# Design energy shaping swing-up controller
system.get_config('theta').q = qBar
eBar = system.total_energy()
KEnergy = 1

# Reset discrete system state
dsys.set(np.array([q0, 0.]), np.array([0.]), 0)

# Simulate the system forward
T = [mvi.t1] # List to hold time values
Q = [mvi.q1] # List to hold configuration values
X = [dsys.xk] # List to hold state values
U = [] # List to hold input values
while mvi.t1 < tf-dt:
```

```
    x = dsys.xk # Get current state
    xDot = x[1] # Get equivilant of angular velocity
    e = system.total_energy() # Get current energy of the system
    eTilde = e - eBar # Get difference between desired energy and current energy
    # Calculate input
    if xDot == 0:
        u = np.array([uMax]) # Kick if necessary
    else:
        u = np.array([-xDot*KEnergy*eTilde])
    u = min(np.array([uMax]), max(np.array([-uMax]),u)) # Constrain input
    dsys.step(u) # Step the system forward by one time step
    T.append(mvi.t1) # Update lists
    Q.append(mvi.q1)
    X.append(x)
    U.append(u)

# Visualize the system in action
trep.visual.visualize_3d([ trep.visual.VisualItem3D(system, T, Q) ])

# Plot results
ax1 = pylab.subplot(211)
pylab.plot(T, X)
pylab.title("Energy Shaping Controller")
pylab.ylabel("X")
pylab.legend(["qd","p"])
pylab.subplot(212, sharex=ax1)
pylab.plot(T[1:], U)
pylab.xlabel("T")
pylab.ylabel("U")
pylab.show()
```

CHAPTER 12

Calculate optimal switching time

Now let us calculate the optimal time to switch between the energy shaping swing-up controller and the linear feedback stabilizing controller. In order to determine this time, tools from optimal control are employed; however, the details of why these tools work is not explained here. The point is to show how trep facilitates the implementation because it has already precomputed all the necessary derivatives.

Create pendulum system

The code used to create the pendulum system and the controllers is identical to the previous sections, except a new variable has been added that designates the switching time. This is initially set to an initial guess equal to one-half the final time. We will then use this initial guess to find the optimal switching time. This variable is `tau` is highlighted in the code below.

```
import math
from math import pi
import numpy as np
from numpy import dot
import trep
import trep.discopt
from trep import tx, ty, tz, rx, ry, rz
import pylab

# Build a pendulum system
m = 1.0 # Mass of pendulum
l = 1.0 # Length of pendulum
q0 = 0. # Initial configuration of pendulum
t0 = 0.0 # Initial time
tf = 10.0 # Final time
dt = 0.01 # Sampling time
qBar = pi # Desired configuration
Qi = np.eye(2) # Cost weights for states
Ri = np.eye(1) # Cost weights for inputs
uMax = 5.0 # Max absolute input value
```

```
tau = 5.0 # Switching time

system = trep.System() # Initialize system

frames = [
    rx('theta', name="pendulumShoulder"), [
        tz(-l, name="pendulumArm", mass=m)]]
system.import_frames(frames) # Add frames

# Add forces to the system
trep.potentials.Gravity(system, (0, 0, -9.8)) # Add gravity
trep.forces.Damping(system, 1.0) # Add damping
trep.forces.ConfigForce(system, 'theta', 'theta-torque') # Add input

# Create and initialize the variational integrator
mvi = trep.MidpointVI(system)
mvi.initialize_from_configs(t0, np.array([q0]), t0+dt, np.array([q0]))

# Create discrete system
TVec = np.arange(t0, tf+dt, dt) # Initialize discrete time vector
dsys = trep.discopt.DSystem(mvi, TVec) # Initialize discrete system
xBar = dsys.build_state(qBar) # Create desired state configuration

# Design linear feedback controller
Qd = np.zeros((len(TVec), dsys.system.nQ)) # Initialize desired configuration␣
↪trajectory
thetaIndex = dsys.system.get_config('theta').index # Find index of theta config␣
↪variable
for i,t in enumerate(TVec):
    Qd[i, thetaIndex] = qBar # Set desired configuration trajectory
    (Xd, Ud) = dsys.build_trajectory(Qd) # Set desired state and input trajectory

Qk = lambda k: Qi # Create lambda function for state cost weights
Rk = lambda k: Ri # Create lambda function for input cost weights
KVec = dsys.calc_feedback_controller(Xd, Ud, Qk, Rk) # Solve for linear feedback␣
↪controller gain
KStabilize = KVec[0] # Use only use first value to approximate infinite-horizon␣
↪optimal controller gain

# Design energy shaping swing-up controller
system.get_config('theta').q = qBar
eBar = system.total_energy()
KEnergy = 1
```

Create cost

In order to perform an optimization anything there must be some sort of cost functional that is desirable to minimize. Trep has built in objects to deal with cost functions and that is what is used here.

```
cost = trep.discopt.DCost(Xd, Ud, Qi, Ri)
```

The cost object is created with the *trep.discopt.DCost* method, which takes in a state trajectory, input trajectory, state weighting matrix, and input weighting matrix.

In this case we used the same trajectories and weights that were used for the design of the linear feedback controller that stabilizes the pendulum to the upright unstable equilibrium.

Below is a list of the properties and methods of the cost class.

```
>>> cost.
cost.Q        cost.R        cost.l_du     cost.l_dx     cost.l_dxdx   cost.m_dx     cost.ud
cost.Qf       cost.l        cost.l_dudu   cost.l_dxdu   cost.m        cost.m_dxdx   cost.xd
```

Please refer to the *Discrete Trajectory Cost* documentation to learn more this object.

Angle utility functions

Since we would like the *stabilizing controller* to stabilize to any multiple of π, two helper functions are used to wrap the angles to between 0 and 2π and between $-\pi$ and π.

```
def wrapTo2Pi(ang):
    return ang % (2*pi)
```

```
def wrapToPi(ang):
    return (ang + pi) % (2*pi) - pi
```

Simulate function

The simulation of the system is done exactly the same way as in previous sections, except this time the forward simulation is wrapped into its own function. This makes running the optimization simpler because the simulate function can be called with different switching times instead of having to have multiple copies of the same code.

```
def simulateForward(tau, dsys, q0, xBar):
    mvi = dsys.varint
    tf = dsys.time[-1]

    # Reset discrete system state
    dsys.set(np.array([q0, 0.]), np.array([0.]), 0)

    # Initial conditions
    k = dsys.k
    t = dsys.time[k]
    q = mvi.q1
    x = dsys.xk
    fdx = dsys.fdx()
    xTilde = np.array([wrapToPi(x[0] - xBar[0]), x[1]])
    e = system.total_energy() # Get current energy of the system
    eTilde = e - eBar # Get difference between desired energy and current energy

    # Initial list variables
    K = [k] # List to hold discrete update count
    T = [t] # List to hold time values
    Q = [q] # List to hold configuration values
    X = [x] # List to hold state values
    Fdx = [fdx] # List to hold partial to x
    E = [e] # List to hold energy values
    U = [] # List to hold input values
```

```
    L = [] # List to hold cost values

    # Simulate the system forward
    while mvi.t1 < tf-dt:
        if mvi.t1 < tau:
            if x[1] == 0:
                u = np.array([uMax]) # Kick if necessary
            else:
                u = np.array([-x[1]*KEnergy*eTilde]) # Swing-up controller
        else:
            u = -dot(KStabilize, xTilde) # Stabilizing controller
        u = min(np.array([uMax]), max(np.array([-uMax]), u)) # Constrain input

        dsys.step(u) # Step the system forward by one time step

        # Update variables
        k = dsys.k
        t = TVec[k]
        q = mvi.q1
        x = dsys.xk
        fdx = dsys.fdx()
        xTilde = np.array([wrapToPi(x[0] - xBar[0]), x[1]])
        e = system.total_energy()
        eTilde = e - eBar
        l = cost.l(np.array([wrapTo2Pi(x[0]), x[1]]), u, k)

        # Append to lists
        K.append(k)
        T.append(t)
        Q.append(q)
        X.append(x)
        Fdx.append(fdx)
        E.append(e)
        U.append(u)
        L.append(l)

    J = np.sum(L)
    return (K, T, Q, X, Fdx, E, U, L, J)
```

The choice of which controller to use within the simulation function is decided based on if the simulation time is less than the switching time (highlighted above). If the simulation time is less than `tau` then the swing-up controller is used, otherwise the linear stabilizing controller is used. Also note that we store and return a vector of the derivatives of the dynamics with respect to the state (highlighted above) i.e. *trep.discopt.DSystem.fdx*.

Optimize

The optimization of the switching time is done in the standard way of using a gradient decent method to minimize a cost function with respect to the switching time. Both a descent direction and a step size must be calculated. The negative gradient of the cost to the switching time is used for the descent direction. Therefore, the switching time is updated with the following

$$\tau(k+1) = \tau(k) - \gamma * \frac{\partial J}{\partial \tau(k)}$$

where τ is the switching time, γ is the step size, and J is the cost function.

To calculate the negative gradient in each iteration of the optimization, first the system is simulated forward from the initial time to the final time. Then the costate of the system is simulated backward from the final time to the switching time. The gradient at the switching time is the inner product of the difference between the two control laws and the costate.

Trep greatly simplifies this calculation because it has already calculated the necessary derivatives. In the simulation function the derivative of the system with respect to x (highlighted above) is stored at each time instance. Also, the derivative of the cost with respect to x has been already calculated with trep. Both of these derivatives are used in the backward simulation of the costate (highlighted below).

An Armijo line search is used to calculate the step size in each iteration of the optimization.

```
cnt = 0
Tau = []
JVec = []
JdTau = float('Inf')
while cnt < 10 and abs(JdTau) > .001:
    cnt = cnt + 1

    # Simulate forward from 0 to tf
    (K, T, Q, X, Fdx, E, U, L, J) = simulateForward(tau, dsys, q0, xBar)

    # Simulate backward from tf to tau
    k = K[-1]
    lam = np.array([[0],[0]])
    while T[k] > tau + dt/2:
       lamDx = np.array([cost.l_dx(X[k], U[k-1], k)])
       f2Dx = Fdx[k]
       lamDt = -lamDx.T - dot(f2Dx.T,lam)
       lam = lam - lamDt*dt
       k = k - 1

    # Calculate dynamics of swing-up controller at switching time
    x = X[k]
    xTilde = np.array([wrapToPi(x[0] - xBar[0]), x[1]])
    u1 = -dot(KStabilize, xTilde)
    u1 = min(np.array([uMax]), max(np.array([-uMax]), u1))
    dsys.set(X[k], u1, k)
    f1 = dsys.f()

    # Calculate dynamics of stabilizing controller at switching time
    eTilde = E[k] - eBar
    u2 = np.array([-x[1]*KEnergy*eTilde])
    u2 = min(np.array([uMax]), max(np.array([-uMax]), u2))
    dsys.set(X[k], u2, k)
    f2 = dsys.f()

    # Calculate value  of change in cost to change in switch time
    JdTau = dot(f1-f2, lam)

    # Armijo - used to determine step size
    chi = 0
    alpha = .5
    beta = .1
    tauTemp = tau - (alpha**chi)*JdTau
    (KTemp, TTemp, QTemp, XTemp, FdxTemp, ETemp, UTemp, LTemp, JTemp) =␣
↪simulateForward(tauTemp, dsys, q0, xBar)
    while JTemp > J + (alpha**chi)*beta*(JdTau**2):
        tauTemp = tau - (alpha**chi)*JdTau
```

```
        (KTemp, TTemp, QTemp, XTemp, FdxTemp, ETemp, UTemp, LTemp, JTemp) =␣
↪simulateForward(tauTemp, dsys, q0, xBar)
        chi = chi + 1
    gamma = alpha**chi # Step size

    # Calculate new switching time
    tauPlus = tau - gamma*JdTau

    # Print iteration results
    print "Optimization iteration: %d" % cnt
    print "Current switch time: %.2f" % tau
    print "New switch time: %.2f" % tauPlus
    print "Current cost: %.2f" % J
    print "Parital of cost to switch time: %.2f" % JdTau
    print ""

    # Update to  new switching time
    tau = tauPlus

print "Optimal switch time: %.2f" % tau
```

The results of each iteration are then printed to the screen. The optimization stops when the gradient is smaller than 0.001 or the number iteration is 10 or more. Once the optimization completes the optimal switching time is printed to the screen.

Simulate with optimal switching time

Once the optimal switching time is calculated the system can be simulated forward from its initial configuration. It will reach its desired configuration given the constrained input by switching between the two controllers. In addition, this switch is performed at the optimal time.

```
(K, T, Q, X, Fdx, E, U, L, J) = simulateForward(tau, dsys, q0, xBar)
```

From the output of the simulation you can see that it take 3 iteration to converge to the optimal switching time of 4.63.

```
>>> %run optimalSwitchingTime.py
Optimization iteration: 1
Current switch time: 5.00
New switch time: 4.79
Current cost: 9680.81
Parital of cost to switch time: 1.70
---
Optimization iteration: 2
Current switch time: 4.79
New switch time: 4.63
Current cost: 9214.82
Parital of cost to switch time: 0.62
---
Optimization iteration: 3
Current switch time: 4.63
New switch time: 4.63
Current cost: 9204.84
Parital of cost to switch time: 0.00
```

```
---
Optimal switch time: 4.63
```

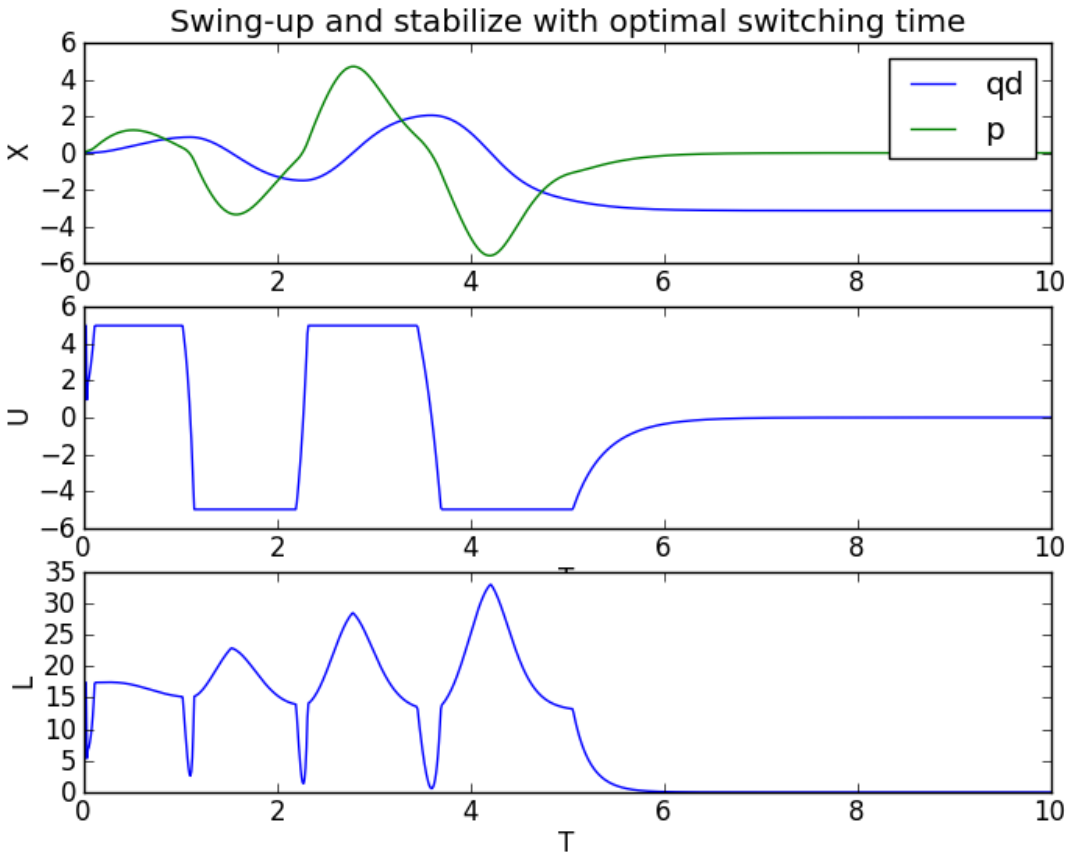
Visualize the system in action

The visualization is created just as in previous sections.

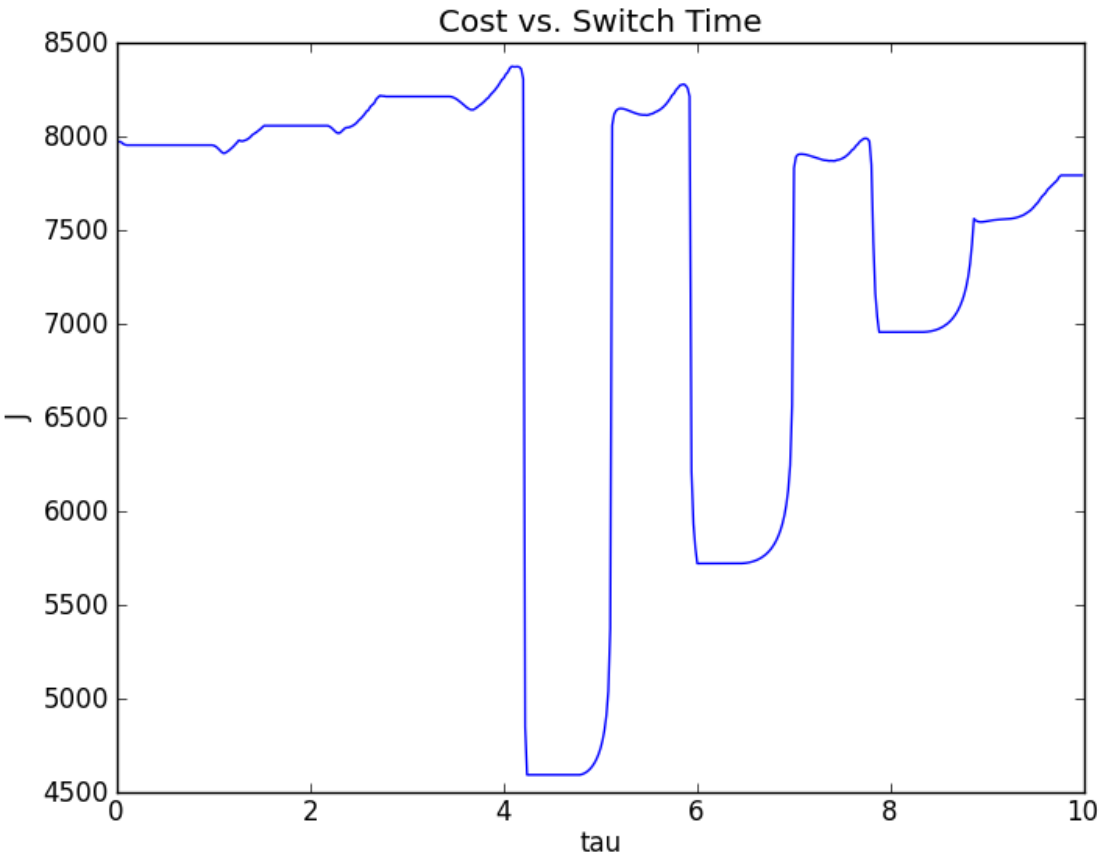
```
trep.visual.visualize_3d([ trep.visual.VisualItem3D(system, T, Q) ])

# Plot results
ax1 = pylab.subplot(311)
pylab.plot(T, X)
pylab.title("Swing-up and stabilize with optimal switching time")
pylab.ylabel("X")
pylab.legend(["qd","p"])
pylab.subplot(312, sharex=ax1)
pylab.plot(T[1:], U)
pylab.xlabel("T")
pylab.ylabel("U")
pylab.subplot(313, sharex=ax1)
pylab.plot(T[1:], L)
pylab.xlabel("T")
pylab.ylabel("L")
pylab.show()
```

Let's also plot the state, input, and cost verse time.



Just for reference the cost verse switching time is shown for all switching times from time zero to the final time. You can see that the optimization does find a switching time with a local minimum in the cost. Clearly, this cost verse switching time is non-convex, thus our gradient descent method only finds a local minimizer, and not a global minimizer.



optimalSwitchingTime.py code

Below is the entire script used in this section of the tutorial.

```
# optimalSwitchingTime.py

# Import necessary python modules
import math
from math import pi
import numpy as np
from numpy import dot
import trep
import trep.discopt
from trep import tx, ty, tz, rx, ry, rz
import pylab

# Build a pendulum system
m = 1.0 # Mass of pendulum
l = 1.0 # Length of pendulum
q0 = 0. # Initial configuration of pendulum
t0 = 0.0 # Initial time
tf = 10.0 # Final time
dt = 0.01 # Sampling time
```

```
qBar = pi # Desired configuration
Qi = np.eye(2) # Cost weights for states
Ri = np.eye(1) # Cost weights for inputs
uMax = 5.0 # Max absolute input value
tau = 5.0 # Switching time

system = trep.System() # Initialize system

frames = [
    rx('theta', name="pendulumShoulder"), [
        tz(-l, name="pendulumArm", mass=m)]]
system.import_frames(frames) # Add frames

# Add forces to the system
trep.potentials.Gravity(system, (0, 0, -9.8)) # Add gravity
trep.forces.Damping(system, 1.0) # Add damping
trep.forces.ConfigForce(system, 'theta', 'theta-torque') # Add input

# Create and initialize the variational integrator
mvi = trep.MidpointVI(system)
mvi.initialize_from_configs(t0, np.array([q0]), t0+dt, np.array([q0]))

# Create discrete system
TVec = np.arange(t0, tf+dt, dt) # Initialize discrete time vector
dsys = trep.discopt.DSystem(mvi, TVec) # Initialize discrete system
xBar = dsys.build_state(qBar) # Create desired state configuration

# Design linear feedback controller
Qd = np.zeros((len(TVec), dsys.system.nQ)) # Initialize desired configuration␣
↪trajectory
thetaIndex = dsys.system.get_config('theta').index # Find index of theta config␣
↪variable
for i,t in enumerate(TVec):
    Qd[i, thetaIndex] = qBar # Set desired configuration trajectory
    (Xd, Ud) = dsys.build_trajectory(Qd) # Set desired state and input trajectory

Qk = lambda k: Qi # Create lambda function for state cost weights
Rk = lambda k: Ri # Create lambda function for input cost weights
KVec = dsys.calc_feedback_controller(Xd, Ud, Qk, Rk) # Solve for linear feedback␣
↪controller gain
KStabilize = KVec[0] # Use only use first value to approximate infinite-horizon␣
↪optimal controller gain

# Design energy shaping swing-up controller
system.get_config('theta').q = qBar
eBar = system.total_energy()
KEnergy = 1

# Create cost
cost = trep.discopt.DCost(Xd, Ud, Qi, Ri)

# Helper functions
def wrapTo2Pi(ang):
    return ang % (2*pi)

def wrapToPi(ang):
    return (ang + pi) % (2*pi) - pi

```

```
74  def simulateForward(tau, dsys, q0, xBar):
75      mvi = dsys.varint
76      tf = dsys.time[-1]
77
78      # Reset discrete system state
79      dsys.set(np.array([q0, 0.]), np.array([0.]), 0)
80
81      # Initial conditions
82      k = dsys.k
83      t = dsys.time[k]
84      q = mvi.q1
85      x = dsys.xk
86      fdx = dsys.fdx()
87      xTilde = np.array([wrapToPi(x[0] - xBar[0]), x[1]])
88      e = system.total_energy() # Get current energy of the system
89      eTilde = e - eBar # Get difference between desired energy and current energy
90
91      # Initial list variables
92      K = [k] # List to hold discrete update count
93      T = [t] # List to hold time values
94      Q = [q] # List to hold configuration values
95      X = [x] # List to hold state values
96      Fdx = [fdx] # List to hold partial to x
97      E = [e] # List to hold energy values
98      U = [] # List to hold input values
99      L = [] # List to hold cost values
100
101     # Simulate the system forward
102     while mvi.t1 < tf-dt:
103         if mvi.t1 < tau:
104             if x[1] == 0:
105                 u = np.array([uMax]) # Kick if necessary
106             else:
107                 u = np.array([-x[1]*KEnergy*eTilde]) # Swing-up controller
108         else:
109             u = -dot(KStabilize, xTilde) # Stabilizing controller
110         u = min(np.array([uMax]), max(np.array([-uMax]), u)) # Constrain input
111
112         dsys.step(u) # Step the system forward by one time step
113
114         # Update variables
115         k = dsys.k
116         t = TVec[k]
117         q = mvi.q1
118         x = dsys.xk
119         fdx = dsys.fdx()
120         xTilde = np.array([wrapToPi(x[0] - xBar[0]), x[1]])
121         e = system.total_energy()
122         eTilde = e - eBar
123         l = cost.l(np.array([wrapTo2Pi(x[0]), x[1]]), u, k)
124
125         # Append to lists
126         K.append(k)
127         T.append(t)
128         Q.append(q)
129         X.append(x)
130         Fdx.append(fdx)
131         E.append(e)
```

```
        U.append(u)
        L.append(l)

    J = np.sum(L)
    return (K, T, Q, X, Fdx, E, U, L, J)

# Optimize
cnt = 0
Tau = []
JVec = []
JdTau = float('Inf')
while cnt < 10 and abs(JdTau) > .001:
    cnt = cnt + 1

    # Simulate forward from 0 to tf
    (K, T, Q, X, Fdx, E, U, L, J) = simulateForward(tau, dsys, q0, xBar)

    # Simulate backward from tf to tau
    k = K[-1]
    lam = np.array([[0],[0]])
    while T[k] > tau + dt/2:
       lamDx = np.array([cost.l_dx(X[k], U[k-1], k)])
       f2Dx = Fdx[k]
       lamDt = -lamDx.T - dot(f2Dx.T,lam)
       lam = lam - lamDt*dt
       k = k - 1

    # Calculate dynamics of swing-up controller at switching time
    x = X[k]
    xTilde = np.array([wrapToPi(x[0] - xBar[0]), x[1]])
    u1 = -dot(KStabilize, xTilde)
    u1 = min(np.array([uMax]), max(np.array([-uMax]), u1))
    dsys.set(X[k], u1, k)
    f1 = dsys.f()

    # Calculate dynamics of stabilizing controller at switching time
    eTilde = E[k] - eBar
    u2 = np.array([-x[1]*KEnergy*eTilde])
    u2 = min(np.array([uMax]), max(np.array([-uMax]), u2))
    dsys.set(X[k], u2, k)
    f2 = dsys.f()

    # Calculate value  of change in cost to change in switch time
    JdTau = dot(f1-f2, lam)

    # Armijo - used to determine step size
    chi = 0
    alpha = .5
    beta = .1
    tauTemp = tau - (alpha**chi)*JdTau
    (KTemp, TTemp, QTemp, XTemp, FdxTemp, ETemp, UTemp, LTemp, JTemp) =␣
↪simulateForward(tauTemp, dsys, q0, xBar)
    while JTemp > J + (alpha**chi)*beta*(JdTau**2):
        tauTemp = tau - (alpha**chi)*JdTau
        (KTemp, TTemp, QTemp, XTemp, FdxTemp, ETemp, UTemp, LTemp, JTemp) =␣
↪simulateForward(tauTemp, dsys, q0, xBar)
        chi = chi + 1
    gamma = alpha**chi # Step size
```

```
188
189     # Calculate new switching time
190     tauPlus = tau - gamma*JdTau
191
192     # Print iteration results
193     print "Optimization iteration: %d" % cnt
194     print "Current switch time: %.2f" % tau
195     print "New switch time: %.2f" % tauPlus
196     print "Current cost: %.2f" % J
197     print "Parital of cost to switch time: %.2f" % JdTau
198     print ""
199
200     # Update to  new switching time
201     tau = tauPlus
202
203 print "Optimal switch time: %.2f" % tau
204
205 # Simulate with optimal switching time
206 (K, T, Q, X, Fdx, E, U, L, J) = simulateForward(tau, dsys, q0, xBar)
207
208 # Visualize the system in action
209 trep.visual.visualize_3d([ trep.visual.VisualItem3D(system, T, Q) ])
210
211 # Plot results
212 ax1 = pylab.subplot(311)
213 pylab.plot(T, X)
214 pylab.title("Swing-up and stabilize with optimal switching time")
215 pylab.ylabel("X")
216 pylab.legend(["qd","p"])
217 pylab.subplot(312, sharex=ax1)
218 pylab.plot(T[1:], U)
219 pylab.xlabel("T")
220 pylab.ylabel("U")
221 pylab.subplot(313, sharex=ax1)
222 pylab.plot(T[1:], L)
223 pylab.xlabel("T")
224 pylab.ylabel("L")
225 pylab.show()
```

CHAPTER 13

Discrete Dynamics and Variational Integrators

The variational integrator simulates a discrete mechanical systems. Given a known state (time t_{k-1}, configuration q_{k-1}, and discrete momentum p_{k-1}) and inputs (force inputs u_{k-1} and next kinematic configuration ρ_{k-1}), the integrator finds the next state:

$$(t_{k-1}, q_{k-1}, p_{k-1}), (u_{k-1}, \rho_{k-1}) \Rightarrow (t_k, q_k, p_k)$$

The integrator also finds the discrete constraint force variable λ_{k-1}.

The variational integrator finds the next state by numerically solving the constrained *Discrete Euler-Langrange (DEL)* equation for q_k and λ_{k-1}:

$$\begin{aligned} p_{k-1} + D_1 L_d(q_{k-1}, q_k, t_{k-1}, t_{k-1}) + f_d^-(q_{k-1}, q_k, u_{k-1}, t_{k-1}, t_k) - Dh^T(q_{k-1})\dot{\lambda}_{k-1} \\ h(q_k) = 0 \end{aligned}$$

and then calculating the new discrete momentum:

$$p_k = D_2 L_d(q_{k-1}, q_k, t_{k-1}, t_k)$$

In `trep`, we simplify notation by letting $k = 2$, so that we consider t_1, q_1, and p_1 to be the previous state, and t_2, q_2, and p_2 to be the new or current state.

$$\begin{aligned} L_2 &= L_d(q_1, q_2, t_1, t_2) \\ f_2^{\pm} &= f_d^{\pm}(q_1, q_2, u_1, t_1, t_2) \\ h_1 &= h(q_1) \\ h_2 &= h(q_2) \end{aligned}$$

CHAPTER 14

Indices and tables

- genindex
- modindex

Python Module Index

t

Index

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y