LICENSE TERMS

Guide for PyRoboCOP: Python-based Robotic Control & Optimization Package for Manipulation and Collision Avoidance

Arvind U. Raghunathan[1], **Devesh K. Jha**[1], **and Diego Romeres**[1]

[1]*Mitsubishi Electric Research Laboratories* (201 Broadway, 02139, Cambridge, Massachusetts, USA
{raghunathan, jha, romeres}@merl.com

PyRoboCOP is a lightweight Python-based package for control and optimization of robotic systems described by nonlinear Differential Algebraic Equations (DAEs). In particular, the package can handle systems with contacts that are described by complementarity constraints and provides a general framework for specifying obstacle avoidance constraints.

Installation

- Download the code from `External Material`
- See the detailed installation guide `PyRoboCOPInstallationSteps.md`
- See `README.md` for instructions on running the codes.

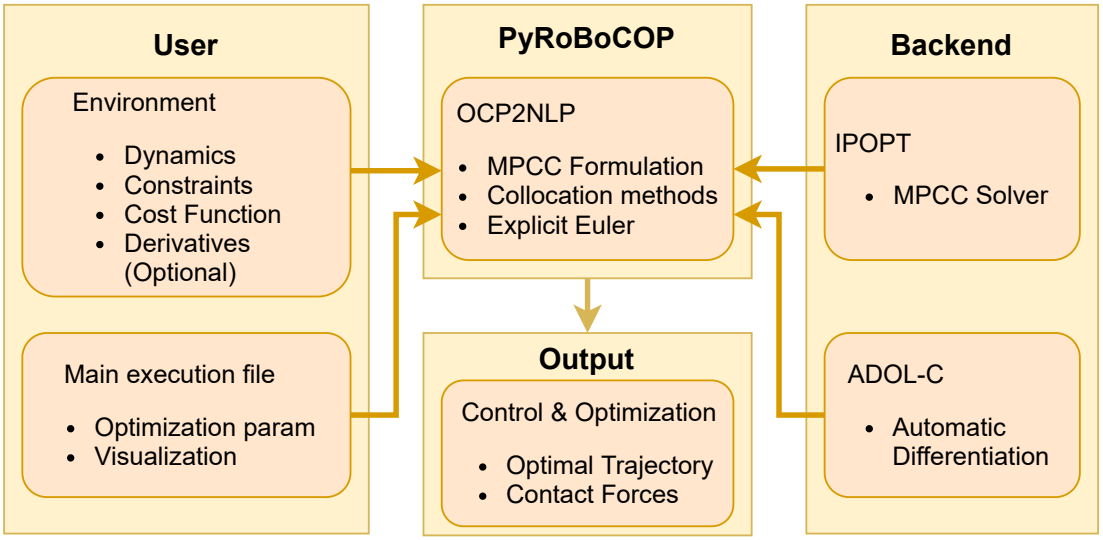

Figure 1. Workflow in PyRoBoCOP. The dynamics provided by the user to create a MPCC which is then optimized using IPOPT and the gradients are evaluated using automatic differentiation via ADOL-C.

Figure 1 provides a high-level summary of the flow of control in PyRoBoCOP. A user provided class specifies the dynamic optimization problem. This is also briefly described in Figure 1. The user needs to provide the equality constraints for the dynamical system. These constraints include the differntial and algebraic equations describing the dynamics of the system. Note that software dos not support specification of inequality constraints directly. Instead the user simply converts the inequality to an equality constraint through the introduction of a additional algebraic variable and specifying the bounds on the introduced variables, examples will be provided in the following sections. The bounds on the system state, algebraic variables and inputs, and information about complementarity constraints, if any, are provided through additional methods. Furthermore, a user needs to provide the objective function, and also has the option to provide derivative information (note the derivative information is not required when using ADOL-C). PyRoBoCOP expects the user provided class to implement certain methods in order to formulate a Mathametical Problem with Complementarity Constraints (MPCC) (or Non Linear Program (NLP)) (also shown in Figure 1). Using the example of planar pushing in the presence of obstacles, we describe the different methods that a user has to implement to solve the corresponding Optimal control problem. Note that not all these methods need to be implemented for all systems. For example, the methods for `complementarity_info` needs to be implemented only for systems with complementarity constraints. Similarly, the methods for collision avoidance need to be implemented only for problems with collision avoidance constraints.

Scenario : Planar Pushing with Obstacles

We explain how to present an Optimal Control Problem (OCP) to PyRoBoCOP through an example, planar pushing in the presence of obstacles. The example is an OCP with complementarity as well as collision avoidance constraints. Using the analytical model for planar pushing, we will explain how to create the OCP class and solve the underlying trajectory optimization problem.

The frictional interaction between the pusher and slider leads to a linear complementarity system which we describe next. The pusher interacts with the slider by exerting forces in the normal and tangential direction denoted by $f_{\vec{n}}$, $f_{\vec{t}}$ (as shown in Figure 2) as well as a torque τ about the center of the mass of the object. Assuming quasi-static interaction, the limit surface defines an invertible relationship between applied wrench $\mathbf{w}$ and the twist of the slider $\mathbf{t}$. The applied wrench $\mathbf{w}$ causes the object to move in a

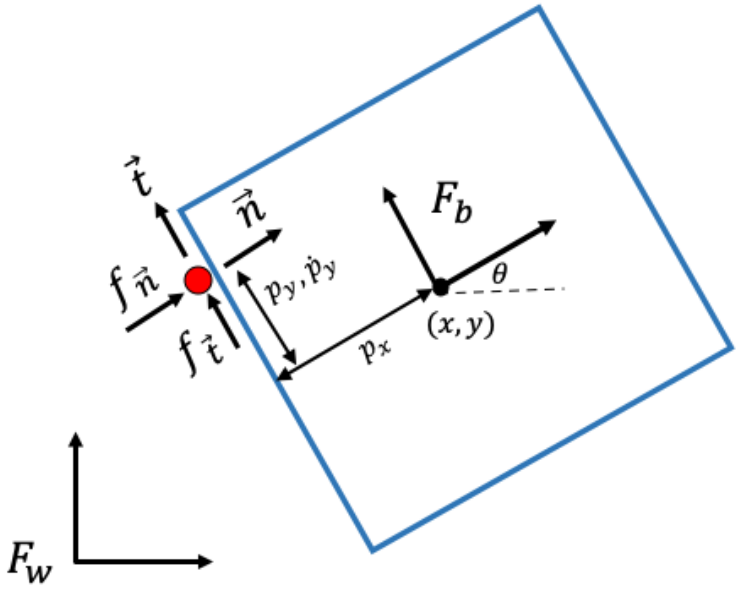

Figure 2. The planar pushing model where the pusher is the red circle and the slider is the rectangle. State of the system is given by $[x, y, \theta, p_y]^T$ assuming that the pusher only comes in contact with the left edge as shown in the figure. The world frame and the body frame of reference are denoted by F_w and F_b respectively.

perpendicular direction to the limit surface $\mathbf{H}(\mathbf{w})$. Consequently, the object twist in body frame is given by $\mathbf{t} = \nabla\mathbf{H}(\mathbf{w})$, where the applied wrench $\mathbf{w} = [f_{\vec{n}}, f_{\vec{t}}, m]$ could be written as $\mathbf{w} = \mathbf{J}^T(\vec{n} f_{\vec{n}} + \vec{t} f_{\vec{t}})$. For the contact configuration shown in Figure 2, the normal and tangential unit vectors are given by $\vec{n} = [1 \quad 0]^T$ and $\vec{t} = [0 \quad 1]^T$. The Jacobian $\mathbf{J}$ is given by $\mathbf{J} = \begin{bmatrix} 1 & 0 & -p_y \\ 0 & 1 & p_x \end{bmatrix}$.

The equation of motion of the pusher-slider system is then given by

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} \mathbf{R}\mathbf{t} \\ \dot{p}_y \end{bmatrix} \tag{1}$$

where $\mathbf{R}$ is the rotation matrix given by $\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$. Since the wrench applied on the system depends of the point of contact of pusher and slider, the state of the system is given by $\mathbf{x} = [x \quad y \quad \theta \quad p_y]^T$ and the input is given by $\mathbf{u} = [f_{\vec{n}} \quad f_{\vec{t}} \quad \dot{p}_y]^T$. The elements of the input vector must follow the laws of coulomb friction which can be expressed as complementarity conditions as follows:

$$0 \leq \dot{p}_{y+}(t) \perp (\mu_p f_{\vec{n}}(t) - f_{\vec{t}}(t)) \geq 0$$

$$0 \leq \dot{p}_{y-} \perp (\mu_p f_{\vec{n}}(t) + f_{\vec{t}}(t)) \geq 0 \tag{2}$$

where $\dot{p}_y = \dot{p}_{y+} - \dot{p}_{y-}$ and the μ_p is the coefficient of friction between the pusher and the slider. The complementarity conditions in Eq. (2) mean that both $\dot{p}_{y+}$ and $\dot{p}_{y-}$ are non-negative and only one of them is non-zero at any time instant. Furthermore, $\dot{p}_y$ is non-zero only at the boundary of friction-cone. Consequently, the slipping velocity $\dot{p}_y$ cannot be chosen as an independent control input and is optimized

while satisfying the conditions in Eq. (2). The OCP for the pusher-slider system could then be written in a minimum time formulation as:

$$\min_{\mathbf{x},\mathbf{y},\mathbf{u},p} \; t_f \tag{3a}$$

$$\text{s.t.}, \frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x},\mathbf{u}) = \begin{bmatrix} \mathbf{Rt} \\ \dot{p}_{y+} - \dot{p}_{y-} \end{bmatrix}, \mathbf{x}(0) = \hat{\mathbf{x}}, \mathbf{x}(t_f) = \tilde{\mathbf{x}} \tag{3b}$$

$$0 \leq \dot{p}_{y+}(t) \perp (\mu_p f_{\overrightarrow{n}}(t) - f_{\overrightarrow{t}}(t)) \geq 0, 0 \leq \dot{p}_{y-} \perp (\mu_p f_{\overrightarrow{n}}(t) + f_{\overrightarrow{t}}(t)) \geq 0 \tag{3c}$$

$$|f_{\overrightarrow{t}}(t)| \leq \mu_p f_{\overrightarrow{n}}(t) \tag{3d}$$

$$\underline{\mathbf{x}} \leq \mathbf{x}(t) \leq \overline{\mathbf{x}}, \underline{\mathbf{y}} \leq \mathbf{y}(t) \leq \overline{\mathbf{y}}, \underline{\mathbf{u}} \leq \mathbf{u}(t) \leq \overline{\mathbf{u}}, \underline{t}_f \leq t_f \leq \overline{t}_f \tag{3e}$$

$$\text{dist}(\mathscr{O}_0, \mathscr{O}_1(t)) > 0, \text{dist}(\mathscr{O}_2, \mathscr{O}_1(t)) > 0 \tag{3f}$$

where the quantities with $\underline{\bullet}, \overline{\bullet}$ represent the lower and upper bounds on $\bullet$ respectively. The inequality constraint representing the friction cone at the point of contact is converted into equality constraints using two algebraic variables. Similarly, $\dot{p}_{y+}$ and $\dot{p}_{y-}$ are introduced as algebraic variables which are used to represent the complementarity constraints. Furthermore, we also add collision avoidance constraints where the goal for the slider is set between two static obstacles. The objective minimizes the time required to execute the task. The sets $\mathscr{O}_0, \mathscr{O}_2$ represent the polytope bounding the extent of the static obstacles. The set $\mathscr{O}_1(t)$ denotes the polytope bounding the extent of the object that is pushed. The function $\text{dist}(\cdot,\cdot)$ returns the shortest distance between the two polytopes.

To provide more concrete details, we provide details on how we introduce algebraic variables to convert inequality constraints to equality constraints, and how do they appear in the corresponding python script. Note that in Eq (3a), we have not explained how the algebraic variables appear. We will explain that by introducing it next. We will introduce a 4-dimensional algebraic variable $\mathbf{y} = [y_0, y_1, y_2, y_3]^T$ where y_0, y_1 are defined in (4c) and $y_2 = \dot{y}_{p+}$, $y_3 = \dot{y}_{p-}$. The final-time OCP (3) can be cast in the canonical form that is solved by PyRoboCOP by first modeling the final-time as a time-invariant parameter $\mathbf{p} = [t_f]$. Further, the time is now transformed from absolute time horizon $t \in [0, t_f]$ to the scaled time horizon $\tau \in [0, 1]$. The time derivative of the differential variable $\frac{d\mathbf{x}}{dt}$ is equal to $\frac{1}{t_f}\frac{d\mathbf{x}}{d\tau}$. Then the OCP that PyRoboCOP expects to be provided is the following:

$$\min_{\mathbf{x},\mathbf{y},\mathbf{u},\mathbf{p}} \; p_0 \tag{4a}$$

$$\text{s.t.}, \frac{d\mathbf{x}}{d\tau} - p_0 \mathbf{f}(\mathbf{x},\mathbf{u}) = 0, \mathbf{x}(0) = \hat{\mathbf{x}}, \mathbf{x}(1) = \tilde{\mathbf{x}} \tag{4b}$$

$$y_0(t) = (\mu_p f_{\overrightarrow{n}}(t) + f_{\overrightarrow{t}}(t)), y_1(t) = (\mu_p f_{\overrightarrow{n}}(t) - f_{\overrightarrow{t}}(t)) \tag{4c}$$

$$0 \leq y_2(t) \perp y_1(t) \geq 0, 0 \leq y_3(t) \perp y_0(t) \geq 0 \tag{4d}$$

$$\underline{\mathbf{x}} \leq \mathbf{x}(t) \leq \overline{\mathbf{x}}, \underline{\mathbf{y}} \leq \mathbf{y}(t) \leq \overline{\mathbf{y}}, \underline{\mathbf{u}} \leq \mathbf{u}(t) \leq \overline{\mathbf{u}}, \underline{\mathbf{p}} \leq \mathbf{p} \leq \overline{\mathbf{p}} \tag{4e}$$

$$\text{dist}(\mathscr{O}_0, \mathscr{O}_1(\tau)) > 0, \text{dist}(\mathscr{O}_2, \mathscr{O}_1(\tau)) > 0 \tag{4f}$$

Note that in the above formulation, $\mathbf{u} = [f_{\overrightarrow{n}}, f_{\overrightarrow{t}}]^T$ and $\dot{p}_y = y_2 - y_3$. As can be seen above, two of the algebraic variables y_0 and y_1 are introduced to convert the inequality constraint arising due to the friction cone in Eq (3d) into equality constraints in Eq (4c). In the next section, we describe the methods that a user can use to define an OCP that is then solved using PyRoboCOP.

1 Defining an Optimal Control Problem to solve using PyRoboCOP

`pushing_with_slipping_and_obstacles.py`

This example shows the methods that PyRoboCOP *expects a user to implement in order to solve an optimal control problem with contacts in the presence of obstacles.*

The example in the file `pushing_with_slipping_and_obstacles.py` describes how to create an optimal control problem (OCP) instance that can be then solved using PyRoboCOP. We will explain the relevant variables that are used to create an OCP instance in PyRoboCOP. The example that we present covers contact constraints as well as collision avoidance constraints. For nonlinear systems without complementarity and collision avoidance constraints, a user does not need to implement some methods. We will explain as we describe the code. After the description of creation of OCP, we will describe how to pass this OCP to the solver with the different options available for the solver.

(A) Import libraries, classes and methods.

```
import numpy as np
import copy
from Envs.Dynamics.planar_pushing_slipping import PlanarPusherSlipping
```

The python file `planar_pushing_slipping.py` defines the dynamics class for planar pushing with slipping contact. This dynamics is used for the pushder-slider system manipulation in the presence of obstacles.

(B) User-defined optimal control problem (OCP) instance.

In the initialization method, we store information on the dimensions of the statem algebraic and control variables, length of time intervals and the number of time intervals. The information on the number of nonzeros in the Jacobian and Hessian are not set since ADOL-C is used for providing derivative information.

```
class pushingobstacles(object):

    def __init__(self):
        self.n_d = 4 # number of differential variables
        self.n_a = 4 # number of algebraic variables
        self.n_u = 2 # number of control variables
        self.n_p = 1 # number of time-invariant parameters
        self.n_cc = 2 # number of complementarity constraints
        self.T = 200 # number of time intervals or number of finite elements
        self.dt = 0.2 # length of the time interval
        self.times = (1./self.T)*np.array(list(range(self.T+1)))# number of finite elements
        self.nnz_jac = 0 # number of non-zeros in the Jacobian of the DAE.This only needs to be non-zero
     if  user provides derivatives.
        self.nnz_hess = 0 # number of non-zeros in the hessian of the Lagrangian of the OCP at each time
     step. This only needs to be non-zero if  user provides derivatives.
        self.mu = 0.3 # system parameter

        self.pushing_dynamics = PlanarPusherSlipping(self.dt, self.mu) # the dynamic system
```

It should be easy to verify for the user that n_d, n_a, n_u, n_p are consistent with the dimensions of $\mathbf{x}$, $\mathbf{y}$, $\mathbf{u}$, and $\mathbf{p}$ respectively. The number of complementarity constraints n_{cc} can be inferred from (4d). Notice that the number of time intervals T corresponds to the number of finite elements N_e defined in the paper.

The next method `get_info` passes the information on the problem to PyRoboCOP. The comments should clarify the role of each of the arguments expected to be returned by the user.

```
    def get_info(self):
```

```
        """
        Method to return OCP info
        n_d - number of differential vars
        n_a - number of algebraic vars
        n_u - number or controls vars
        n_cc - number of complementarity variables (part of algebraic vars)
        n_p - number of parameters
        T   - number of time-steps
        times - the time at start of each of the time intervals, an array of (T+1)
        nnz_jac - number of nonzeros in jacobian of DAE
        nnz_hess - number of nonzeros in hessian of OCP at each time-step
        """
        return self.n_d, self.n_a, self.n_u, self.n_p, self.n_cc, self.T, self.times, self.nnz_jac, self
    .nnz_hess
```

In the next method `bounds`, a user is supposed to provide upper and lower bound for all variables for the optimal control problem. For each time t, the user returns the lower and upper bounds on the states, time-derivatives of states, algebraic and control variables. Note that the ordering of the bounds should be consistent with ordering the variables in other methods (such as objective, constraints, etc.). All the bounds are to be provided as Numpy arrays. The $\bullet$lb and $\bullet$ub arrays should be identified with the arrays $\underline{\bullet}, \overline{\bullet}$ respectively.

```
    def bounds(self, t):
        """
        Method to return the bounds information on the OCP instance
        lb = [xlb,xdotlb,ylb,ulb]
        ub = [xub,xdotub,yub,uub]
        """
        # Lower bounds of the state variables.
        xlb = np.array([-5.0, -5.0, -2*np.pi, -1]) # lower bound for differential variables
        xub = np.array([5.0, 5.0, 2*np.pi, 1]) # upper bound for differential variables
        # Bounds for the state derivatives
        xdotlb = np.array([-5, -5, -5, -0.5]) # lower bound for time-derivative of differential
    variables
        xdotub = np.array([5, 5, 5, 0.5]) # upper bound for time-derivative of differential variables
        # Bounds of algebraic variable
        ylb = np.array([0.0, 0.0, 0, 0]) # lower bound for algebraic variables
        yub = np.array([np.infty, np.infty, 0.5, 0.5]) # upper bound for algebraic variables
        # Bounds of control inputs
        ulb = np.array([0, -1.0]) # lower bound for control variables
        uub = np.array([0.5, 1.0]) # upper bound for control variables

        # Stack them to get the ub and lb vector
        lb = np.hstack((xlb, xdotlb, ylb, ulb))
        ub = np.hstack((xub, xdotub, yub, uub))

        return lb, ub
```

The next method `bounds_params` provides the lower and upper bound for time-invariant parameters for the OCP. This is not a mandatory method and is only used on problems with time-invariant parameters. For instance, the user should provide this method when solving minimum-time problems. In our example, these are the lower and upper bounds for t_f.

```
    def bounds_params(self):
        """
        Method to return the bounds on the parameters in the OCP instance
        lbp - lower bound array
        ubp - upper bound array
        """
        return np.array([0.01]), np.array([100.0])
```

Using the next method `initialpoint`, a user can provide an initial guess to the OCP. In general, the performance of PyRoboCOP could improve with better initialization. PyRoboCOP passes in the time

value and the user returns a guess for the state, time-derivative of states, algebraic and control variables at that time. This method allows the user define an initial trajectory for the dynamical system if available.

```
    def initialpoint(self, t):
        """
        Method to return the initial guess for the OCP instance
        """
        x0 = np.array([0, 0, 0, 0]) # initial guess for differential variables
        xdot0 = np.array([0, 0, 0, 0]) # initial guess for time-derivatives of differential variables
        y0 = np.array([0., 0., 0., 0.]) # initial guess for algebraic variables
        u0 = np.array([0, 0]) # initial guess for control variables
        return x0, xdot0, y0, u0
```

In the next method `initialpoint_params`, we provide an initial guess for the time-invariant parameter. Note that this is only required when the time-invariant parameters are present in the problem, for example in minimum-time problems.

```
    def initialpoint_params(self):
        """
        Method to return the initial guess for the parameters in the OCP instance

        """
        return np.array([10.0])
```

The user provides the initial condition of the dynamical system through the following method `initialcondition`.

```
    def initialcondition(self):
        """
        Method to return the initial condition for the OCP instance
        """
        xic = [0, 0, 0, 0.0] # Initial state of the dynamical system
        return xic
```

This corresponds to specifying $\mathbf{x}(0) = \hat{\mathbf{x}}$ in (4b).

The method allows `bounds_finaltime` a user to pass bounds on the terminal state of the dynamical system. Setting the bounds to identical values ensures convergence to the desired terminal state. This allows to specify that $\mathbf{x}(1) = \tilde{\mathbf{x}}$. in (4b).

```
    def bounds_finaltime(self):
        """
        Method to return the bounds on the states at final time
        """
        lbxf = np.array([0.45, 0.4, 3*np.pi/2., -0.5]) # lower bound for the desired terminal state of
    the system
        ubxf = np.array([0.45, 0.4, 3*np.pi/2., 0.5]) # upper bound for the desired terminal state of
    the system
        return lbxf, ubxf
```

The next method `get_complementarity_info` allows the user to provide the complementarity information. In particular, a user needs to provide indices of all the algebraic variables which appear in complementarity constraints. Note that the indices of the complementarity variables is based on the ordering in $\mathbf{y}$. Furthermore, the user also needs to indicate whether complementarity constraint involves the upper or lower bound. Note that this method is required only for systems with complementarity constraints. The comments provided below should clarify the role of the arrays that are expected to be returned by the user.

```
    def get_complementarity_info(self):
```

```
        """
        Method to return the complementarity info
        cc_var1   - index of complementarity variables (>= 0, < n_a)
        cc_bnd1   - 0/1 array 0: lower bound, 1: upper bound
        cc_var2   - index of complementarity variables (>= 0, < n_a)
        cc_bnd2   - 0/1 array 0: lower bound, 1: upper bound
        complemntarity constraints for i = 1,...,n_cc
        y : algebraic variables
        if cc_bnd1 = 0, cc_bnd2 = 0
        y[cc_var1[i]]-lb[cc_var1[i]] >= 0 \perp y[cc_var2[i]]-lb[cc_var2[i]] >= 0
        if cc_bnd2 = 0, cc_bnd2 = 1
        y[cc_var1[i]]-lb[cc_var1[i]] >= 0 \perp ub[cc_var2[i]]-y[cc_var2[i]] >= 0
        if cc_bnd1 = 1, cc_bnd2 = 0
        ub[cc_var1[i]]-y[cc_var1[i]] >= 0 \perp y[cc_var2[i]]-lb[cc_var2[i]] >= 0
        if cc_bnd2 = 1, cc_bnd2 = 1
        ub[cc_var2[i]]-y[cc_var1[i]] >= 0 \perp ub[cc_var2[i]]-y[cc_var2[i]] >= 0
        """
        cc_var1 = np.reshape([1, 0], (self.n_cc, 1)) #
        cc_bnd1 = np.array([0, 0])
        cc_var2 = np.reshape([2, 3], (self.n_cc, 1))
        cc_bnd2 = np.array([0, 0])
        return cc_var1, cc_bnd1, cc_var2, cc_bnd2
```

The assignments above imply that the complementarity constraints are $y_1 \geq \underline{y}_1 \perp y_2 \geq \underline{y}_2$ and $y_0 \geq \underline{y}_0 \perp y_3 \geq \underline{y}_3$ which is consistent with (4d).

The objective that is part of an integral over the time horizon is provided in the method `objective`. For example, the following represents a quadratic objective function for a regulation OCP.

```
    def objective(self, t, x, xdot, y, u, params):
    #     """
    #     Method to return the objective function of the OCP instance
    #     x - numpy 1-d array of differential variables at time t
    #     xdot - numpy 1-d array of time derivative differential variables at time t
    #     y - numpy 1-d array of algebraic variables at time t
    #     u - numpy 1-d array of control avriables at time t
    #     params - parameters
    #     """
       c = (x[0]-x_goal[0])**2+(x[1]-x_goal[1])**2 + \
             (x[2]--x_goal[2])**2+(x[3]-x_goal[3])**2
         return c
```

In our example, the objective method returns 0 as shown below.

```
    def objective(self, t, x, xdot, y, u, params):
        """
        Method to return the objective function of the OCP instance
        x - numpy 1-d array of differential variables at time t
        xdot - numpy 1-d array of time derivative differential variables at time t
        y - numpy 1-d array of algebraic variables at time t
        u - numpy 1-d array of control avriables at time t
        params - parameters
        """
        return 0.0
```

The next method `objective_mayer` allows a user to implement the objective function that is dependent on state at final-time or dependent on time-invariant parameters, for example minimum time problems. In case of the minimum-time problem, the method `objective` must return 0.0. The method `objective_mayer` needs to be implemented and returns the parameter corresponding to the time horizon.

```
    def objective_mayer(self, x0, xf, params):
        """
        Method to return the objective function of the OCP instance that is not integral
```

```
        x - numpy 1-d array of differential variables at time t
        xdot - numpy 1-d array of time derivative differential variables at time t
        y - numpy 1-d array of algebraic variables at time t
        u - numpy 1-d array of control avriables at time t
        params - parameters
        """
        tf = params[0]
        return tf
```

The next method `constraint` provides the constraints for the underlying dynamical system. Note that the constraint does not need to be specified in state-space form. PyRoboCOP allows to specify the dynamics in implicit form as generalized non-linear functions.

```
    def constraint(self, c, t, x, xdot, y, u, params):
        """
        Method to return the constraint of the OCP instance
        x - numpy 1-d array of differential variables at time t
        xdot - numpy 1-d array of time derivative differential variables at time t
        y - numpy 1-d array of algebraic variables at time t
        u - numpy 1-d array of control avriables at time t
        params - parameters
        """
        #c = np.zeros((self.n_d+self.n_a,1))
        #c = c.astype(object)
        tf = params[0]
        # use this relationship to replace input
        #u_p = y[2]-y[3]

        dxdt = self.pushing_dynamics.dynamics(x, u, y)
        c[0] = -xdot[0] + tf*dxdt[0] # dynamic constraint number 1
        c[1] = -xdot[1] + tf*dxdt[1] # dynamic constraint number 1
        c[2] = -xdot[2] + tf*dxdt[2] # dynamic constraint number 1
        c[3] = -xdot[3] + tf*(y[2]-y[3]) # dynamic constraint number 1

        # # algebraic constraints for friction cone
        c[4] = y[0]-u[1]-self.mu*u[0]
        c[5] = y[1]+u[1]-self.mu*u[0]
```

The next method provides information for objects present in the OCP. This method is required only for OCP with collision avoidance constraints. This methods returns the number of objects, specifies whether they are static or dynamic and the number of vertices of the polytopes enclosing each object. Note that the collision avoidance formulation in PyRoboCOP uses a polytopic representation of objects for collision avoidance.

```
    def get_objects_info(self):
        """
        Method to return the info on the objects (assumed to be polytopic)
        n_objects - number of objects
        object_dynamic - a boolean array set to True if the object is dynamic, False otherwise
        n_vertices - array indicating the number of vertices in the polytope bounding the objects
        """
        n_objects = 3 # number of objects present in the space
        object_dynamic = [False, True, False] # True means the object is dynamic
        n_vertices = [4, 4, 4] # number of vertices for all objects
        return n_objects, object_dynamic, n_vertices
```

In our example, the objects with index $0, 2$ are static obstacles and the object being pushed is indexed as 1. The settings for `object_dynamic` are appropriately defined. All three objects are modeled using a square and so the number of vertices is 4.

The following method is also required for collision avoidance problems only. This method provides the coordinates of the vertices of the polytopes bounding the different objects in the OCP. Note that the coordinates of the vertices are functions of the states and algebraic variables in the dynamics. Note that

`verts` is a pre-allocated array and the user should only include assignments statements to the elements of the array. Commands that create a new memory address for `verts` will result in incorrect behavior.

```
    def get_object_vertices(self, verts, ind, x, xdot, y, u):
        """
        The vertices of the polytope (in 3D) bounding the object are assumed to be of the form
        V(x,xdot,y,u) = R(x,x,dot,y) * V_0 + q_c(x,xdot,y,u)
        where R(x,xdot,y,u) is the rotation matrix, V_0 is matrix with the vertices of the polytope in
    the
        columns (when objective is positioned at origin), q_c is the position of the object.
        V, V0 are 3 x (# vertices) matrices
        """
        if ind == 0:
            # this is the obstacle
            nv = 4
            #verts = np.zeros((3, nv))
            corners = 0.05*np.array([[1., 1., -1., -1.],
                                     [1., -1., 1., -1.],
                                     [0, 0, 0, 0]])

            verts[:, :] = corners + np.array([[0.30], [0.4], [0]]) # vertices of object 1


        if ind == 1:
            # this is the object - car
            nv = 4
            #verts = np.zeros((3, nv))
            #verts = verts.astype(object)
            sidexby2 = 0.045
            sideyby2 = 0.045
            diagby2 = np.sqrt(sidexby2**2 + sideyby2**2)
            x_c = x[0]
            y_c = x[1]
            theta = x[2]
            verts[0, 0] = x_c + diagby2*np.cos(np.pi/4-theta)
            verts[1, 0] = y_c + diagby2*np.sin(np.pi/4-theta)
            verts[0, 1] = x_c + diagby2*np.cos(3*np.pi/4-theta)
            verts[1, 1] = y_c + diagby2*np.sin(3*np.pi/4-theta)
            verts[0, 2] = x_c + diagby2*np.cos(5*np.pi/4-theta)
            verts[1, 2] = y_c + diagby2*np.sin(5*np.pi/4-theta)
            verts[0, 3] = x_c + diagby2*np.cos(7*np.pi/4-theta)
            verts[1, 3] = y_c + diagby2*np.sin(7*np.pi/4-theta) # vertices for object 2. Note that this
    object is dynamic, hence the dependence on x_c and y_c

        if ind == 2:
            # this is an obstacle
            nv = 4
            #verts = np.zeros((3, nv))
            corners = 0.05*np.array([[1., 1., -1., -1.],
                                     [1., -1., 1., -1.],
                                     [0, 0, 0, 0]])

            verts[:, :] = corners + np.array([[0.55], [0.4], [0]]) # vertices for object 3
```

2 Defining the main file to solve the OCP using PyRoboCOP.

`main_pushing_slipping_obstacles.py`

In the next example, we are going to explain the python script `main_pushing_slipping_obstacles.py` which calls the OCP python class and passes the OCP to IPOPT for solution. We explain them in the following with the solver options that are available to a user as part of PyRoboCOP. This example is supposed to serve as a guide to solve an OCP described earlier using PyRoboCOP.

```
import Envs.Dynamics.pushing_with_slipping_and_obstacles as ocp # import the OCP class for pushing in
    the presence of obstacles
```

```
import Solvers.ocp2nlp.OCP2NLPcompladolc as ocp2nlp # import the main solver class
import ipopt # import ipopt
import matplotlib.patches as patches # import for plotting
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.patches import Rectangle

import adolc as ad # import adolc for Automatic differentiation
from Envs.Dynamics.rotations import transform_points_twod # a utility function for plotting pusher-
    slider in global frame of reference
```

In the next snippet of code, we explain how a user can set different parameters that are used for automatic transcription provided by PyRoboCOP. When using OCP with complementarity constraints, the variable `ncolloc` is always set to 1. For smooth non-linear systems, up to 5 collocation points can be used. A user can choose either `radau`, `legendre` or `explicit` as possible options for roots. `explicit` corresponds to using the Excplicit Euler discretization for the dynamics. The user has flexibility in specifying how the complementarity constraints are solved. The choices are:

1. `compl = 0`: Pose the complementarity constraitns individually at each collocation point of each time interval

   $$y_0(t)y_3(t) \leq \delta, y_1(t)y_2(t) \leq \delta$$

   where $\delta > 0$ (explained next).

2. `compl = 1`: Aggregate all the complementarity constraints over each time interval

   $$y_0y_3 + y_1y_2 \leq \delta$$

   where $\delta > 0$ (explained next). The factor 2 indicate the number of complementarity constraints

3. `compl = 2`: Replace the complementarity constraints with a term in the objective function that sums over all the collocation points in all time intervals. The contribution to the objective function for each collocation point takes the form

   $$(1/\delta)(y_0y_3 + y_1 + y_2).$$

   where $\delta > 0$ (explained next).

The δ parameter in the complementarity constraints can be fixed or varied during the solution algorithm. This behavior is controlled with the parameter `compladapt`. With `compladapt = 1` δ is set equal to the barrier parameter of the interior point algorithm. The δ is kept fixed through the solution process when `compladapt = 0`. The convergence behavior of formulations can be quite different and we provide these implementations so the user can choose one that works best for the problem at hand. The variable `autodiff` should be set to 1 if the user wants to use ADOL-C for automatic differentiation.

```
ocpi = ocp.pushingobstacles() # OCP instance for the pushing example
ncolloc = 1 # Number for collocation points
roots = "radau" #
compl = 1 # set according to the description above
autodiff = 1 # set to 1 if using adolc for AD
compladapt = 1 #
ocp2nlpi = ocp2nlp.OCP2NLP(
    ocpi, ncolloc, roots, compl=compl, compladapt=compladapt, autodiff=autodiff)

ocp2nlpi.print_ocp()
```

In the following we show how to create an IPOPT problem instance in PyRoboCOP. This block of code could remain the same for all examples, and a user may not need to change these.

```
"""
    Create the Ipopt Problem Instance
"""
x0_ipopt = ocp2nlpi.initialpoint()
lb_ipopt, ub_ipopt, lbcon_ipopt, ubcon_ipopt = ocp2nlpi.bounds()
nlp = ipopt.problem(
    n=ocp2nlpi.n_ipopt,
    m=ocp2nlpi.m_ipopt,
    problem_obj=ocp2nlpi,
    lb=lb_ipopt,
    ub=ub_ipopt,
    cl=lbcon_ipopt,
    cu=ubcon_ipopt
)
```

A user can provide the following options for the solver. It is suggested that a user try the `derivative_test` when trying a new OCP. The default `mu_strategy` is `monotone`. Other possible options for `mu_strategy` is `adpative`. Using hessian of the lagrangian in the OCP solution allows faster convergence to solution although, it might result in slower iterations. A user can turn on `hessian_approximation` if they want to use hessian approximation for the OCP.

```
"""
    Set solver options
"""
#nlp.addOption('derivative_test', 'first-order')
#nlp.addOption('derivative_test', 'second-order')
#nlp.addOption('mu_strategy', 'monotone')
nlp.addOption('tol', 1e-6)
#nlp.addOption('max_iter', 1)
#nlp.addOption('hessian_approximation', 'limited-memory')
```

The following lines of code are used to solve the OCP.

```
"""
    Solve
"""
xopt, info = nlp.solve(x0_ipopt)

# x_sol and u_sol are the output of the NLP

x_sol, xdot_sol, y_sol, u_sol, params = ocp2nlpi.parsesolution(xopt)
```

Finally, the following lines of code could be used to extract trajectories from the OCP solution. Note that the `diff` flag is used to indicate if a variable is a differential variable. Other than the differential variables, it should be set to `False`.

```
for i in range(ocpi.n_d):
    tarray, xarray = ocp2nlpi.extract_trajectory(x_sol, i, diff=True)
```

Similarly, the input trajectories can be extracted from the underlying OCP as following.

```
for i in range(ocpi.n_u):
    tarray, uarray = ocp2nlpi.extract_trajectory(u_sol, i, diff=False)
```

A user can then plot or save the optimal trajectories obatined from PyRoboCOP.