colibry

Institute of Flight System Dynamics

Technical University of Munich

colibry.fsd@ed.tum.de

January 26, 2023

Contents

1 Prologue

Thorough testing and validation of dynamic systems are crucial to guarantee safe operation. Consider for example the task of flight control law clearance, which ensures that all relevant requirements to the closed-loop system are satisfied [1]. In practice, this assessment requires an immense amount of testing, especially for modern, highly augmented aircraft for which the flight control laws are central to safety and performance. Here, a mixture of sampling- and optimization-based methods is typically used to test whether requirements are fulfilled throughout the entirety of the flight envelope. The essential idea of optimization-based testing, as opposed to conventional sampling-based testing, is to leverage the effectiveness of numerical optimization methods to identify violations of the investigated criteria efficiently [2]. Optimization-based worst-case analysis, referred to as counter optimization, aims at driving the system under test (SUT) as close as possible to - or even beyond - the admissible limit performance. This analysis allows for the efficient identification of design weak points or, more generally, regions of presumed worst-case performance. Such information is extremely valuable for the design, verification, and validation of complex systems, especially if obtained in early stages of the development process.

The **Counter Optimization LiBraRY** (COLIBRY) constitutes a collection of optimization-based testing methods, intended for performing worst-case analyses of dynamic systems considering a combination of time-varying inputs and parameters.

If you intend to use this toolbox for scientific studies, please cite the following paper:

```
@article{colibryToolbox2023,
  author  = {D. Braun and F. Schwaiger and F. Holzapfel and J.
      Diepolder and J.Z. Ben-Asher},
  title   = {COLIBRY - A Counter Optimization Library for MATLAB},
  journal = {AIAA Scitech 2023 Forum},
  year    = {2023},
  doi     = {10.2514/6.2023-2584},
}
```

2 Getting Started

If you are a newcomer and not really familiar with MATLAB, you should start here. Otherwise you may skip ahead and directly learn the interface of COLIBRY by studying the provided demonstration examples.

2.1 Installation

COLIBRY is primarily tested for MATLAB 2021a or newer versions.

The toolbox- and external software dependencies depend on the respective method:

Method	Toolbox Dependencies
Linear Optimal Control	IBM ILOG CPLEX (commercial) or Coin-OR CLP (open-source)
BiLevelLinear Optimal Control	IBM ILOG CPLEX (commercial) or Coin-OR CLP (open-source)
Nonlinear Optimal Control	Falcon.m
Single Shooting	MATLAB "Global Optimization Toolbox"
Learning	MATLAB "Reinforcement Learning Toolbox" and MATLAB "Deep Learning Toolbox"

Moreover, in order to be able to use parallelization, the MATLAB "Parallel Computing Toolbox" is required.

To install the release version of our toolbox, first retrieve the archive from our Downloads page. Then unzip to a folder on your machine you have access to. And done! Go ahead to configuration.

Should you instead have access to our development version, which is a git repository, you may instead add the toolbox as a submodule dependency to your project. If we assume you aggregate all your submodules in a folder called `External`, you may do:

```
git submodule add <insert/url/to/colibry.git> External/colibry
```

You can also use console commands from within MATLAB. Prefix the command above with an exclamation mark `!` on the integrated shell:

```
>> !git submodule add ...
```

2.1.1 Linear Solver Dependencies

At the core of the linear- and the bi-level-linear-optimalcontrol-based method is the solution of large-scale Linear Programs (LP). In version 1.00 two LP solvers are supported:

2.1.1.1 The open-source Coin-OR CLP solver.

This solver needs some libraries installed. See below for the prerequisites to install.

Prerequisites:

- on macOS, use homebrew (https://brew.sh) to install clp and coinutils

```
brew install clp coinutils
```

- on win64, install the official DLLs from https://bintray.com/coin-or/download/Clp into $CLP_INSTALL_DIR and set the environment variable
- on linux, you need some apts: if you are just using the library

```
sudo apt-get install libgfortran4
```

if you also want to recompile the mex file

```
sudo apt-get install -y gfortran liblapack-dev libbz2-dev
```

COLIBRY will then automatically compile the clp mex file once the clp solver is called.

2.1.1.2 The commercial IBM ILOG CPLEX solver.

Follow the installation instructions at (https://www.ibm.com/docs/en/icos/12.9.0?topic=matlab-getting-started-cplex). Note that the academic licence only covers academic research work!

2.2 Configuration

Unpacking the toolbox code does not immediately make it visible in MATLAB. To use COLIBRY in your own code, adding `code/` to the MATLAB path is sufficient. Be aware, however, that future versions of this toolbox might add additional dependencies and more folders, so we suggest using our included project file instead. Opening the project will add all required folders to your search path automatically:

```
openProject path/to/colibry/
```

If you are using MATLAB projects already, you can add COLIBRY as a **Project Reference**. To do so, open your own project, click the **+ References** button, and add COLIBRY's project.

If you do not use projects, we encourage you to try them out. They help keeping software modules together.

The toolbox is structured into the following folders:

- `code/`
- `demo/`
- `resources/`
- `Colibry.prj`

2.3 Run a First Demo

Test your installation of COLIBRY by running the demonstration example `main.m` in the subfolder `demo/01-linear-optimalcontrol`.

This example considers the worst-case input analysis of a fixed-wing aircraft equipped with a flight control system as well as servos for the primary control surfaces (elevator, aileron, and rudder). The states describing the aircraft's rigid body are the absolute velocity V, the angle-of-attack α, the angle-of-sideslip β, as well as the Euler angles ϕ (roll), θ (pitch), and ψ (yaw) with the corresponding angular rates p, q, and r.

The analysis considers the linearized closed-loop dynamics of the aircraft. The operating point, used for the linearization of the nonlinear closed-loop system dynamics, represents a horizontal, wing level, and steady-state flight condition. In the vicinity of the operating condition, the equations for the longitudinal and lateral plane decouple, and are thus considered individually.

The example focuses on the lateral dynamics and investigates the maximum lateral load factor response to a combined roll angle ϕ_c and lateral load factor $n_{y,c}$ command over 8 seconds. Constraints of the physical actuation system are accounted for using state constraints. Specifically, the position and velocity of the control surfaces aileron ζ and rudder ξ are bounded by box-bounds.

Before considering the definition, set-up, and solution of this counter optimization problem in detail, we investigate the output produced by running `main.m`.

```
mySolution =

  Solution with properties:

                    OptimalCost: [1x1 colibry.linearoptimalcontrol.OptimalCost]
          OptimalStateGridVector: [10x1 colibry.linearoptimalcontrol.
              OptimalState]
        OptimalControlGridVector: [2x1 colibry.linearoptimalcontrol.
            OptimalControl]
         OptimalOutputGridVector: [2x1 colibry.linearoptimalcontrol.
             OptimalOutput]
      SwitchingFunctionGridVector: [2x1 colibry.linearoptimalcontrol.
          SwitchingFunction]
              CostateGridVector: [10x1 colibry.linearoptimalcontrol.Costate]
                    TimeToSolve: 7.3103
```

All results of the analysis - including the maximal attainable load factor n_y (`mySolution.OptimalCost.Value`), the corresponding worst-case control inputs (`mySolution.OptimalControlGridVector`) as well as the resulting state and output trajectories (`mySolution.OptimalStateGridVector`, `mySolution.OptimalOutputGridVector`) - are contained in the `colibry.Solution` object `mySolution`.

The solution of the counter optimization problem can be visualized using the `colibry.Visualizer` object. It is found that the maximum lateral load factor attainable within 8 seconds is 0.278.

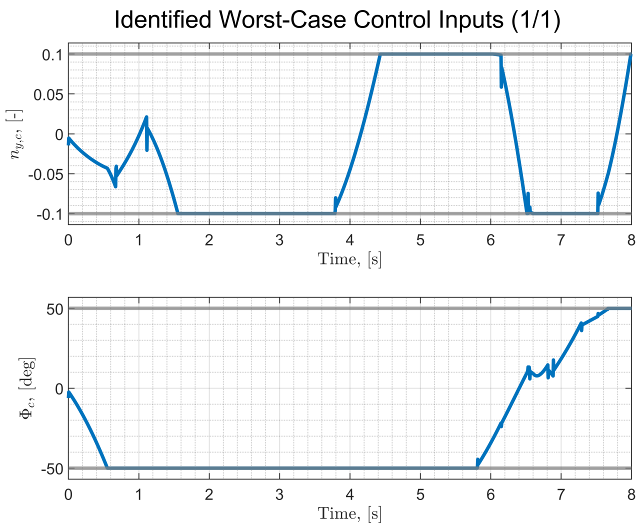

Figure 2.1: Worst-Case Control Inputs

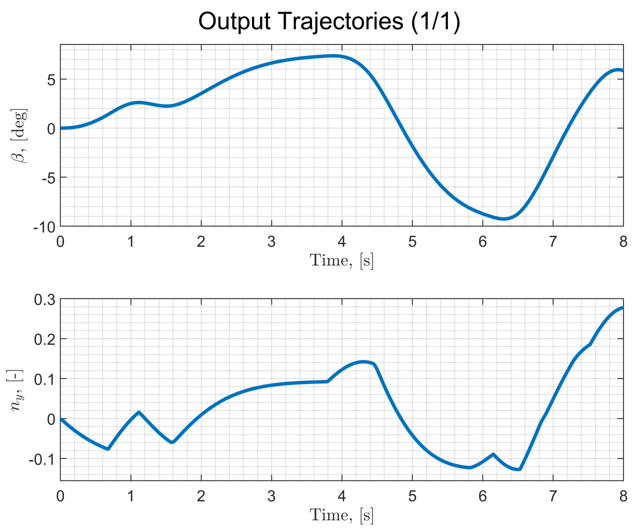

Figure 2.2: Worst-Case Outputs

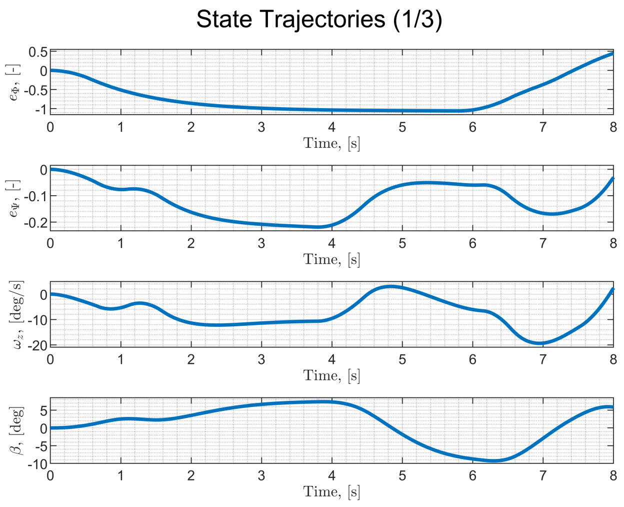

Figure 2.3: Worst-Case States 1

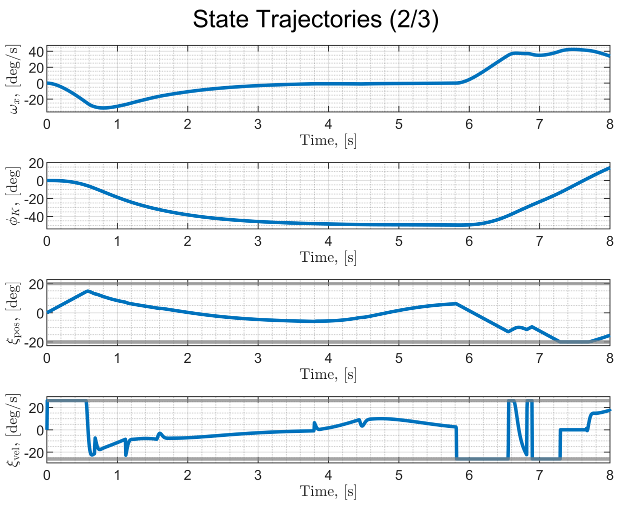

Figure 2.4: Worst-Case States 2

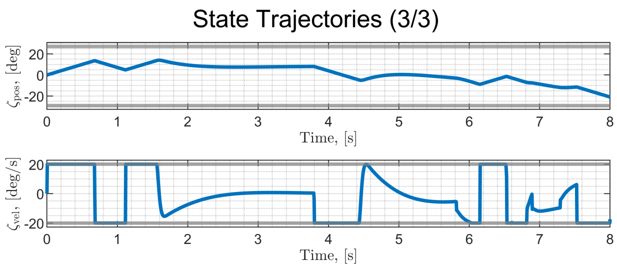

Figure 2.5: Worst-Case States 3

3 Basic Concepts

In the following the demonstration example `main.m` in the subfolder `demo/01-linear-optimalcontrol` is used to explain the definition, set-up, and solution of a counter optimization problem in COLIBRY.

3.1 Definition, Setup, and Solution of a Counter Optimization Problem in COLIBRY

As a first step a `colibry.linearoptimalcontrol.Problem` instance is created.

```matlab
myProblem = colibry.linearoptimalcontrol.Problem('Name', 'Maximum Lateral Load
    Factor - Control Input Analysis')...
```

This particular subclass of the superclass `colibry.Problem` requires the definition of the following problem components: - states - parameters (optionally) - controls - outputs - model - final time - cost - options - method-specific solver arguments

Additionally, the problem instance allows for the specification of a `Name`.

In the script `main.m` the problem components are defined as follows:

```matlab
%% Create problem instance
myProblem = colibry.linearoptimalcontrol.Problem('Name', 'Maximum Lateral Load
    Factor - Control Input Analysis')...
    .addState('beta', 'PlotName', '$\beta$', 'PlotUnit', 'deg', 'PlotConv', @
        rad2deg) ...
    .addState('phi', 'PlotName', '$\phi$', 'PlotUnit', 'deg', 'PlotConv', @
        rad2deg) ...
    ... other rigid body and controller states
    .addState('xi_pos', 'Bounds', deg2rad([-20, 20]), 'PlotName', '$\xi_{pos}$'
        , ...
    'PlotUnit', 'deg', 'PlotConv', @rad2deg) ...
    .addState('xi_vel', 'Bounds', deg2rad([-26, 26]), 'PlotName', '$\xi_{vel}$'
        , ...
    'PlotUnit', 'deg/s', 'PlotConv', @rad2deg) ...
    ... other (bounded) states for control surfaces
    .addOutput('n_y', 'PlotName', '$n_y$') ...
    .addControl('n_y_c', 'Bounds', [-0.10, 0.10], 'PlotName', '$n_{y,c}$') ...
    ... other controls
    .setModel(myModel) ...
    .setFinalTime(8) ...
    .setObjective(myCost) ...
    .setOption('StepSize', 0.005) ...
    .setOption('DiscretizationMethod', 'BackwardEuler') ...
    ... other options
    .setSolverArgument("MaxTime", 10) ...
    ... other solver arguments
```

Problem components such as the final time, step size, the discretization method, or other solver options can be specified directly. The cost and model instances, however, constitute implementations against predefined abstract class interfaces.

```
%% Create model instance
myModel = AircraftModel();
%% Create cost instance
myCost = colibry.objective.MayerCost("max")...
    .addComponentAndWeight("n_y", 1);
```

The linear model of the aircraft is implemented in `AircraftModel`. This class constitutes a specific implementation of the abstract superclass `colibry.model.Linear`, which governs the model interface to linear systems in COLIBRY. The function `provideLinearModel` implements a routine that provides the linearized model for a particular parameter vector.

The definition of the objective function is accomplished through the instantiation of an objective of type `colibry.objective.MayerCost`, which is the standard class for Mayer-type objective functions in COLIBRY. By default, the toolbox considers objectives as cost functions and thus minimizes the returned objective value. Because in this example, however, the lateral load factor is to be maximized, `"max"` is passed to the constructor. The objective is specified by adding components with corresponding weights to the cost object. For these components, outputs with matching identifiers need to be specified in the problem definition. COLIBRY will then consider the weighted sum of the specified outputs at the final time point in the cost function of the optimal control problem. In our illustrative example, only a single quantity, the lateral load factor n_y, is under investigation. Hence, one component (`"n_y"`) with unit weight (`1.0`) is added to the object.

Having set up the problem, we can investigate the pole-zero and step characteristics of the linear system. To this end we instantiate a a `colibry.linearoptimalcontrol.Inspector` object and call its methods `plotPoleZeroPlot()` and `plotStepFunction()`:

```
myInspector = colibry.linearoptimalcontrol.Inspector();
myInspector.plotPoleZeroPlot(myProblem);
myInspector.plotStepFunction(myProblem);
```

After having set up and inspected the problem, we start the counter optimization analysis by calling:

```
%% Solve problem and obtain solution
mySolution = solve(myProblem);
```

This triggers the solution of the counter optimization problem. In the case of the Linear Optimal Control-based approach, the counter optimization problem is transcribed into a Linear Programming problem and subsequently solved using numerical optimization.

All results are contained in the `colibry.linearoptimalcontrol.Solution` object `mySolution` . For the linear optimal control-based analysis, the solution comprises the following components: - `OptimalCost` - contains the maximum value of the cost function as well as first-order sensitivity information - `OptimalStateGridVector` - optimum values of the state variables over time - `OptimalControlGridVector` - optimum values of the control variables over time - `OptimalOutputGridVector` - optimum values of the output variables over time - `SwitchingFunctionGridVector` - the values of the switching function over time - `CostateGridVector` - the values of the costate variables over time - `TimeToSolve` - the required runtime of the counter optimization analysis

Additional information, e.g., the raw solver output, can be accessed by calling `returnInternalSolution ()`:

```
debugSolution = mySolution.returnInternalSolution();
```

Note that the content and structure of the internal solution may change in future releases.

To facilitate the interpretation of the solution, each `colibry.Problem` implements a `colibry. Visualizer` object that can be used for the visualization of the obtained solution.

```
%% Visualize solution
myVisualizer = mySolution.visualize();
myVisualizer.plotControls();
myVisualizer.plotStates();
myVisualizer.plotOutputs();
myVisualizer.plotSwitchingFunctions();
myVisualizer.plotSensitivities();
```

A pdf-report of the solution to the counter optimization problem can be obtained by means of the `colibry.ReportGenerator`.

```
%% Generate pdf-report
mySolution.report();
```

At this point, you are able to define, set-up, and solve counter optimization problems using the Linear Optimal Control-based approach. Because the modeling syntax used in COLIBRY is designed to be homogeneous across different counter optimization methods, you should be able to quickly apply other counter optimization methods yourself. This can be confirmed by checking out the other demonstration examples located in the folder `demo/`.

4 Overview of the Implemented Counter Optimization Methods

This release of COLIBRY includes five counter optimization methods, referred to as library entries. Each of the methods is briefly summarized in the following sections. More detailed information about the implemented approaches is available in the literature.

4.1 Linear Optimal Control-based Analysis

An optimal control-based method that identifies worst-case control inputs to quasi-linear systems [3]. Quasi-linear systems are understood here as linear models subject to additional state constraints. These constraints can be used, for example, to model physical limitations such as actuator position and rate bounds. Moreover, by means of post-optimal sensitivity analysis, the influence of parameters on the worst-case value of the testing criterion can be computed efficiently [5]. For an application of this method a linearization routine, capable of computing the linearized system, input, and output matrix for particular values of the parameters, needs to be provided by the user. The solution obtained from this type of analysis is globally optimal.

4.2 Bi-level Linear Optimal Control-based Analysis

By leveraging a two-layered optimization scheme, this method computes the worst-case combination of control inputs and system parameters for quasi-linear systems [4]. On the one side, the lower level in this scheme performs linear optimal control-based analyses for fixed values of the parameters. From a user perspective, the same requirements for the application of this method as for the linear optimal control-based analysis apply. On the other side, the upper level solves a (typically low-dimensional) parameter search problem. Several optimization methods such as gradient-based or global schemes can be chosen by the user for the upper level. Note that despite their efficiency and the ability to check the optimality conditions for local solution candidates, gradient-based methods do not guarantee the identification of globally optimal solutions for the parameter search. Similarly, global methods are to be understood here in the sense that in practical applications these methods can increase the chance of identifying a globally optimal solution. The same holds for the use of global optimization methods in the single shooting-based analysis.

4.3 Nonlinear Optimal Control-based Analysis

This method implements an interface for the application of direct optimal control methods. In the current version of COLIBRY this interface maps to *FALCON.m* [6] (https://www.fsd.lrg.tum.de/software/falcon-m/). Using *FALCON.m*, the nonlinear optimal control problem is transcribed to a discretized version that can be efficiently solved using sparse Nonlinear Programming (NLP) solvers. Under this approach, the optimization of control inputs and parameters can be treated simultaneously. The *FALCON.m* framework and its dependencies, such as additional NLP solvers, represent a (free) software dependency outside COLIBRY. For the application of this method, the closed-loop system under test needs to be implemented as a model understood by *FALCON.m*. In particular, that requires implementing a procedure for the efficient computation of partial derivatives. Despite there being no guarantee of finding a globally optimal solution, the necessary and sufficient conditions of optimality can be checked for local solution candidates.

4.4 Single Shooting-based Analysis

This counter optimization method uses repeated simulation of the SUT to identify detrimental combinations of time-varying control inputs and static parameters. Prerequisite for an application of this method in practice is the implementation of an efficient simulation routine, which then is used within an optimization routine. At the time of this publication, COLIBRY primarily uses the *MATLAB Global Optimization Toolbox* to solve the underlying, typically nonlinear, black-box-type optimization problem. Recalling the remark in the section about the Bi-level Linear Optimal Control-based analysis, the use of global optimization schemes does not guarantee globally optimal solutions.

4.5 Learning-based Analysis

Utilizing RL techniques, this method trains a Double Deep Q-Network [8] agent to identify the worst-case sequence of time-varying system inputs [9]. Because the training of the RL agent constitutes a repeated interaction between the agent and the non-simplified SUT, this counter optimization approach is particularly well suited for the analysis of highly complex systems. However, similar to the single shooting-based approach, this library entry does not provide a guarantee for global optimality.

5 The COLIBRY User Interface

5.1 Model Interfaces

The following paragraphs explain the abstract model interfaces used in COLIBRY.

5.1.1 colibry.model.Linear

This abstract class interface constitutes the model interface to linear systems.

In order to implement the interface to a linear system, create a new class that inherits from `colibry.model.Linear`. Then, implement the abstract method `provideLinearModel()` by implementing a routine that returns the linearized model `A`, `B`, `C` and its initial condition `x_0`, `u_0`, `y_0` for incoming parameters `p`.

The interface of `provideLinearModel()` is defined as follows:

```
provideLinearModel(self, A, B, C, x_0, u_0, y_0, p);
% Provide the linearized model 'A', 'B', 'C' and its initial condition 'x_0', '
    u_0', 'y_0' for parameters 'p'.
%
% Note: The results should be stored in the handle objects 'A', 'B', 'C', 'x_0
    ', 'u_0', and 'y_0'.
%
% Arguments:
%   A    -  (:, :) colibry.Scalar, linear system matrix.
%   B    -  (:, :) colibry.Scalar, linear input matrix.
%   C    -  (:, :) colibry.Scalar, linear output matrix.
%   x_0  -  (:, 1) colibry.Scalar, reference state vector.
%   u_0  -  (:, 1) colibry.Scalar, reference control vector.
%   y_0  -  (:, 1) colibry.Scalar, reference output vector.
%   p    -  (:, 1) colibry.Scalar, parameters.
```

Note that all user interfaces in COLIBRY have the following argument structure: (self, outputs, inputs). In case of `provideLinearModel()`, the outputs comprise the linear model (`A`, `B`, `C`) and its initial condition (`x_0`, `u_0`, `y_0`), while the inputs comprise the parameters (`p`). All arguments are arrays of type `colibry.Scalar`. `colibry.Scalar` objects are handle classes used for storing scalar data.

As demonstrated in the example `demo/01-linear-optimalcontrol/AircraftModel.m`, an implementation against `provideLinearModel()` must “fill” the outputs `A`, `B`, `C`, `x_0`, `u_0`, `y_0` with data obtained from computing the linear system for the incoming parameter values `p`.

5.1.2 colibry.model.Nonlinear

This abstract class interface constitutes the model interface to nonlinear systems and requires the implementation of two abstract methods: `getInitialCondition()` and `evaluateJacobian()`.

```
getInitialCondition(self, x0Bounds);
% Provide the initial condition for the states by specifying 'x0Bounds'.
%
% Note: The result should be stored in the handle object 'x0Bounds'.
%
% Arguments:
%   x0Bounds  -  (:, 2) colibry.Scalar, lower and upper bounds of the states.

evaluateJacobian(self, xDot, y, jacXDot, jacY, x, u, p);
% Provide the state derivative 'xDot', the output 'y', and the gradients '
    jacXDot' and 'jacY'.
%
% Note: The results should be stored in the handle objects 'xDot', 'y', '
    jacXDot', and 'jacY'.
%
% Arguments:
%   xDot     -  (:, 1) colibry.TimeSeries, state derivative.
%   y        -  (:, 1) colibry.TimeSeries, output.
%   jacXDot  -  (:, 1) colibry.TimeSeries, gradient of state derivative.
%   jacY     -  (:, 1) colibry.TimeSeries, gradient of output derivative.
%   x        -  (:, 1) colibry.TimeSeries, states.
%   u        -  (:, 1) colibry.TimeSeries, controls.
%   p        -  (:, 1) colibry.Scalar, parameters.
```

Similar to the linear model interface, the nonlinear interface requires users to implement routines that “fill” the outputs of the respective method. Thus, `getInitialCondition()` requires the user to store the lower and upper bounds of the states in the handle object `x0Bounds`. Analogously, the method `evaluateJacobian()` requires the user to implement a routine that computes and stores the state derivative, the output, and the gradients of the state derivative and the output in the handle objects `xDot`, `y`, `jacXDot`, and `jacY`.

Exemplary implementations against the nonlinear model interface `colibry.model.Nonlinear` are provided in `demo/05-nonlinear-optimalcontrol/MRACModelFalcon.m` and `demo/05-nonlinear-optimalcontrol/MRACModelManual.m`.

5.1.3 colibry.model.Simulation

This abstract class interface for a simulation model requires the implementation of the abstract method `simulate()`.

```
simulate(self, x, y, u, p, tsim)
% Provide simulation results 'x' and 'y' for controls 'u' and parameters 'p'.
%
% Note: The results should be stored in the handle objects 'x' and 'y'.
%
% Arguments:
%   x     -  (:, 1) colibry.TimeSeries, states.
%   y     -  (:, 1) colibry.TimeSeries, outputs.
%   u     -  (:, 1) colibry.TimeSeries, controls.
%   p     -  (:, 1) colibry.Scalar, parameters.
%   tsim  -  (:, 1) colibry.Scalar, final simulation time.
```

An example of an implementation against the model interface `colibry.model.Simulation` is provided in `demo/02-singleshooting/MRACModel.m`.

5.1.4 colibry.model.Discrete

This abstract class interface requires the implementation of two abstract methods: `init()` and `step()`
.

```
init(self, xInit, out, p)
% Initialize the system for parameters 'p' and return the initial state 'xInit'
% as well as additional output information 'out' at the initial time.
%
% Note: The results should be stored in the handle objects 'xInit' and 'out'.
%
% Arguments:
%   xInit  -  (:, 1) colibry.Scalar, vector of states after initialization.
%   out    -  (:, :) colibry.Scalar, output information to be used in the
    reward function.
%   p      -  (:, 1) colibry.Scalar, vector of static parameters.

step(self, xNext, out, xPrev, a, dt)
% Simulate the system starting from state 'xPrev' for time 'dT' using actions '
    a'
% as control inputs. Return the next state 'xNext' as well as additional output
% information 'out' containing information obtained throughout the performed
    simulation.
%
% Note: The results should be stored in the handle objects 'xNext' and 'out'.
```

```
%
% Arguments:
%   xNext  -  (:, :) colibry.Scalar, vector of states after step.
%   out    -  (:, :) colibry.Scalar, output information to be used in the
    reward function.
%   xPrev  -  (:, 1) colibry.Scalar, vector of states prior to step.
%   a      -  (:, 1) colibry.Scalar, actions to be used as control inputs.
%   dt     -  (1, 1) colibry.Scalar, simulation time of step.
```

An example of an implementation against the model interface `colibry.model.Discrete` is provided in `demo/04-learning/MRACModel.m`.

5.2 Objective Interfaces

The following paragraphs explain the abstract objective interfaces used in COLIBRY.

5.2.1 colibry.objective.Cost

`colibry.objective.Cost` constitutes an abstract class interface for user-implemented cost functions.

In order to implement a custom cost function, create a new class that inherits from `colibry.objective.Cost`. Then, implement the abstract method `evaluate()` by implementing a routine that computes the value of the cost function `c`.

The interface of `evaluate()` is defined as follows:

```
evaluate(self, c, x, y, u, p)
% Compute and return the cost function value 'c'.
%
% Note: The result should be stored in the handle object 'c'.
%
% Arguments:
%   c  -  (1, 1) colibry.Scalar, cost function value.
%   x  -  (:, 1) colibry.TimeSeries, states.
%   y  -  (:, 1) colibry.TimeSeries, outputs.
%   u  -  (:, 1) colibry.TimeSeries, controls.
%   p  -  (:, 1) colibry.Scalar, parameters.
```

An example of a custom cost function, implemented against the abstract interface `colibry.objective.Cost` is provided in `demo/02-singleshooting/MRACCost.m`.

5.2.2 colibry.objective.Reward

This class constitutes an abstract interface for user-implemented reward functions.

An implementation of a custom reward function is analogue to that of a custom cost function. First, create a new class that inherits from `colibry.objective.Reward`. Second, implement the abstract method `evaluate()` by implementing a routine that computes the value of the reward function `r`.

The interface of `evaluate()` is defined as follows:

```
evaluate(self, r, out, iter, done)
% Given output information 'out', collected thorughout the preceeding simulation steps,
% and information whether the episode is terminal 'done', this method computes the value
% of the reward 'r'.
%
% Note: The result should be stored in the handle object 'r'.
%
% Arguments:
%   r      -  (1, 1) colibry.Scalar, reward function value.
%   out    -  (:, 1) colibry.TimeSeries, output information collected
%                    throughout the course of the episode.
%   iter   -  (1, 1) integer, iteration index.
%   done   -  (1, 1) logical, true if the episode is terminal.
```

An example of a custom reward function, implemented against the abstract interface `colibry.objective.Reward` is provided in `demo/04-learning/MRACReward.m`.

6 Contributors

Contributors to this project are:

- David Braun (Maintainer)
- Johannes Diepolder
- Florian Schwaiger
- Joseph Z. Ben-Asher
- Florian Holzapfel

References

[1] C. Fielding, V. Andras, B. Samir, and S. Michiel, *Advanced techniques for clearance of flight control laws*. Springer Berlin Heidelberg, 2002.

[2] V. Andras, H. Anders, and P. Guilhem, *Optimization based clearance of flight control laws*. Springer Berlin Heidelberg, 2012.

[3] A. Herrmann and J. Ben Asher, "Flight control law clearance using optimal control theory," *Journal of Aircraft*, vol. 1, pp. 1–15, Oct. 2015, doi: 10.2514/1.C033517.

[4] J. Diepolder, J. Z. Ben-Asher, and F. Holzapfel, "Flight control law clearance using worst-case inputs under parameter uncertainty," *Journal of Guidance, Control, and Dynamics*, vol. 43, no. 10, pp. 1967–1974, 2020, doi: 10.2514/1.G005236.

[5] J. Diepolder, "Optimal control based clearance of flight control laws," Dissertation, Technische Universität München, München, 2021.

[6] M. Rieck *et al.*, "FALCON.m user guide: Version 1.27." Institute of Flight System Dynamics, Technical University of Munich, 2022, [Online]. Available: http://www.falcon-m.com.

[7] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi: 10.1038/nature14236.

[8] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning." arXiv, 2015, doi: 10.48550/ARXIV.1509.06461.

[9] D. Braun, R. Steffensen, A. Steinert, and F. Holzapfel, "Counter optimization-based testing of flight envelope protections in a fly-by-wire control law using deep q-learning," 2022.