The Python Control Systems Library (python-control)

Sawyer Fuller, Ben Greiner, Jason Moore, Richard Murray, René van Paassen, Rory Yorke
https://python-control.org

***Abstract*— The Python Control Systems Library (python-control) is an open source set of Python classes and functions that implement common operations for the analysis and design of feedback control systems. In addition to support for standard LTI control systems (including time and frequency response, block diagram algebra, stability and robustness analysis, and control system synthesis), the package provides support for nonlinear input/output systems, including system interconnection, simulation, and describing function analysis. A MATLAB compatibility layer provides an many of the common functions corresponding to commands available in the MATLAB Control Systems Toolbox. The library takes advantage of the Python "scientific stack" of Numpy, Matplotlib, and Jupyter Notebooks and offers easy interoperation with other category-leading software systems in data science, machine learning, and robotics that have largely been built on Python.**

I. INTRODUCTION

The Python Control Systems Library (python-control) is a Python package that implements basic operations for analysis and design of feedback control systems. The package was created in 2009, shortly after the publication of Feedback Systems (FBS) by Åström and Murray [1]. The initial goal of the project was to implement the operations needed to carry out all the examples in FBS. A primary motivation for the creation of the python-control library was the need for open-source control design software built on the Python general-purpose programming language. The "scientific stack" of NumPy, SciPy, and Matplotlib provide fast and efficient array operations, linear algebra and other numerical functions, and plotting capabilities to Python users. Python-control has benefited from this foundation, using, e.g., optimization routines from SciPy in its optimal control methods, and Matplotlib for Bode diagrams.

The scientific stack is fast-moving, however, and the python-control package has had to keep up with changes. One example is Matplotlib moving away from a 1990s MATLAB-like plotting paradigm, characterized by global state (e.g., the current figure) to one in which Matplotlib library users are encouraged to more directly manage the figures, axes, etc., currently in use. Another example is the "soft" deprecation of NumPy's matrix class, which was used in python-control's linear-algebra-heavy code.

The Python Control Systems Library is one of many open source tools that are available on different platforms. The GNU Octave application [3] is mainly compatible with MATLAB's command line interface, allowing rapid conversion of code from MATLAB to an open source alternative. Scilab [7] is a free and open source software for engineers and scientists, also mainly compatible with MATLAB and provides a rich, graphical interface. JuliaControl [4] is an emerging open source effort that builds on the numerically robust Julia language. The primary difference between these libraries is the underlying programming language. Several other Python-based packages were under development around the same time as python-control was started, including packages by Roberto Bucher [2] and Ryan Krauss [5], both of whom have contributed ideas and code to the python-control library.

An early decision in developing the package was to make use of the SLICOT library of functions [8], which provides a set of FORTRAN subroutines for carrying out control computations. The use of SLICOT was enabled by the existence of the slycot library created by Enrico Avventi [9], which provided a set of Python wrappers around the FORTRAN code. By making use of slycot, it was possible to implement many standard control functions quickly and easily. SLICOT was designed to be highly efficient, numerically stable, and accurate, allowing many python-control functions to inherit these properties. As of 2019, slycot is available on Windows, Mac, and Linux in binary form through the Anaconda software distribution system and conda-forge [10]. This removes the need to install and run a FORTRAN compiler, broadening accessibility by simplifying the installation process for most users.

We are encouraged by growing usage numbers and worldwide adoption. According to `pypistats.org`, downloads using `pip` were 47,302/month in May 2021. The `condastats` tool provides historical download numbers for Anaconda; they were 3686, 1969, 987, 315, and 71 for python-control during the month of June the past five years, indicating a download rate that is approximately doubling each year. This growth is partially attributable to being open source, which makes python-control accessible to people in industry, hobbyists, and educators who may not want to pay for proprietary software. Users are also freely able to modify the code to suit specific "long-tail" uses. But perhaps more important to its success is that python-control is written in Python, a general-purpose language that has become a *de facto* language for science. Leading libraries in machine learning and data science, e.g. Pandas, TensorFlow, and PyTorch, and in robotics, e.g. Robot Operating System and the Robotics Toolkit, are written in Python. This is complemented by wide availability of purpose-built libraries for graphics and user interfaces, to name a few. This facilitates easy migration and interoperation between libraries and domains if desired. Integration with Jupyter notebooks leverages elements of the broader open source software community to allow for a simple and intuitive design environment.

The remainder of this article provides a brief overview of the python-control package. While we indicate the calling structure of the code and include a few simple examples, our intent here is not to provide a complete guide or tutorial to using python-control, but rather to give a high level overview that provides a flavor of what is available and documents some of our design decisions. More detailed documentation is available at `http://python-control.org`.

II. PACKAGE STRUCTURE AND BASIC FUNCTIONALITY

The python-control package implements an inheritance hierarchy of dynamical system objects. For the most part, when two systems are combined in some way through a mathematical operation, one will be promoted to the type that is the highest of the two. Arranged in order from most to least general, they are:

- `InputOutputSystem`: Input/output system that may be nonlinear and time-varying
  - `InterconnectedSystem`: Interconnected I/O system consisting of multiple subsystems
  - `NonlinearIOSystem`: Nonlinear I/O system
  - `LinearICSystem`: Linear interconnected I/O systems
  - `LinearIOSystem`: Linear I/O system
- `LTI`: Linear, time-invariant system
  - `FrequencyResponseData`: Frequency response data systems
  - `StateSpace`: State space systems
  - `TransferFunction`: Transfer functions

Each can be either discrete-time, that is, $x(k+1) = f(x(k),u(k))$; $y(k) = g(x(k),u(k))$ or continuous time, that is, $\dot{x} = f(x,u)$; $y = g(x,u)$. A discrete-time system is created by specifying a nonzero 'timebase' dt when the system is constructed:

- $dt = 0$: continuous time system (default)
- $dt > 0$: discrete time system with sampling period dt
- $dt =$ `True`: discrete time with unspecified sampling period
- $dt =$ `None`: no timebase specified

Linear, time-invariant systems can be interconnected using mathematical operations `+`, `-`, $*$, and `/`, as well as the domain-specific functions `feedback`, `parallel` (`+`), and `series` ($*$). Some important functions for `LTI` systems and their descriptions are given in Table I. Other categories of tools that are available include model simplification and reduction tools, matrix computations (Lyapunov and Ricatti equations), and a variety of system creation, interconnection and conversion tools. A MATLAB compatibility layer is provided that has functions and calling conventions that are equivalent to their MATLAB counterparts, e.g. `tf, ss, step, impulse, bode, margin, nyquist` and so on. A complete list is available at `http://python-control.org`.

III. EXAMPLE

To illustrate the use of the package, we present an example of the design of an inner/outer loop control architecture for the planar vertical takeoff and landing (PVTOL) example in FBS [1]. A slightly different version of this example is available in the python-control GitHub repository.

We begin by initializing the Python environment with the packages that we will use in the example:

```
# pvtol-nested.py - inner/outer design for
#   vectored thrust aircraft
# RMM, 5 Sep 2009 (updated 11 May 2021)
#
# This file works through a control design and
# analysis for the planar vertical takeoff and
# landing (PVTOL) aircraft in Astrom and Murray.

import control as ct
import matplotlib.pyplot as plt
import numpy as np
```

We next define the system that we plan to control (see [1] for a more complete description of these dynamics):

```
# System parameters
m = 4               # mass of aircraft
J = 0.0475          # inertia around pitch axis
r = 0.25            # distance to center of force
g = 9.8             # gravitational constant
c = 0.05            # damping factor (estimated)

# Transfer functions for dynamics
Pi = ct.tf([r], [J, 0, 0])  # inner loop (roll)
Po = ct.tf([1], [m, c, 0])  # outer loop (posn)
```

The control design is performed by using a lead compensator to control the inner loop (roll axis):

```
# Inner loop control design
#
# Controller for the pitch dynamics: the goal is
# to have a fast response so that we can use this
# as a simplified process for the lateral dynamics

# Design a simple lead controller for the system
k_i, a_i, b_i = 200, 2, 50
Ci = k_i * ct.tf([1, a_i], [1, b_i])
Li = Pi * Ci
```

We can now analyze the results by plotting the frequency response as well as the Gang of 4:

```
# Loop transfer function Bode plot, with margins
plt.figure(); ct.bode_plot(Li, margins=True)
plt.savefig('pvtol-inner-ltf.pdf')

# Make sure inner loop specification is met
plt.figure(); ct.gangof4_plot(Pi, Ci)
plt.savefig('pvtol-gangof4.pdf')
```

Figures 1a and b show the outputs from these commands.

The outer loop (lateral position) is designed using a second lead compensator, using the roll angle as the input:

```
# Design lateral control system (lead compensator)
```

TABLE I: Sample functions available in the python-control package.

Frequency domain analysis:

`sys(x[, squeeze])`	Evaluate frequency response of an LTI system at complex frequenc(ies) x
`sys.frequency_response(omega[, squeeze])`	Evaluate frequency response of an LTI system at real angular frequenc(ies) omega
`stability_margins(sysdata[, returnall, ...])`	Calculate stability margins and associated crossover frequencies
`phase_crossover_frequencies(sys)`	Compute frequencies and gains at intersections with the real axis in a Nyquist plot
`bode_plot(syslist[, omega, plot, ...])`	Bode plot for a system
`nyquist_plot(syslist[, omega, plot, ...])`	Nyquist plot for a system
`gangof4_plot(P, C[, omega])`	Plot the "Gang of 4" transfer functions for a system
`nichols_plot(sys_list[, omega, grid])`	Nichols plot for a system

Time domain analysis:

`forced_response(sys[, T, U, X0, transpose, ...])`	Simulated response of a linear system to a general input
`impulse_response(sys[, T, X0, input, ...])`	Compute the impulse response for a linear system
`initial_response(sys[, T, X0, input, ...])`	Initial condition response of a linear system
`step_response(sys[, T, X0, input, output, ...])`	Compute the step response for a linear system
`step_info(sys[, T, X0, input, output, ...])`	Compute step response characteristics
`phase_plot(odefun[, X, Y, scale, X0, T, ...])`	Phase plot for 2D dynamical systems

Other analysis functions and methods:

`sys.dcgain()`	Return the zero-frequency (or DC) gain of an LTI system
`sys.pole()`	Compute poles of an LTI system
`sys.zero()`	Compute zeros of an LTI system
`sys.damp()`	Compute natural frequency and damping ratio of LTI system poles
`pzmap(sys[, plot, grid, title])`	Plot a pole/zero map for a linear system
`root_locus(sys[, kvect, xlim, ylim, ...])`	Root locus plot
`sisotool(sys[, kvect, xlim_rlocus, ...])`	Sisotool style collection of plots inspired by MATLAB

Synthesis tools:

`acker(A, B, poles)`	Pole placement using the Ackermann method
`h2syn(P, nmeas, ncon)`	H_2 control synthesis for plant P
`hinfsyn(P, nmeas, ncon)`	H_∞ control synthesis for plant P
`lqr(A, B, Q, R[, N])`	Linear quadratic regulator design
`lqe(A, G, C, QN, RN, [, N])`	Linear quadratic estimator design (Kalman filter) for continuous-time systems
`mixsyn(g[, w1, w2, w3])`	Mixed-sensitivity H-infinity synthesis
`place(A, B, p)`	Place closed-loop poles

```
a_o, b_o, k_o = 0.3, 10, 2
Co = -k_o * ct.tf([1, a_o], [1, b_o])
Lo = -m * g * Po * Co

# Compute real outer-loop loop transfer function
L = Co * Hi * Po
```

We can analyze the results using Bode plots, Nyquist plots and time domain simulations:

```
# Compute stability margins
gm, pm, wgc, wpc = ct.margin(L)

# Check to make sure that the specification is met
plt.figure(); ct.gangof4_plot(-m * g * Po, Co)

# Nyquist plot for complete design
plt.figure(); ct.nyquist_plot(L)
plt.savefig('pvtol-nyquist.pdf')

# Step response
t, y = ct.step_response(T, np.linspace(0, 20))
plt.figure(); plt.plot(t, y)
plt.savefig('pvtol-step.pdf')
```

Figures 1c and d show the outputs from the `nyquist_plot` and `step_response` commands (note that the `step_response` command only computes the response, unlike MATLAB).

IV. SPECIALIZED FUNCTIONALITY

In addition to basic control functions and MATLAB compatibility, the Python Control Systems Library has some specialized functions that allow analysis of nonlinear feedback control systems.

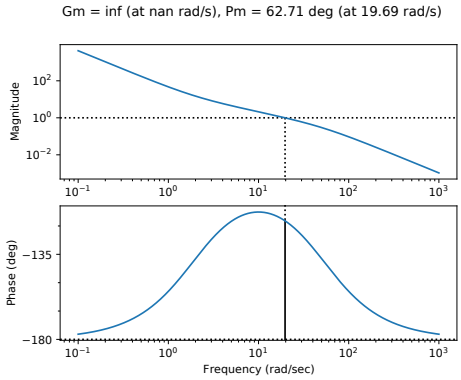
A. Input/output systems

Python-control supports the notion of an input/output system in a manner that is similar to the MATLAB "S-function" implementation. Input/output systems can be combined using standard block diagram manipulation functions (including overloaded operators), simulated to obtain input/output and initial condition responses, and linearized about an operating point to obtain a new linear system that is both an input/output and an LTI system.

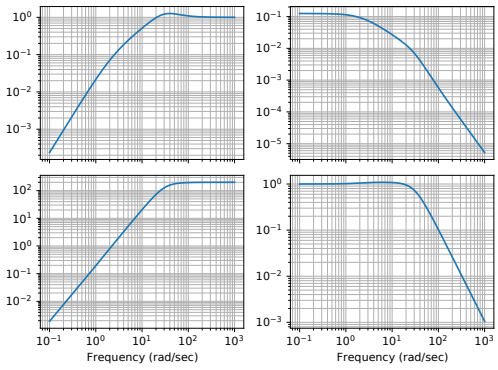
An input/output system is defined as a dynamical system that has a system state as well as inputs and outputs (either inputs or states can be empty). The dynamics of the system can be in continuous or discrete time. To simulate an input/output system, the `input_output_response()` function is used:

```
t, y = input_output_response(io_sys, T, U,
    X0, params)
```

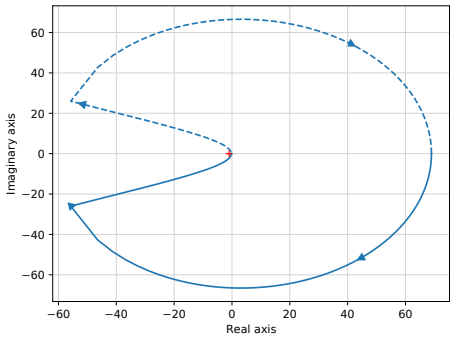
Here, the variable `T` is an array of times and the variable `U` is the corresponding inputs at those times. The output will be evaluated at those times, though the NumPy `interp`



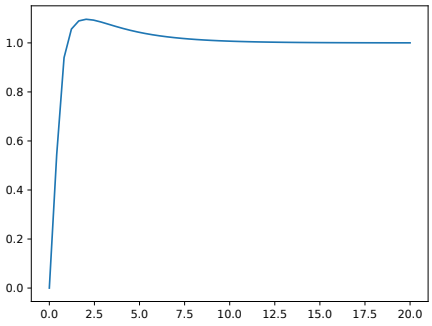
(a) Inner loop, with margins



(b) Gang of 4 for inner loop



(c) Nyquist plot for full system



(d) Step response for full system

Fig. 1: Sample outputs for PVTOL example.

function can be used to interpolate inputs at a finer timescale, if desired.

An input/output system can be linearized around an equilibrium point to obtain a state space linear system. The `find_eqpt()` function can be used to obtain an equilibrium point and the `linearize()` function to linearize about that equilibrium point:

```
xeq, ueq = find_eqpt(io_sys, X0, U0)
ss_sys = linearize(io_sys, xeq, ueq)
```

The resulting `ss_sys` object is a `LinearIOSystem` object, which is both an I/O system and an LTI system, allowing it to be used for further operations available to either class.

Input/output systems can be created from state space LTI systems by using the `LinearIOSystem` class:

```
io_sys = LinearIOSystem(ss_sys)
```

Nonlinear input/output systems can be created using the NonlinearIOSystem class, which requires the definition of an update function (for the right-hand side of the differential or difference equation) and output function (computes the outputs from the state):

```
io_sys = NonlinearIOSystem(updfcn, outfcn,
    inputs=M, outputs=P, states=N)
```

More complex input/output systems can be constructed by using the `interconnect()` function, which allows a collection of input/output subsystems to be combined with internal connections between the subsystems and a set of overall system inputs and outputs that link to the subsystems:

```
steering = ct.interconnect(
    [plant, controller], name='system',
    connections=[['controller.e', '-plant.y']],
    inplist=['controller.e'], inputs='r',
    outlist=['plant.y'], outputs='y')
```

In addition to explicit interconnections, signals can also be interconnected automatically using signal names by simply omitting the `connections` parameter.

Interconnected systems can also be created using block diagram manipulations such as the `series()`, `parallel()`, and `feedback()` functions. The `InputOutputSystem` class also supports various algebraic operations such as $*$ (series interconnection) and + (parallel interconnection).

B. Describing functions

For nonlinear systems consisting of a feedback connection between a linear system and a static nonlinearity, it is possible to obtain a generalization of Nyquist's stability criterion based on the idea of describing functions. The basic concept involves approximating the response of a static nonlinearity to an input $u = Ae^{j\omega t}$ as an output $y = N(A)(Ae^{j\omega t})$, where $N(A) \in \mathbb{C}$ represents the (amplitude-dependent) gain and phase associated with the nonlinearity.

Stability analysis of a linear system $H(s)$ with a feedback nonlinearity $F(x)$ is done by looking for amplitudes A and

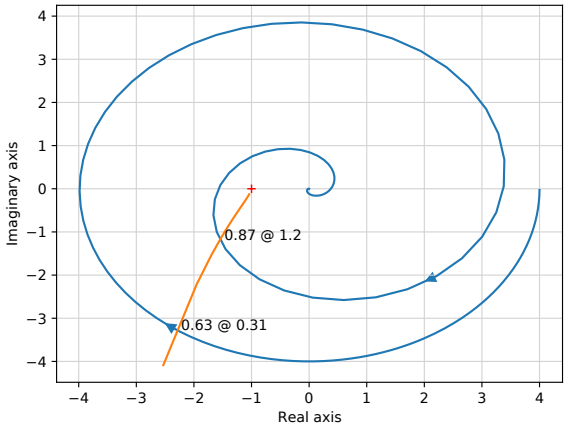


Fig. 2: Polar plot of the frequency response of a describing function

frequencies ω such that

$$H(j\omega)N(A) = -1$$

If such an intersection exists, it indicates that there may be a limit cycle of amplitude A with frequency ω. An example is shown in Figure 2. Describing function analysis is a simple method, but it is approximate because it assumes that higher harmonics can be neglected. For more information, see [1].

C. Optimization-based control

The optimal module provides support for optimization-based controllers for nonlinear systems with state and input constraints.

The optimal control module provides a means of computing optimal trajectories for nonlinear systems and implementing optimization-based controllers, including model predictive control. The basic optimal control problem is to solve the optimization

$$\min_{u(\cdot)} \int_0^T L(x,u)\,dt + V\big(x(T)\big)$$

subject to the constraint

$$\dot{x} = f(x,u), \qquad x \in \mathbb{R}^n, u \in \mathbb{R}^m.$$

Constraints on the states and inputs along the trajectory and at the end of the trajectory can also be specified:

$$\begin{aligned} \text{lb}_i \le g_i(x,u) \le \text{ub}_i, &\qquad i = 1,\ldots,k \\ \psi_i(x(T)) = 0, &\qquad i = 1,\ldots,q. \end{aligned}$$

The python-control implementation of optimal control follows the basic problem setup described here, but carries out all computations in discrete time (so that integrals become sums) and over a finite horizon.

To describe an optimal control problem we need an input/output system, a time horizon, a cost function, and (optionally) a set of constraints on the state and/or input, either along the trajectory and at the terminal time. The optimal control module operates by converting the optimal control problem into a standard optimization problem that can be solved by `scipy.optimize.minimize()`. The optimal control problem can be solved by using the `solve_ocp()` function:

```
res = obc.solve_ocp(sys, horizon, X0,
    cost, constraints)
```

The `sys` parameter should be an `InputOutputSystem` and the `horizon` parameter should represent a time vector that gives the list of times at which the cost and constraints should be evaluated.

The cost function has call signature `cost(t, x, u)` and should return the (incremental) cost at the given time, state, and input. It will be evaluated at each point in the horizon vector. The `terminal_cost` parameter can be used to specify a cost function for the final point in the trajectory.

The constraints parameter is a list of constraints similar to that used by the `scipy.optimize.minimize()` function. Each constraint is a tuple of one of the following forms:

```
(LinearConstraint, A, lb, ub)
(NonlinearConstraint, f, lb, ub)
```

For a linear constraint, the 2D array `A` is multiplied by a vector consisting of the current state `x` and current input `u` stacked vertically, then compared with the upper and lower bound. This constraint is satisfied if

```
lb <= A @ np.hstack([x, u]) <= ub
```

A nonlinear constraint is satisfied if

```
lb <= f(x, u) <= ub
```

By default, constraints are taken to be trajectory constraints holding at all points on the trajectory. The `terminal_constraint` parameter can be used to specify a constraint that only holds at the final point of the trajectory.

The return value for `solve_ocp()` is a bundle object that has the following elements:

`res.success`: `True` if solved successfully
`res.inputs`: optimal input
`res.states`: state trajectory (if `return_x == True`)
`res.time`: copy of the time horizon vector

In addition, the results from `scipy.optimize.minimize()` are also available.

D. Differentially flat systems

A nonlinear differential equation of the form

$$\dot{x} = f(x,u), \qquad x \in \mathbb{R}^n, u \in \mathbb{R}^m$$

is differentially flat if there exists a function α such that

$$z = \alpha(x, u, \dot{u} \ldots, u^{(p)})$$

and we can write the solutions of the nonlinear system as functions of z and a finite number of derivatives

$$\begin{aligned} x &= \beta(z, \dot{z}, \ldots, z^{(q)}) \\ u &= \gamma(z, \dot{z}, \ldots, z^{(q)}). \end{aligned} \tag{1}$$

For a differentially flat system, all of the feasible trajectories for the system can be written as functions of a flat output $z(\cdot)$ and its derivatives. The number of flat outputs is always equal to the number of system inputs. See FBS [1] for a slightly longer (but still brief) description of differentially flat systems.

Differentially flat systems are useful in situations where explicit trajectory generation is required. Since the behavior of a flat system is determined by the flat outputs, we can plan trajectories in output space, and then map these to appropriate inputs. Suppose we wish to generate a feasible trajectory for the nonlinear system

$$\dot{x} = f(x,u), \qquad x(0) = x_0, x(T) = x_f.$$

If the system is differentially flat then

$$x(0) = \beta\big(z(0), \dot{z}(0), \ldots, z^{(q)}(0)\big) = x_0,$$
$$x(T) = \gamma\big(z(T), \dot{z}(T), \ldots, z^{(q)}(T)\big) = x_f,$$

and we see that the initial and final condition in the full state space depends on just the output z and its derivatives at the initial and final times. Thus, any trajectory for z that satisfies these boundary conditions will be a feasible trajectory for the system, using equation (1) to determine the full state space and input trajectories.

The `control.flatsys` package contains a set of classes and functions that can be used to compute trajectories for differentially flat systems. It allows both "simple" trajectory generation (no constraints, no cost function) as well as constrained, optimal trajectory generation (with the same basic structure as the optimal control problems described in the previous section). The primary advantage of solving trajectory generation problems using differentially flat structure, when it applies, is that the all operations are algebraic in nature, with no need to integrate the differential equations describing the dynamics of the system. This can substantially speed up the computation of trajectories.

A differentially flat system is defined by creating an object using the `FlatSystem` class, which has member functions for mapping the system state and input into and out of flat coordinates. The `point_to_point()` function can be used to create a trajectory between two endpoints, written in terms of a set of basis functions defined using the `BasisFamily` class. The resulting trajectory is returned as a `SystemTrajectory` object and can be evaluated using the `eval()` member function.

To create a trajectory for a differentially flat system, a `FlatSystem` object must be created. This is done by specifying the forward and reverse mappings between the system state/input and the differentially flat outputs and their derivatives ("flat flag").

The `forward()` method computes the flat flag given a state and input:

```
zflag = sys.forward(x, u)
```

The reverse() method computes the state and input given the flat flag:

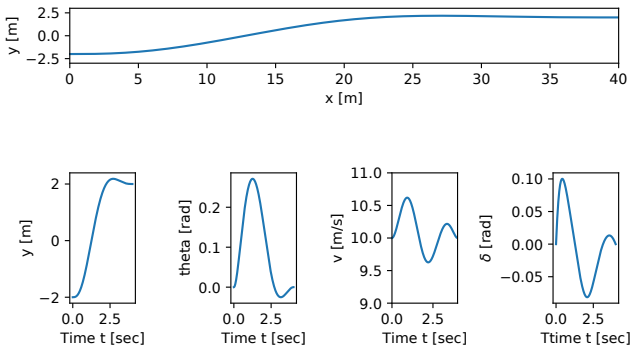


Fig. 3: Trajectory generation using differential flatness.

```
x, u = sys.reverse(zflag)
```

The flag $\bar{z}$ is implemented as a list of flat outputs z_i and their derivatives up to order q_i:

$$\texttt{zflag[i][j]} = z_i^{(j)}$$

The number of flat outputs must match the number of system inputs.

For a linear system, a flat system representation can be generated using the `LinearFlatSystem` class:

```
sys = control.flatsys.LinearFlatSystem(linsys)
```

For more general systems, the `FlatSystem` object must be created manually:

```
sys = control.flatsys.FlatSystem(nstate, ninputs,
    forward, reverse)
```

In addition to the flat system description, a set of basis functions $\phi_i(t)$ must be chosen. The `FlatBasis` class is used to represent the basis functions. A polynomial basis function of the form $1, t, t^2, \ldots$ can be computed using the `PolyBasis` class, which is initialized by passing the desired order of the polynomial basis set:

```
polybasis = control.flatsys.PolyBasis(N)
```

Once the system and basis function have been defined, the `point_to_point()` function can be used to compute a trajectory between initial and final states and inputs:

```
traj = control.flatsys.point_to_point(
    sys, Tf, x0, u0, xf, uf, basis=polybasis)
```

The returned object has class `SystemTrajectory` and can be used to compute the state and input trajectory between the initial and final condition:

```
xd, ud = traj.eval(T)
```

where `T` is a list of times on which the trajectory should be evaluated (e.g., `T = numpy.linspace(0, Tf, M)`).

The `point_to_point()` function also allows the specification of a cost function and/or constraints, in the same format as `solve_ocp()`. An example is shown in Figure 3.

V. DISCUSSION

The python-control package provides a variety of operations for the analysis and design of feedback control systems. In this section we discuss some of the issues, uses, and future directions for the library.

A. Experience using python-control in education

In 2013, the Aerospace Engineering department at the Delft University of Technology transitioned to teaching Python in the first bachelor year. The second-year course in control theory, taught by Dr. René van Paassen, had been using MATLAB for control system design. To provide students with an opportunity to continue their use of Python, provisions were made to accommodate the python-control package in the course. The primary difficulty at the time was installation of python-control and Slycot, which was needed for introducing state space systems. This was initially solved by creating and providing a Windows package to students using Windows, and build and installation instructions for students who wished to use other operating systems. Students had, and still have, the option to choose between using MATLAB and Python; initial adoption of Python and python-control was in the order of 25%. However, in later years, with an increasing number of other courses and assignments in the Bachelor program also transitioning to Python from a variety of programming options (such as spreadsheets and Visual Basic), and the increased ease of installation by integration of python-control and slycot in the conda-forge ecosystem, has led to Python and python-control becoming the de facto choice for most students.

In 2021, Python-control's support for discrete-time systems became sufficient to offer it as an alternative to MATLAB in the University of Washington's Digital Control Systems Design course taught by Dr. Sawyer B. Fuller. Of 29 enrolled students, 11 responded to an anonymous survey at the middle of the quarter. Six of the respondents (55%) opted to use Python-control rather than MATLAB where possible. Student satisfaction with the software was encouraging: of the six Python-control users, five indicated they would either "definitely" or "probably" use it for their next control design project if the option was available. Reasons given by these students for using Python-control included that "Python is more versatile and free," and "python can be used more easily with [the Robot Operating System], which is used in most things I am interested in." One student was positive about the package's usability: "The error messages for python were more helpful and well-explained than I thought they would be." A key limiting factor for Python-control was the lack of an alternative to Simulink for simulating interconnected systems. PysimCoder [6], introduced in 2019, is one open-source graphical interface that may satisfy this deficiency.

B. Python versus compiled languages for control design

The interpreted nature of Python has facilitated the scientific Python stack (NumPy, SciPy, Matplotlib, and Jupyter Notebooks), which are well-suited to a control design library. It is natural to ask whether a compiled language could be better. Compiled languages like C can execute as much as 100 times faster than Python. Currently, however, the flexibility and fast iteration afforded by an interpreted language provide distinct advantages. Many control design tasks, such as PID design, can be simulated in less than a second and therefore are more limited by the speed of the user interface than computation time. In fact, the task of executing Python itself and the python-control is also completed within that second. This short period, which is characteristic of interpreted languages, facilitates rapid, exploratory cross-domain investigations that are the cornerstone of design. No compilation also greatly aids development of the library itself. Furthermore, execution speed is faster than a 100-fold reduction because most numerical computation is performed in low-level compiled libraries such as Numpy. Further speedups are possible by compiling heavily-executed sections using Cython or Numba. Julia is an emerging numerical language that includes a just-in-time (JIT) compiler. This requires compiling at startup, requiring 10–30 seconds to plot a system step response in `ControlSystems.jl` [4], after which execution runs at the speed of a compiled language. As compilation time reduces and functionality expands in Julia, this tradeoff may become more advantageous, especially for compute-heavy nonlinear control (Section IV).

C. Future plans

Desired future functionality includes support for time delays and a SIMULINK-like graphical user environment (building on pysimCoder [6]) available on platforms other than Linux. There is also a nascent effort to build a symbolic version of the library that will work with SymPy.

Acknowledgements. The following individuals have contributed to the python-control package (individuals with 10 or more commits to the repository, ordered by the number of commits): Richard Murray, Ben Greiner, Sawyer Fuller, Clancy Rowley, René van Paassen, Rory Yorke, David de Jong, Anthony De Bortoli, [gonmolina], Mikhail Pak, James Goppert, Arnold Braker, and Scott Livingston.

REFERENCES

[1] K. J. Åström and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2008. Available at `http://fbsbook.org`.

[2] R. Bucher. Practical experiences with Python and Linux RT at the SUPSI laboratory. *IFAC-PapersOnLine*, 52(9):133–138, 2019.

[3] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring. *GNU Octave version 6.2.0 manual: a high-level interactive language for numerical computations*, 2021.

[4] Controlsystems.jl. `http://juliacontrol.github.io/ControlSystems.jl/latest/`. Accessed: 2021-05-14.

[5] R. W. Krauss and W. J. Book. A Python module for modeling and control design of flexible robots. *Computing in Science and Engineering*, 9(3):41–45, 2007.

[6] PysimCoder – SIMULINK like editor for Python. `http://robertobucher.dti.supsi.ch/python/pysimcoder`. Accessed: 2021-05-14.

[7] Scilab. `https://www.scilab.org`. Accessed: 2021-05-13.

[8] SLICOT – A subroutine library in control and systems theory. `http://slicot.org`. Accessed: 2021-05-14.

[9] Slycot – Python wrapper for SLICOT. `https://github.com/python-control/Slycot`. Accessed: 2021-05-14.

[10] Slycot conda-forge distribution. `https://anaconda.org/conda-forge/slycot`. Accessed: 2021-05-14.