# C4DYNAMICS: The Python framework for state-space modeling and algorithm development

Ziv Meri ⬛¹

1 Independent Researcher, Israel

Summary

Dynamic systems play a central role across robotics, aerospace, and guidance, navigation, and control (GNC). The state-space model, introduced by Kalman in the mid-20th century, is the most common way to describe dynamic systems over time using internal variables (Kalman, 1959). While Python provides robust numerical tools, it lacks a dedicated framework specifically designed for state-space modeling. **C4DYNAMICS** bridges this gap by introducing a Python-based platform designed for state-space modeling and analysis. The framework's modular architecture, with "state objects" at its core, simplifies the development of algorithms for sensors, filters, and detectors. This allows researchers, engineers, and students to effectively design, simulate, and analyze dynamic systems. By integrating state objects with a scientific library, *c4dynamics* provides a scalable and efficient foundation for modeling dynamic systems.

Statement of Need

Modeling and simulation of dynamical systems are essential across robotics, aerospace, and control engineering. In these fields, engineers design state-space level algorithms, algorithms that operate directly on the mathematical representation of system states (e.g., position, velocity, or attitude). While Python provides powerful numerical libraries (e.g., NumPy (Harris, 2020), SciPy (Virtanen et al., 2020)) and several domain-specific frameworks (for example, robot simulators and control toolboxes), none directly support low-level algorithm development, where the state vector is explicitly modeled and manipulated.

c4dynamics is designed for engineers who prefer code-based modeling and want to explicitly define the variables encapsulated in the system's state vector. It streamlines mathematical operations (e.g. scalar multiplication or dot products), and data operations (state storage, history retrieval, plotting).

For example, in a guidance or control system, engineers can directly model the position and velocity of a vehicle, apply Kalman filters to estimate its motion, and visualize results. All within a unified, Python-native workflow.

Comparison with Existing Software

Existing tools generally fall into two categories:

1) Block-diagram frameworks (e.g., SimuPy (Margolis, 2017), BdSim (Nevay et al., 2020)) mimic Simulink and simplify model building through graphical interfaces, but they abstract away the state vector and limit direct mathematical manipulation.
2) High-level simulators (e.g., IR-Sim, RobotDART (Chatzilygeroudis et al., 2024)) allow algorithm testing in predefined environments but lack flexibility for low-level system modeling and algorithm design.

Both operate at higher abstraction levels, concealing the underlying state-space formulation. In block-diagram frameworks, users connect functional blocks, while individual state variables remain implicit. In high-level simulators, states are often predefined by the environment (e.g., robot position, velocity) rather than exposed to the user.

In contrast, *c4dynamics* brings the state-space representation back into focus, allowing engineers to define, manipulate, and analyze the system's state vector directly — bridging the gap between physical modeling and algorithm design.

For illustration, consider the modeling and simulation of a pendulum:

- In a block-diagram framework, the user connects integrators and functional blocks, focusing on signal flow rather than the physical meaning of the state variables.

- High-level simulators abstract the system even further: users define parameters in configuration files and observe the resulting motion at the behavioral level, rather than interacting with the underlying mathematical or algorithmic model.

- In a stateful approach, the user explicitly defines the state variables and their initial conditions, performing algorithmic operations — both mathematical and data-related — within the main loop.

c4dynamics adopts this stateful approach and extends it into a modular, Python-native framework for state-space modeling of dynamical systems. Built around explicit state representations and complemented by a scientific library of filters and sensor models, it enables reproducible modeling, testing, and optimization of dynamic systems within the scientific Python ecosystem.

Example

The following example demonstrates how to model a simple pendulum using *c4dynamics*. The state of the pendulum consists of two variables: $X = [\theta, q]$, where θ is the angular displacement (rod angle), and q is the angular rate.

Initial conditions: $X_0 = [50, 0]$ (degrees, degrees per second, respectively).
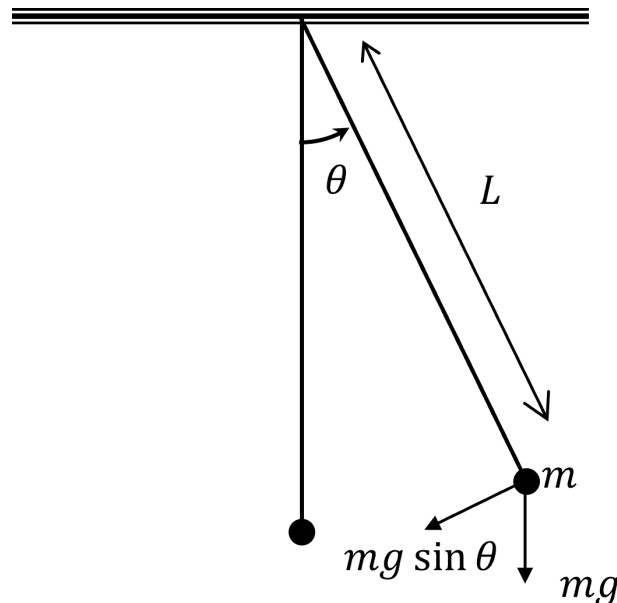

Figure 1: Simplified pendulum configuration.

Meri. (2025). C4DYNAMICS: The Python framework for state-space modeling and algorithm development. *Journal of Open Source Software*, *10*(116), 8776. https://doi.org/10.21105/joss.08776

The pendulum dynamics in state-space form is given by:

$$\dot{\theta} = q$$

$$\dot{q} = \frac{g}{L}\sin\theta$$

Figure 1 shows a schematic of the simple pendulum, and the system parameters are listed below:
- Rod length: $L = 1$ [m] (rigid, massless)
- Gravity: $g = 9.8$ [m/s²]
- Time step: 0.01 [s]
- Simulation duration: 5 [s]
- Integration function: `solve_ivp` (SciPy)

The expected result is an oscillatory motion of the angle $\theta(t)$ in Figure 2, representing the pendulum swinging back and forth.

Import required packages:

```python
from scipy.integrate import solve_ivp
from matplotlib import pyplot as plt
import c4dynamics as c4d
import numpy as np
```

Define the state object and initial conditions:

```python
pend  = c4d.state(theta = 50 * c4d.d2r, q = 0)
```

Run the simulation loop and store the state at each step:

```python
dt = 0.01
for ti in np.arange(0, 5, dt):
  pend.store(ti)
  pend.X = solve_ivp(lambda t, y: [y[1], -9.8 * c4d.sin(y[0])]
            , [ti, ti + dt], pend.X).y[:, -1]
```

Plot the angular displacement history:

```python
pend.plot('theta', scale = c4d.r2d, darkmode = False)
plt.show()
```
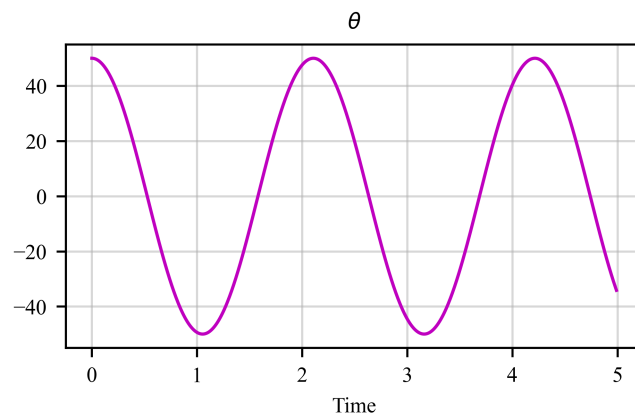

Figure 2: Pendulum angle $\theta(t)$ over time.

Through this example, we see how *c4dynamics* lets users describe a system's physics directly, run the simulation, and visualize results — all within a consistent, state-based framework. The same workflow applies seamlessly to more advanced dynamic models.

References

Chatzilygeroudis, K., Totsila, D., & Mouret, J.-B. (2024). RobotDART: A versatile robot simulator for robotics and machine learning researchers. *Journal of Open Source Software*, *9*(102), 6771. https://doi.org/10.21105/joss.06771

Harris, M., C. R. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Kalman, R. (1959). On the general theory of control systems. *IRE Transactions on Automatic Control*, *4*(3), 110–110. https://doi.org/10.1109/TAC.1959.1104873

Margolis, B. W. (2017). SimuPy: A Python framework for modeling and simulating dynamical systems. *J. Open Source Software*, *2*(17), 396. https://doi.org/10.21105/joss.00396

Nevay, L. J., Boogert, S. T., Snuverink, J., Abramov, A., Deacon, L. C., Garcia-Morales, H., Lefebvre, H., Gibson, S. M., Kwee-Hinzmann, R., Shields, W., & Walker, S. D. (2020). BDSIM: An accelerator tracking code with particle–matter interactions. *Computer Physics Communications*, *252*, 107200. https://doi.org/10.1016/j.cpc.2020.107200

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2