cadCAD Formal Specification

Michael Zargham and Emanuel Lima *

May 2021

Abstract

Complex Adaptive Dynamics Computer Aided Design (cadCAD) is a language for encoding Generalized Dynamical Systems (GDS) as computer programs. As a modeling framework cadCAD is based on best practices from control systems engineering; this framework was designed by Michael Zargham, Markus Koch and Matt Barlin in 2018, in order to bypass the need for powerful closed source software such as Matlab/Simulink in their economic systems design work at BlockScience. The first implementation of cadCAD was developed by Joshua Jodesty, in parallel with the framework development. Python was chosen for the initial implementation due to the wide range of numerical computing tools already available in Python open source data science ecosystem. The Python implementation of cadCAD was Open Sourced in partnership with the Commons Stack in 2019 and has since exploded in use, most notably amongst cryptoeconomic systems designers and token engineers. As of May 2021, there are multiple implementations of cadCAD. This specification was written in part to document the unifying principles for multiple cadCAD implementations spanning Python, Javascript and Rust, and in part to serve as the basis for a new julia implementation of cadCAD for which Emanuel Lima is the lead developer.

Overview

Complex Adaptive Dynamics Computer Aided Design (cadCAD) is a language for encoding Generalized Dynamical Systems (GDS) as computer programs. This formal specification is a minimalist description of what a cadCAD engine does.

> A cadCAD engine is a program which takes an object or record $x_0 \in \mathcal{X}$ and function or method $h : \mathcal{X} \to \mathcal{X}$ and produces a sequence of records x_k such that $x_k = h(x_{k-1})$.

*The Authors would like to thank Joshua Jodesty, Danilo Lessa Bernadineli, Jamsheed Shorish, Matthew Barlin and Christopher Catoya for their contribution to the formalization of cadCAD as systems modeling and simulation language.

State

The *state* of the system is called x. The initial state is x_0 and x_k is the state of the system at timestep k. Each state is a point in the statespace $\mathcal{X}$; said another way, $\mathcal{X}$ is a class of record objects and each x is an instance of that class.

It is common but not required to implement the statespace as a schema containing a list of keys, and types constraining the values that can be stored at that key. Following this convention improves interpretability of the code and associated results. However, alternative implementations could improve computational performance.

Dynamics

The dynamics are the rules about how the state at timestep k is transformed into the state at timestep $k+1$. In order to encode any generalized dynamical system the transform $h : \mathcal{X} \to \mathcal{X}$ is broken down into a *state update pipeline*

$$h = h_{n-1} \circ h_{n-2} \circ \cdots \circ h_1 \circ h_0$$

where each $h_i : \mathcal{X} \to \mathcal{X}$ is called a *partial state update block*. This pipelining allows the dynamics to be described as a sequence of transformations each from $\mathcal{X}$ to $\mathcal{X}$. The indices i denote substeps. Due to this construction there exist intermediate or virtual states $x_{(k,i)} \in \mathcal{X}$. While the intermediate states may be valuable for debugging, they are merely artifacts of the implementation.

Critical to GDS is the notion of separating non-deterministic (strategic or stochastic) inputs from the determinist consequences of those inputs. Therefore each individual h_i may be decomposed as

$$g_i : \mathcal{X} \to \mathcal{U}_i$$
$$f_i : \mathcal{X} \times \mathcal{U}_i \to \mathcal{X}$$

such that

$$h_i(x) = f_i(x, g_i(x)).$$

We sometimes call g_i the blackbox operator and f_i the whitebox operator.

Trajectories

While the input to a cadCAD program is an initial state $x_0 \in \mathcal{X}$ and state update pipeline $h : \mathcal{X} \to \mathcal{X}$ the output of a cadCAD program is a dataset, that data set is a sequence of records x_k conforming to the schema $\mathcal{X}$, which satisfies the relation

$$x_k = h(x_{k-1})$$

starting from x_0. A *Trajectory* is a sequence

$$x_0, x_1, \ldots, x_k, \ldots$$

generated by this process. The length of the sequence is a priori unbounded. A practical implementation must either specify an exit condition, the simplest of which is a maximum timestep at which the iteration will terminate.

In the case where the statespace $\mathcal{X}$ includes keys and types for the values associated with those keys one can cast the trajectory as a table in a database where the record index is the timestep k and the columns are the keys in the statespace. This a common approach as it lends itself well to data analysis once the results are produced.

Termination Conditions

Consider the truncated trajectory $\vec{X}_k = x_0, x_1, \ldots, x_{k-1} \in \mathcal{X}^k$ produced by (x_0, h) where $h : \mathcal{X} \to \mathcal{X}$ and $x_0 \in \mathcal{X}$. In order to complete a trajectory and return results it is necessary to define a termination condition s, such that when s returns True, the iteration is terminated.

$$s : \mathcal{X} \times \mathcal{X} \times \cdots \to \{\text{True, False}\}$$

where the stopping map $s(\vec{X}_k)$ is determines whether another iteration is computed. The stopping map can be decomposed into two components

Maximum Iterations: K, for $k = |\vec{X}_k|$

$$(k < K) \in \{\text{True, False}\}$$

Absorbing States: $A : \mathcal{X} \to$ True, False

$$\mathcal{S} = \{x \in \mathcal{X} | A(x) = \text{True}\} \subset \mathcal{X}$$

Thus the stopping map is given by

$$s(\vec{X}_k) = A(x_{k-1}) \vee (k \geq K)$$

computation is terminated at step k if and only if $s(\vec{X}_k) =$True.

As such, a GDS simulation, or job, is a 3-tuple composed of an initial state, a state update pipeline (that is, the composition of a blackbox and a whitebox operator) and a termination condition.

Parameter Sweep

When running an experiment, it can be desirable to parameterize the dynamics of the simulation in order to understand the changes that different parameters produce on the resulting trajectories. There are three components of the system

model that should be parameterizable: the state, the state update functions and the policies. The expected behavior for an engine implementation would be to set simulations based on the parameters and run these simulations (possibly in parallel, see Implementation Observations). Thus, parameter sweeping is running a different simulation for each of M unique configurations constructed from a Cartesian product of the parameters used to characterize the state, state update functions and policies.

Consider an instance of the GDS job $J = (x_0, h, s)$ which produces a bounded trajectory $\vec{X}_k$ of length $k \leq K$ and stays within the region $\mathcal{X} \setminus \mathcal{S}$. Suppose there is a family of related jobs $\mathcal{J}$, with each job parameterized by a configuration

$$z = z_1, \dots z_m \in \mathcal{Z}_1 \times \cdots \times \mathcal{Z}_m$$

where $\mathcal{Z}_i$ is the domain for parameter $i = 1, \dots, m$, and m is the total number of different parameter types. A typical job $J \in \mathcal{J}$ would then be configured by the parameter vector z, i.e. one may write $J = \text{config}(z) \in \mathcal{J}$.

A parameter sweep $\mathcal{P} \subset \mathcal{J}$ is a finite set of jobs $J = (x_0, h, s)$ constructed by specificing a set of cardinality n_i for each parameter $z_i \in \mathcal{Z} i^{n_i}$, which results in $M = \prod i = 1^m n_i$ distinct jobs indexed $j \in 1, \dots M$ and constructed as

$$J_j = \text{config}(z_j).$$

The parameter sweep is the set of jobs

$$\mathcal{P} = \text{config}(z) | z \in \mathcal{Z}_1^{n_1} \times \cdots \times \mathcal{Z}_m^{n_m} \subset \mathcal{Z}_1 \times \cdots \times \mathcal{Z}_m$$

with finite cardinality $M = |\mathcal{P}|$.

Considerations for Computer Aided Design

Results from multiple trajectories can be compared side-by-side as long as experiment meta-data is maintained. As cadCAD is intended for computation science, additional features for executing repeated experiments are recommended. Useful controlled repeated experiments include:

- Monte Carlo experiments for models with stochastic processes
- including managing seeds to ensure reproducibility
- repeated experiments with the same dynamics but different initial state to test sensitivity to initial conditions
- parametrizing the dynamics by introducing explicit parameters and testing sensitivity to those parameters
- modularization of the decomposed dynamics such that a "g_i" block can be swapped out to account for uncertainty in the modelers assumptions

- modularization of the decomposed dynamics such that "f_i" block can be swapped out to account for a mechanism design change
- read and write APIs allowing the cadCAD engine runtime to interact with other environments

References

[cadCAD] J. Jodesty et al (2018-present), *cadCAD Reference Implementation*; https://github.com/cadCAD-org/cadCAD

[intro] M. Zargham & C. Rice (2019), *Introducing Complex Adaptive Dynamics Computer Aided Design*; BlockScience Medium

[tegg] J. Emmett (2019), *Official cadCAD Open Source Announcement*; Token Engineering Youtube