Behavioral Approach to Dynamical Systems: Implementation of a Julia package

Cédric Simal

December 1, 2022

Abstract

As part of the SOCN Doctoral School on Behavioral Systems Theory, we implemented a small Julia package providing tools from Behavioral Theory for the study of Dynamical Systems. We document the main features of the package, and illustrate them with examples.

Introduction

Notes on Notations

In the notation used throughout this document, we depart slightly from the one used in the course material. In particular, we use the subscript notation $w = [w_0, w_1, \dots]$, rather than the functional one for elements of $(\mathbb{R}^q)^{\mathbb{N}}$. We also abuse the original notation by using $\mathscr{B}_T$ to denote restricted behaviors.

In the rest of this document, unless mentioned otherwise, we implictly assume to be working with an LTI system with n states, m inputs p outputs and lag ℓ. Moreover, we define $q = m + p$ to be the number of observables of the system at our disposition.

1 Elementary Functions

In the making of this package, several utility functions where quickly needed.

Hankel and Multiplication Matrices

The Hankel and Multiplication matrix structures intervene at multiple points throughout the course. Based on the appendix to the exercises, we implemented functions for computing these matrices, and their orthogonal projections for both the scalar and block variants.

```
julia> hankel_matrix(1:5, 2)
2×4 Matrix{Int64}:
 1  2  3  4
 2  3  4  5

julia> x = [1:5;;1:5]'
2×5 adjoint(::Matrix{Int64}) with eltype Int64:
```

```
 1  2  3  4  5
 1  2  3  4  5

julia> hankel_matrix(x, 2)
4×4 Matrix{Int64}:
 1  2  3  4
 1  2  3  4
 2  3  4  5
 2  3  4  5

julia> hankel_matrix([1:5,1:3], 2)
2×6 Matrix{Int64}:
 1  2  3  4  1  2
 2  3  4  5  2  3
```

Our implementation currently uses dense matrices filled out with the correct elements. This is somewhat inefficient due to the structure of e.g. the Hankel matrix. A good direction for future developments would be to implement those matrices in lazy form, which would simply reference the data from the original trajectory without allocating a new array.

Random state space systems and random trajectories

For most of the basic features for Control Systems, rely on the `ControlSystems` package [1], which provides an interface very similar to the Matlab Control Toolbox. One function that was missing however, was `drss`, which generates a random discrete-time state space model. We based ourselves on the implementation from [2]. This function takes as input a triplet (n, m, p) denoting the desired numbers of states, inputs and outputs and generates a random state space model with those dimensions.

```
julia> sys = drss(2,1,1)
ControlSystemsBase.StateSpace{ControlSystemsBase.Discrete{Float64}, Float64}
A =
  1.1495107582998578   0.6155356737880789
 -0.9879718937104236  -0.12389286364849494
B =
 -0.5554201977346566
  1.3123220191281468
C =
 -0.873958753779606  -2.8332765216867277
D =
 0.0

Sample Time: 1.0 (seconds)
Discrete-time state-space model
```

In addition to this, since many algorithms in the course take an arbitrary trajectory of the system being studied as input, we implemented a function `random_trajectory` which takes a state-space system and time to generate a random trajectory of the appropriate length.

```
julia> w = random_trajectory(sys, 3)
2×3 Matrix{Float64}:
  0.106744  0.895347   0.849628
 -1.20746   0.609445  -1.48571
```

Another function in this category is `canonical_perturbation` which takes a triplet (m, p, T) as argument and returns the permutation that maps a vector of the form $[u_1, \ldots, u_T, y_1, \ldots, y_T]'$ to $[u_1 y_1, \ldots, u_T y_T]'$

2 Behavior Representations

The behavior of an LTI system can be represented in multiple ways, some of which were to be implemented as exercises. Broadly speaking, a representation of a behavior $\mathscr{B}$ is a parametrization that enables one to check whether a given trajectory is in $\mathscr{B}$, and to sample trajectories from $\mathscr{B}$. Since the behaviors of LTI systems are shift-invariant subspaces, most representations involve finding a basis of the restricted behaviors $\mathscr{B}_T$ or its orthogonal complement.

State Space and Data-Driven Representation

Based on exercise 1, we implemented functions for constructing a basis of the restricted behavior $\mathscr{B}_T$ of an LTI in state space form in three different ways.

The model based approached, as outlined in the solution provided, involves explicitly constructing the matrix that maps a vector containing the initial condition x_0 and input u to the trajectory vector containing u and the output y, essentially "unrolling" the evolution equations. This is done by the function `ss2BT_modelbased`.

The data-driven approach instead uses a single trajectory of sufficient length of the system to construct its Hankel matrix, whose range spans $\mathscr{B}_T$. The function `ss2BT_hankel` implements this approach. We also provide the function `ss2BT_datadriven`, based on the example given during the course of sampling qT random trajectories. The Hankel approach is obviously much more efficient.

Kernel Representation

The kernel representation of the behavior $\mathscr{B}$ of a system with p outputs, m inputs and lag ℓ is parametrized by an $p \times q(\ell + 1)$ matrix R such that for any trajectory $w \in \mathscr{B}$

$$R(\sigma)w = 0,$$

where $R(\sigma)$ is a polynomial of degree l with matrix coefficients applied the shift operator σ.

For restricted behaviors, the matrix R completely characterizes $\mathscr{B}_{\ell+1}$ and can be used to generate kernel representations of $\mathscr{B}_T$ for $T \geq \ell + 1$ with the Multiplication matrix discussed in the first section.

In the data-driven approach, finding a parameter R is trivial. Given a sufficiently long trajectory w_d, one need simply compute a basis of the left-kernel of $\mathscr{H}_{\ell+1}$, i.e. the kernel of $\mathscr{H}'_{\ell+1}$. This feature is provided by the function `ss2r_datadriven`.

For the model based approach, a little more work is required. If we take the evolution matrix used in the model based approach in exercise 1, for length $\ell + 1$, we obtain

$$w = \begin{bmatrix} u \\ y \end{bmatrix} = \begin{bmatrix} 0 & I_{m(\ell+1)} \\ \mathscr{O}_{\ell+1} & \mathscr{C}_{\ell+1} \end{bmatrix} \begin{bmatrix} x_1 \\ u \end{bmatrix}.$$

Since the lag of the system is equal to ℓ, we know that the observability matrix $\mathscr{O}_\ell$ has rank n, hence we can recover the initial condition x_1 from the first ℓ samples of the trajectory, that is

$$x_1 = \mathscr{O}_\ell^\dagger(\mathscr{C}_\ell u_{1:\ell} - y_{1:\ell}),$$

where $\dagger$ denotes the pseudo-inverse, and the notations $u_{1:\ell}$, $y_{1:\ell}$ denote the vector of the first ℓ samples of u (respectively for y).

We can inject this in the expression for $y_{\ell+1}$ to obtain an expression that only involves the input and outputs

$$\begin{aligned} y_{\ell+1} &= Cx_{\ell+1} + Du_{\ell+1}, \\ &= CA^\ell x_1 + H_{1:\ell} u_{1:\ell} + Du_{\ell+1}, \\ &= CA^\ell \mathscr{O}_\ell(\mathscr{C}_\ell u_{1:\ell} - y_{1:\ell}) + H_{1:\ell+1} u_{1:\ell+1}, \end{aligned}$$

where $H_{1:\ell+1} = [H_{1:\ell}\ D]$ consists of the last p rows of $\mathscr{C}_{\ell+1}$.

We can rearrange this into

$$CA^\ell \mathscr{O}_\ell^\dagger \mathscr{C}_\ell u_{1:\ell} + H_{1:\ell+1} u_{1:\ell+1} - CA^\ell \mathscr{O}_\ell^\dagger y_{1:\ell} - y_{\ell+1} = 0,$$

from which we can obtain the desired parameter matrix R. In our package, this is implemented by the function `ss2r_modelbased`.

3 Calculating the lag and complexity

4 Most Powerful Unfalsified Model

5 Data-Driven Interpolation and Approximation

References

[1] Fredrik Bagge Carlson, Mattias Fält, Albin Heimerson, and Olof Troeng. Controlsystems.jl: A control toolbox in julia. In *Proceedings of CDC 2021*, 12 2021.

[2] Python control systems library. https://github.com/python-control/python-control.