Behavioral Approach to Dynamical Systems: Implementation of a Julia package

Cédric Simal

December 8, 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

This document presents a small Julia package implementing tools from Behavioral Systems Theory, a subdomain of Systems Theory that concerns itself with the study of dynamical systems through their sets of *trajectories*, called their *Behaviors*. In this package, we focus mainly on the behaviors of Linear Time Invariant Systems.

The source code for the package can be found at the following webpage under the MIT open source license.
https://github.com/csimal/BehavioralSystems.jl.

The repository also contains a Jupyter Notebook showcasing the features of the package. Most of the code snippets presented here can be run using it.

As an important notice, while we believe that the algorithms implemented in the package are correct in theory, some of the results we obtain are incorrect, and we could not correct them in time. That being said, we believe fixing these errors would not take a significant effort.

Remarks on Notation

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 $(\mathbb{R}^q)^{\mathbb{N}}$.

For the rest of this document, unless mentioned otherwise, we implicitly 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. We document them here.

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. Passing a vector of matrices to `hankel_matrix` computes the mosaic hankel matrix.

```
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. However, they would still need to be instantiated to compute their SVD, so it would be of marginal utility for this package.

Random state space systems and random trajectories

For most of the basic features for Control Systems, we 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.

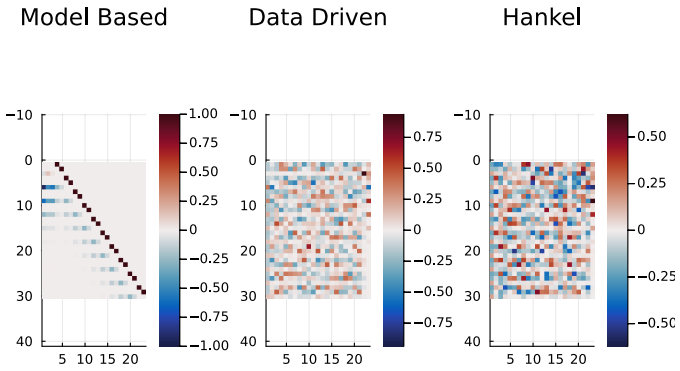

Figure 1: Comparison of the basis of $\mathscr{B}|_T$ obtained through the different methods. The model-based basis is permuted so that its rows are in the same order as the other two.

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 to 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`.

Finally, the function `r2BT` lets us convert from a kernel representation to a basis representation of the behavior.

3 Calculating the lag and complexity

The lag ℓ of a system can be defined as the smallest number of steps that need to be sampled to recover the state from the inputs and output. In other words, ℓ is the smallest number such that the observability matrix $\mathscr{O}_\ell$ has rank n. This exact approach is used in the function `lag_modelbased`.

For the data-driven approach, we provide the function `lag_datadriven` to compute the lag from a trajectory, both when we have prior knowledge of n and m and when we don't.

Our scheme for computing ℓ is as follows. Given estimates of m and n, we know that ℓ must be smaller than n, and we use bisection search to find the smallest l such that rank $\mathscr{H}_l = m*l+n$. When computing the lag of a state-space system, we have access to the exact values of m and n, but when given only a finite trajectory w_d, we estimate m and n by finding the largest l such that rank $\mathscr{H}_l(w_d) <$ rank $\mathscr{H}_{l+1}(w_d)$. This lets us estimate the complexity of the behavior, even based on only a single sample (This is slightly less efficient than using the theoretical bound used for the most powerful unfalsified model).

```
julia> sys = drss(3,2,1);

julia> (lag_modelbased(sys), lag_datadriven(sys))
(3, 3)

julia> w = random_trajectory(sys, 15);

julia> lag_datadriven(w)
3

julia> complexity_datadriven(w)
(2,3,3)
```

4 Most Powerful Unfalsified Model

Computing the Most Powerful Unfalsified Model $\mathscr{B}_{MPUM}(w_d)$ of a finite trajectory w_d is fairly straightforward. From a candidate $\ell_{\max} = \lfloor \frac{T+1}{q+1} \rfloor$, we estimate the corresponding m and n from rank $\mathscr{H}_{\ell_{\max}}(w_d)$ and rank $\mathscr{H}_{\ell_{\max}-1}(w_d)$, and use the `lag_datadriven` function to find the lag. This feature is available through the function `complexity_mpum`.

```
julia> complexity_mpum(w)
(2,3,3)
```

Given the optimal lag ℓ, we can easily find a kernel representation of $\mathscr{B}_{MPUM}(w_d)$ by using the `ss2r_datadriven` function that we previously discussed. This is all wrapped up in a function `most_powerful_unfalsified_model`.

```
julia> most_powerful_unfalsified_model(w)
1×12 adjoint(::Matrix{Float64}) with eltype Float64:
  0.0138837  0.0230249  -0.000869775  …  3.84241e-16  1.40946e-16  0.699837
```

5 Data-Driven Interpolation and Approximation

The final set of features we were able to implement for the package was data interpolation/approximation. This is provided by the function `data_interpolation`. It currently provides the features from exercises 8, 10, 11, 12 and 15. This was made easy by the `Lasso` Julia package [3], which provides an off-the-shelf implementation of Lasso regression with optional weights. Let us also note that exercises 10 and 11 required no additional work, due to the way previous features were implemented (notably `ss2BT_hankel`).

```
julia> sys = drss(3,2,1);

julia> T = 20; T_d = 3*T + 3
63

julia> w_d = random_trajectory(sys, T_d);

julia> w = random_trajectory(sys, T);

julia> w_h = data_interpolation(w_d, w);

julia> norm(w-w_h)
2.903657141709827e-13

julia> w_h2 = data_interpolation(sys, w);

julia> norm(w-w_h2)
9.64816201495751e-13
```

Additionally, we implemented a wrapper around `data_interpolation` for simulating trajectories, called `data_simulation`. It takes a data trajectory w_d as input along with an initial segment w_{ini} and an input segment w_{input} whose outputs are ignored (the number of outputs needs to be provided as well). The general version of the function accepts arrays with `missing`[1] or `NaN` values and interprets them as missing values.

```
julia> w_ini = w[:,1:10]; w_input = w[:,11:end];
```

[1] Julia provides a `Missing` type for the specific purpose of denoting data with missing values.

```
julia> w_s = data_simulation(w_d, w_ini, w_input, p)

julia> norm(w-w_s)
6.654231025147803e-15
```

This can then trivially be used to compute the impulse and step responses in a (mostly) datadriven way. This is provided by the functions `impulse_response` and `step_response`.

```
julia> imp = impulse(sys, 11); # impulse response from ControlSystems

julia> y_imp = impulse_response(sys, 11);

julia> norm(y_imp - imp.y)
1.4684766540114325e-14
```

Discussion

While our implementation covers a basic set of features, it falls somewhat short of the initial goal of implementing the solutions to all the exercises plus a few additional ones. In addition, several of the functions are partially incorrect, notably the model based approaches for `ss2BT` and `ss2r`, which don't span the same subspaces as the data driven representations.

We are nevertheless optimistic that most of these issues should not take undue effort to solve, and that there a few low-hanging fruits in terms of additional features that could be implemented. In particular more features can be built on top of `data_interpolation`, based on the later sections of the exercises.

We conclude with a few comments on further design decisions for the high-level interface of the package.

- Currently, the various functions for computing behavior representations return arrays. It might be useful to wrap those in dedicated types (e.g. `DataDrivenRepresentation`, `KernelRepresentation`), to provide explicit context for what those arrays represent, as well as providing a common interface over them (e.g. a generic function for checking whether a given trajectory belongs to the behavior).
- Many functions have model-based and data-driven variants. A common Julia pattern for such case where one wants to choose between multiple algorithms to solve the same problem is to define special types (e.g. `ModelBased`, `DataDriven`) that can then be passed as an argument to functions to choose which variant to use.
  For example, we would use `ss2BT(sys, T, ModelBased)` and `ss2BT(sys, T, DataDriven)` instead of `ss2BT_modelbased` and `ss2BT_datadriven`. This would primarily make the high level interface cleaner.
- Most of the functions aren't particularly implemented with efficiency in mind (e.g. a lot of them allocate multiple arrays). This would limit the use of the package at scale, and generally makes the implementation less modular (see the implementations of the Hankel and Multiplication matrices for examples of what a better design might look like.)

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.

[3] Lasso.jl. https://juliastats.org/Lasso.jl/stable/.