Behavioral Toolbox

Ivan Markovsky

January 28, 2026

Contents

1 Introduction

The Behavioral Toolbox is a collection of Matlab functions for analysis and design of dynamical systems using the behavioral approach to systems theory and control [1, 2]. It implements newly emerged nonparameteric data-driven methods (see the overviews [3, 4]) and classical parametric representations of linear time-invariant (LTI) systems.

At the core of the behavioral approach is the notion of a dynamical system as a set of trajectories—the behavior—which is appealing but abstract. An often asked question is "What can be done in the behavioral setting that can't be done in the classical setting?" Indeed, for theoretical analysis as well as for numerical computations, a representation of the behavior is needed. However, it need not be state-space, transfer function, or convolution. The Behavioral Toolbox uses a basis for the behavior restricted to a finite-horizon, which is a nonparameteric representation of a discrete-time LTI system. Unlike classical nonparameteric representations, such as the convolution, a basis for the restricted behavior doesn't assume an input/output partitioning and is a direct application of the behavioral approach.

For an in-depth introduction to the philosophy of the behavioral approach, its differences from the classical approach, and its relation to data-driven methods for systems and control, refer to the overview papers [3, 4]. The notation used in this document is also the same as the one in the overview papers. A summary is given in Table 1. Optional extra references for techniques implemented or used in functions of the Behavioral Toolbox are [5–9].

$w \in (\mathbb{R}^q)^{\mathbb{N}}$, $w : \mathbb{N} \to \mathbb{R}^q$	q-variate real discrete-time signal with time axis $\mathbb{N}$
$w\|_T := \big(w(1), \dots, w(T)\big)$	restriction of w to the interval $[1, T]$
$w = w_{\text{ini}} \wedge w_{\text{f}}$	concatenation of trajectories w_{ini} and w_{f}
σ, $(\sigma w)(t) := w(t+1)$	unit shift operator
$\mathscr{B} \subset (\mathbb{R}^q)^{\mathbb{N}}$	discrete-time dynamical system with q variables
$\mathscr{B}\|_T := \{\, w\|_T \mid w \in \mathscr{B} \,\}$	restriction of $\mathscr{B}$ to the interval $[1, T]$
$\mathscr{L}^q$	set of linear time-invariant systems with q variables
$\boldsymbol{m}(\mathscr{B})$ / $\boldsymbol{\ell}(\mathscr{B})$ / $\boldsymbol{n}(\mathscr{B})$	number of inputs / lag / order of $\mathscr{B}$
$\boldsymbol{c}(\mathscr{B}) := \big(\boldsymbol{m}(\mathscr{B}), \boldsymbol{\ell}(\mathscr{B}), \boldsymbol{n}(\mathscr{B})\big)$	complexity of $\mathscr{B}$
$\mathscr{L}^q_c := \{\, \mathscr{B} \in \mathscr{L}^q \mid \boldsymbol{c}(\mathscr{B}) \leq c \,\}$	set of bounded complexity linear time-invariant systems
$\mathscr{H}_T(w)$	Hankel matrix with T block rows constructed from w

Table 1: Summary of notation.

Section 2 showcase the Behavioral Toolbox in examples. The examples admit solutions in the classical as well as in the behavioral settings. The latter are based on standard linear algebraic operations. They are conceptually simple and easy to implement. They are also applicable when the underlying system is unknown but data are available from the system. In this case, the methods in the toolbox achieve a direct map from the observed data to the desired solution. Classical solutions require a preliminary identification step. The current implementation of the methods in the Behavioral Toolbox, however, makes them less efficient computationally than alternative classical methods.

Section 3 is a tutorial to the functions in the toolbox. Optional material involving advanced topics or implementation details that can be skipped are marked with the symbol ☡ in the right margin. Section 4 is a literate program of the functions. Auxiliary functions used for the development of the toolbox are presented in Appendix A.

The webpage of the toolbox is: `https://imarkovs.github.io/bt`

2 Showcases

This section illustrates the behavioral approach on simple to state but realistic examples. First, we consider the question when two systems are equal. In the classical setting, equality of LTI systems represented in state-space can be verified by finding a state transformation that makes the corresponding systems' parameters equal. In the behavioral setting, the question is equivalent to verifying equality of subspaces—a classical linear algebra problem. The second example is interconnection of systems. We show that series connection in the input/output setting is a special case of intersection of behaviors. The third example is optimal signal-from-noise separation. The classical solution, which uses a state-space representation of the data-generating system, is the celebrated Kalman filter. The interpretation of the problem in the behavioral setting is projection on the behavior, *i.e.*, finding the nearest trajectory of a system to a given signal. The fourth example is the observer problem in the behavioral setting, *i.e.*, inferring one set of variables of a dynamical system from another set of variables, applied for the estimation of a missing input. The fifth example illustrates the direct data-driven approach, where instead of a system a trajectory of the system is given.

2.1 When are two systems equal?

The simple question "How to check if two systems are equal?" leads to the questions "How are the systems specified?" and "What does it mean that they are equal?". Intuitively we consider two systems equal when they have the same "external behaviors". The behavioral approach turns this intuition into a definition: the system *is* its external behavior.

Let's create a random discrete-time linear time-invariant system, defined by a state-space representation:

```
m = 2; p = 2; n = 3; sys1 = drss(n, p, m);
```

and another system that is obtained from the first one by a random change of the state-space basis:

```
sys2 = ss2ss(sys1, rand(n));
```

The external behaviors of the two systems are the same and in this sense the systems are *equal*. This fact, however, can not be inferred by comparing directly their state-space parameters and the equal to operation "==", is not supported:

```
try, sys1 == sys2, end % -> Operator '==' is not supported for operands of type 'ss'
```

The problem of finding when two systems are equal admits many solutions. The reason == is not supported for LTI systems in the Control Toolbox of Matlab is due to the perception that LTI systems (`ss` objects in particular) are not comparable. In fact, LTI systems are comparable but the parameters of their state-space representations are not. The shift in perception brought by the behavioral approach is that *parameters of a representation are not the system.* In the behavioral setting, the answer to the question when a system $\mathscr{B}^1$ is equal to a system $\mathscr{B}^2$ follows directly from the definition of a system as a set of trajectories: "when $\mathscr{B}^1 = \mathscr{B}^2$". The answer doesn't suggest methods for doing the job. For this, one has to choose particular representations of the systems; hence the many possible solution methods.

Note 1 (Definitions and problems in the behavioral setting). Another shift of perception brought by the behavioral approach is that definitions (*e.g.*, "$\mathscr{B}^1$ equals $\mathscr{B}^2$" means "$\mathscr{B}^1 = \mathscr{B}^2$") are stated in terms of the behavior, *i.e.*, systems' definitions do not involve systems' representations. Also, high-level problems related to systems (*e.g.*, check if $\mathscr{B}^1 = \mathscr{B}^2$) are stated without involving systems' representations. The high-level problem statement shows "what we are after", irrespective of how we may go about achieving it. Indeed, when the systems are initially given by some representations, say transfer functions, it may be beneficial to switch to another one, say state-space. The person solving the problem should not be a priori obliged to use a particular representation. Lower-level problem formulation, however, may ask specifically about solving a high-level problem using a particular representation or method.

The Behavioral Toolbox makes the abstract set theoretic framework of the behavioral approach actionable by restricting the infinite-horizon behavior $\mathscr{B}$ to a finite time-horizon $[1,T]$, resulting in a finite-dimensional subspace $\mathscr{B}|_T \subset (\mathbb{R}^q)^T$, and constructing a basis for it. Provided that T is long enough (often this means T is larger than the lag of the system), a problem about $\mathscr{B}$ can be reduced to an equivalent problem about $\mathscr{B}|_T$. In particular, one can check $\mathscr{B}^1 = \mathscr{B}^2$ by checking $\mathscr{B}^1|_T = \mathscr{B}^2|_T$—a basic linear algebra problem, for which there are existing methods. As usual in science and engineering, a new problem is solved by reducing it to an already solved problem. An extra benefit of reducing the problem to basic linear algebra operations is existence of high-quality software that is easily accessible.

The function `equal` of the toolbox implements the check for equality of systems:

```
equal(sys1, sys2) % -> true
```

First, it converts the `ss` objects `sys1` and `sys2` to bases for the finite-horizon behaviors $\mathscr{B}^1|_T$, $\mathscr{B}^2|_T$. This is done by a "data-driven approach" that simulates a random trajectory of the system, constructs a Hankel matrix of the trajectory, and computes a basis for the image of the Hankel matrix. Then, $\mathscr{B}^1|_T = \mathscr{B}^2|_T$ is checked numerically by finding the distance (defined as the principal angle) between $\mathscr{B}^1|_T$, $\mathscr{B}^2|_T$ and checking if the distance is smaller than a tolerance.

2.2 Interconnection of systems

In the classical setting, interconnection of systems is done by *input-to-output connection*: the output of one system is fed to an input of another system. In the behavioral setting, interconnection is viewed as *variables sharing*: some variables of one system are set equal to some variables of another system. Input-to-output connection is a special case of variables sharing. Indeed, the former restricts the latter by selecting input/output partitionings of the variables of the systems and equating input to output variables only. In [2], Jan C. Willems argues that interconnection of *physical systems* is *always* variables sharing and only *incidentally* (typically in man-made systems) input-to-output assignment.

Next, we show how series connection fits into the general setting of interconnection by variables sharing. Consider two single-input single-output LTI systems $\mathscr{B}^1$ and $\mathscr{B}^2$ with input/output partitionings

$$w^1 = \begin{bmatrix} w_1^1 \\ w_2^1 \end{bmatrix} = \begin{bmatrix} u^1 \\ y^1 \end{bmatrix} \quad \text{and} \quad w^2 = \begin{bmatrix} w_1^2 \\ w_2^2 \end{bmatrix} = \begin{bmatrix} u^2 \\ y^2 \end{bmatrix}.$$

```
n1 = 2; B1 = drss(n1);
n2 = 2; B2 = drss(n2);
```

The variables sharing corresponding to the series connection of $\mathscr{B}^1$ and $\mathscr{B}^2$ is $w_2^1 = w_1^2$, or written in a "kernel form"

$$\begin{bmatrix} 0 & 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} w_1^1 \\ w_2^1 \\ w_1^2 \\ w_2^2 \end{bmatrix} = 0. \tag{$w_2^1 = w_1^2$}$$

```
R3 = [0 1 -1 0];
```

Equation ($w_2^1 = w_1^2$) defines the static system $\mathscr{B}^3 := \{\, w \mid w_2^1 = w_1^2 \,\}$.

In order to construct the interconnection system $\mathscr{B}_{\text{int}}$, first, we define the joint behavior of $\mathscr{B}^1$ and $\mathscr{B}^2$ without the interconnection:

$$\text{append}(\mathscr{B}^1, \mathscr{B}^2) := \left\{ \begin{bmatrix} w^1 \\ w^2 \end{bmatrix} \;\middle|\; w^1 \in \mathscr{B}^1,\ w^2 \in \mathscr{B}^2 \right\}. \tag{append}$$

The interconnection of $\mathscr{B}^1$ and $\mathscr{B}^2$ with the interconnection law ($w_2^1 = w_1^2$) is obtained then as the intersection of the joint behavior append$(\mathscr{B}^1, \mathscr{B}^2)$ with $\mathscr{B}^3$,

$$\mathscr{B}_{\text{int}} = \text{append}(\mathscr{B}^1, \mathscr{B}^2) \cap \mathscr{B}^3.$$

It can be shown that the dynamics of the interconnected system $\mathscr{B}_{\text{int}}$ is fully specified by its finite horizon behavior with horizon $T = n_1 + n_2$.

```
T = n1 + n2;
```

Thus, for the computation of $\mathscr{B}_{\text{int}}$, we restrict to a finite horizon $[1, T]$. The functions `B2BT` and `R2BT` of the toolbox construct the finite horizon behavior $\mathscr{B}|_T$ of an LTI system $\mathscr{B}$, specified by a state-space and a kernel representation, respectively:

```
BT1 = B2BT(B1, T); BT2 = B2BT(B2, T); BT3 = R2BT(R3, 4, T);
```

The function `BTappend` constructs a basis for the finite-horizon behavior of append$(\mathscr{B}^1, \mathscr{B}^2)$ and `BTintersect` computes the intersection of of two subspaces:

```
BT12    = BTappend(BT1, 2, BT2, 2);
BT12int = BTintersect(BT12, BT3);
```

Next, we verify that the resulting interconnected system $\mathscr{B}_{\text{int}}$ corresponds to the series connection of $\mathscr{B}^1$ and $\mathscr{B}^2$:

```
B12 = B2 * B1;
```

For this purpose, we project the behavior of $\mathscr{B}_{\text{int}}$ on the $\begin{bmatrix} w_1^1 \\ w_2^2 \end{bmatrix}$ variables (thus eliminating the variables w_2^1 and w_1^2):

```
BT12_ = BTproject(BT12int, 4, [1 4]);
```

and, using the `equal` function, verify that the systems defined by `B12` and `BT12_` are equal:

```
equal(B12, BT12_, q) % -> 1
```

We've found the series connection of $\mathscr{B}_1$ and $\mathscr{B}_2$ using the bases for their finite-horizon behaviors.

2.3 Signal from noise separation

The problem considered next is: Given a system $\mathscr{B}$ and a noisy signal $w = \overline{w} + \widetilde{w}$, where $\overline{w} \in \mathscr{B}|_T$ is the to-be-estimated signal and $\widetilde{w}$ is a zero-mean white Gaussian noise signal, find the maximum-likelihood estimate $\widehat{w}$ of $\overline{w}$.

```
m = 1; p = 1; n = 3; q = m + p; B = drss(n, p, m);
T = 10; w0 = B2w(B, T); wn = randn(size(w0));
w = w0 + 0.1 * norm(w0) * wn / norm(wn);
```

The setup described above—the data w is true value $\overline{w}$ plus noise $\widetilde{w}$ is called errors-in-variables [10]. The maximum-likelihood estimate $\widehat{w}$ of $\overline{w}$ in the errors-in-variables setup is the solution of the optimization problem

$$\text{minimize} \quad \text{over } \widehat{w} \quad \|w - \widehat{w}\| \quad \text{subject to} \quad \widehat{w} \in \mathscr{B}|_T, \tag{KS}$$

i.e., the statistically optimal estimate $\widehat{w}$ has a simple geometric interpretation as the projection of the noisy signal w on the finite-horizon behavior $\mathscr{B}|_T$ of the data-generating system $\mathscr{B}$.

Problem (KS) does not involve a particular representation of the system $\mathscr{B}$. It states what we are after without suggesting a method for solving the problem (see Note 1). The solution can be obtained using for example a state-space representation, which leads to a Riccati equation [11]. An alternative solution method using the nonparameteric representation of the restricted behavior is:

```
BT = B2BT(B, T); wh = BT * BT' * vec(w'); wh = reshape(wh, q, T)';
```

Since the basis `BT` of $\mathscr{B}_T$ is orthonormal, `BT * BT′` is the orthogonal projector on $\mathscr{B}_T$, so `wh = BT * BT′ * vec(w′)` is the signal $\widehat{w}$ solving (KS).

The model-based solution using the Riccati equation is computationally more efficiently (in terms of number of floating point operations) however it is harder to derive and implement. The problem formulation as projection has geometric interpretation and the solution `wh = BT * BT′ * vec(w′)` involves two matrix multiplications only.

2.4 Missing input estimation

Consider an system $\mathscr{B}$, which variables w are separated as follows:

$$w = \begin{bmatrix} u_{\text{missing}} \\ u_{\text{given}} \\ y \end{bmatrix}.$$

As the notation suggests, u_{missing} is missing / unknown / to-be-estimated input, u_{given} is given / known / observed input, and y is given / known / observed output.

```
m_missing = 1; m_given = 1; m = m_missing + m_given; p = 2; n = 3; q = m + p;
B = drss(n, p, m); T = 10; w = B2w(B, T);
w_missing = w(:, 1:m_missing); w_given = w(:, m_missing+1:end);
```

The problem considered is to find u_{missing}, given u_{given}, y, and $\mathscr{B}$, *i.e.*, estimate a missing input u_{missing} of the system. The problem is a special case of the observer problem in the behavioral setting [12]. Using the nonparameteric representation of the restricted behavior, the solution of the missing input estimation problem leads to:

$$\widehat{u}_{\text{missing}} = B_T|_{I_{\text{missing}}} \left(B_T|_{I_{\text{given}}}\right)^+ \begin{bmatrix} u_{\text{given}} \\ y \end{bmatrix},$$

where $B_T|_{I_{\text{missing}}}$ and $B_T|_{I_{\text{given}}}$ are the submatrices of B_T corresponding to the missing and the given data, respectively, and $(\cdot)^+$ denotes the pseudo-inverse. In Matlab code, the solution is:

```
BT = B2BT(B, T); [BT_missing, BT_given] = BT2UYT(BT, m_missing, m_given + p);
wh_missing = BT_missing * BT_given' * vec(w_given');
wh_missing = reshape(wh_missing, m_missing, T)';
```

Under the verifiable from the data condition

$$\operatorname{rank} B_T|_{I_{\text{given}}} = \boldsymbol{m}(\mathscr{B})T + \boldsymbol{n}(\mathscr{B}),$$

```
rank(BT_given) == T * m + n % -> 1
```

the problem has a unique solution, so that the missing input is recovered exactly. Indeed,

```
e = norm(w_missing - w_missing) / norm(w_missing) % -> 0
```

so that using standard linear algebra operations (pseudo-inverse computation and multiplication) on the basis B_T of the finite-horizon behavior $\mathscr{B}|_T$ and the given data $\left[\begin{smallmatrix} u_{\text{given}} \\ y \end{smallmatrix}\right]$, we have found the missing input u_{missing}.

2.5 Direct data-driven forecasting

The behavioral toolbox is based on the nonparametric representation of the finite-horizon behavior $\mathscr{B}|_T$. Such a representation can be obtained from another representation or from data, *i.e.*, a trajectory $w_{\text{d}} \in (\mathbb{R}^q)^{T_{\text{d}}}$ of the system $\mathscr{B}$. Leveraging the linearity and time-invariance properties of the system and assuming sufficiently long ($T_{\text{d}} > T$) trajectory w_{d} of the system, the image of the Hankel matrix $\mathscr{H}_T(w_{\text{d}})$ constructed from the data is included in the finite-horizon behavior $\mathscr{B}|_T$. Moreover, assuming that $\operatorname{rank} \mathscr{H}_T(w_{\text{d}})$ is sufficiently high, more specifically

$$\operatorname{rank} \mathscr{H}_T(w_{\text{d}}) = \boldsymbol{m}(\mathscr{B})T + \boldsymbol{n}(\mathscr{B}),$$

we have that

$$\operatorname{image} \mathscr{H}_T(w_{\text{d}}) = \mathscr{B}|_T,$$

i.e., the raw data w_{d} in the form of the Hankel matrix $\mathscr{H}_T(w)$ serves as a nonparameteric representation of $\mathscr{B}|_T$.

The point that a representation of the finite-horizon behavior $\mathscr{B}|_T$ can be constructed directly from data using the Hankel matrix makes the methods in the toolbox applicable in situations where a model is not available but data from the system can be collected. In such situations, classical methods require a parametric system identification step prior to applying the methods. The methods in the toolbox, in contrast, implement the direct map from the given data to the desired solution. This direct map is theoretically interesting. As shown next it is also practically useful.

In practice the data is almost never exact. In case of noisy data, satisfying certain stochastic assumptions (true value plus zero mean white Gaussian noise), the maximum-likelihood solution, which is theoretically optimal, leads to a *Hankel structured low-rank approximation* problem [13]. The Hankel structured low-rank approximation problem is a nonconvex optimization problem. All currently existing methods are heuristics for solving this nonconvex problem. Based on empirical and theoretical evidence, we know that some heuristics are more effective than others. Currently there is no universally best method.

Next, we show the performance of the methods in the toolbox using a naive heuristic based on the pseudo-inverse. For comparison we apply the classical approach of model-identification (using state-of-the-art parametric system identification methods) followed by a model-based solution. The problem considered is prediction, *i.e.*, simulation of a "future" trajectory of unknown system, using past data. The data is the "Robot arm" benchmark from the data-base for system identification DAISY [14].

```
robot_arm_; m = 1; p = 1; n = 8;
wd = [u y]; [Td, q] = size(wd);
```

The data is split into an identification part followed by a validation part. The last $T_{\text{ini}} = 20$ samples of the identification part are also used for the estimation of the initial conditions for the validation part.

```
Tv = 50; Ti = Td - Tv;
wi = wd(1:Ti, :); wv = wd(Ti+1:end, :);
Tini = n; w_ini = wi(end - Tini + 1:end, :);
```

The prediction method based on the nonparameteric finite-horizon representation of the behavior is implemented in the function `u2y` of the Behavioral Toolbox. Note that once the past horizon T_{ini} is chosen, it has no tunable hyper-parameters.

```
tic
yh_dd = u2y(hank(wi, Tv + Tini), q, wv(:, 1:m), w_ini);
t_dd = toc % -> 0.0186
```

The corresponding model-based method is the function `forecast` from the System Identification Toolbox of Matlab, which takes as an input a model. The model is obtained by a subspace identification method, implemented in the `n4sid` function from the System Identification Toolbox. The model-based approach requires a hyper-parameter—the model order `n`. It is chosen as 8 by trail-and-error, aiming for optimal performance. In this hyper-parameter tuning the validation data is used which gives unfair advantage of the model-based method.

```
tic
Bh = n4sid(iddata(wi(:, m+1:end), wi(:, 1:m)), n);
f = forecast(Bh, iddata(w_ini(:, m+1:end), w_ini(:, 1:m)), Tv, wv(:, 1:m));
yh_mb = f.OutputData; t_mb = toc % -> 1.4176
```

Note that although the direct data-driven approach is computationally less efficient than the model-based approach, it is faster in the example.

The results of the two methods are compared in terms of the relative prediction error:

$$e := 100\% \frac{\|y_v - \widehat{y}_v\|}{\|y_v\|},$$

where y_v is the to-be-forecast output of the validation part of the data and $\widehat{y}_v$ is the forecast obtained by the method.

```
e = @(yh) 100 * norm(wv(:, m+1:end) - yh) / norm(wv(:, m+1:end));
[e(yh_dd), e(yh_mb)] % -> 3.5531    3.8398
```

In the example, the results of the two methods are similar. The model-based method, however, required tuning of a hyper-parameter (for which we've selected an optimal value using the validation data), while once T_{ini} is chosen, the direct data-driven method has no hyper-parameters.

3 Tutorial

In order to make the behavioral approach computationally feasible, we consider discrete-time systems and restrict the behavior $\mathscr{B}$ to a finite horizon $[1,T]$. The restricted behavior $\mathscr{B}|_T$ of a linear time-invariant system is a shift-invariant subspace of $(\mathbb{R}^q)^T$. The methods developed in the toolbox use a basis for $\mathscr{B}|_T$ as a representation of the system $\mathscr{B}$. The system analysis, signal processing, and control methods in Sections 3.1–3.6 take as an input a basis for $\mathscr{B}|_T$. The identification methods in Section 3.7 take as an input a trajectory $w_\text{d} \in \mathscr{B}|_{T_\text{d}}$ of the system. The code from this section and Appendix A is extracted in a script file `demo.m`. It serves as a demo as well as a test suite for the toolbox.

3.1 Nonparameteric representation of the restricted behavior

This section introduces the functions of the toolbox for creating and using a basis $\{b^1,\dots,b^r\}$

$$\mathscr{B}|_T = \text{image}\,\underbrace{\begin{bmatrix} b^1 & \cdots & b^r \end{bmatrix}}_{B_T}, \qquad \text{where } r := \dim \mathscr{B}|_T \tag{B_T}$$

for the restricted behavior $\mathscr{B}|_T$ of a linear time-invariant system $\mathscr{B} \in \mathscr{L}^q$. With some abuse of notation, in what follows we refer to the matrix $B_T := \begin{bmatrix} b^1 & \cdots & b^r \end{bmatrix} \in \mathbb{R}^{qT\times r}$ of the basis vectors as the basis. The representation (B_T) of $\mathscr{B}|_T$ is nonparameteric because it involves $r := \dim \mathscr{B}|_T$ parameters $g_1,\dots,g_r \in \mathbb{R}$ to specify $w \in \mathscr{B}|_T$ via $w = B_T g$. For $T \geq \boldsymbol{\ell}(\mathscr{B})+1$, $\mathscr{B}|_T$ uniquely defines $\mathscr{B}$ [8, Lemma 13]. Thus, (B_T) is a *nonparameteric representation* of $\mathscr{B}$.

In order to demonstrate and test the functions of the toolbox on numerical examples, we create a random linear time-invariant system $\mathscr{B} \in \mathscr{L}^q_{(m,\ell,n)}$, defined by an input/state/output representation—a Matlab's `ss` object `B`:

```
m = 2; p = 1; q = m + p; n = 3; B = drss(n, p, m);
```

The construction of the orthonormal basis B_T for the subspace $\mathscr{B}|_T \subset (\mathbb{R}^q)^T$ from an input/state/output representation of $\mathscr{B}$ is done by the function `B2BT`:

```
T = 10; BT = B2BT(B, T);
```

The function `B2BT` implements a data-driven approach. Instead of using the parameters of the state-space representation of $\mathscr{B}$, it computes a random trajectory w_d of $\mathscr{B}$ (using `lsim`), forms the Hankel matrix $\mathscr{H}_T(w_\mathrm{d})$ (using the toolbox's function `hank`), and computes an orthonormal basis for $\mathscr{B}|_T$ (using `svd`):

$$(A,B,C,D) \xmapsto{\texttt{lsim}} w_\mathrm{d} \xmapsto{\texttt{hank}} \mathscr{H}_T(w_\mathrm{d}) \xmapsto{\texttt{svd}} \mathscr{B}|_T$$

Alternative functions to `B2BT` that implement the model-based approach are `ss2BT` and `R2BT`. As the names suggest, `ss2BT` uses a state-space representation, while `R2BT` uses a kernel representation to compute a basis B_T for $\mathscr{B}|_T$.

In order to verify that the output `BT` of `B2BT` is correct, first, we verify that its dimension is correct. By [8, Corollary 5] we have that

$$\dim \mathscr{B}|_T = mT + n, \quad \text{for } T \geq \ell. \tag{dim $\mathscr{B}|_T$}$$

```
check(size(BT, 2) == m * T + n)
```

The function `check` prints "PASS" if its argument is true and "FAIL" otherwise.

Then, we check that a random trajectory $w \in \mathscr{B}|_T$ of the system is in the image of `BT`. The function `B2w` selects a random trajectory of `B`. Its output is a $T \times q$ matrix, where T is the number of samples and q is the number of variables. Checking if $w \in \mathscr{B}|_T$ is a common task and is implemented in a function `w_in_B`:

```
w = B2w(B, T); check(w_in_B(w, B))
```

The columns of `BT` are finite trajectories, *i.e.*, elements of $\mathscr{B}|_T$. The function `BT2W` extracts them into a cell array `W`:

```
W = BT2W(BT, q); check(all(w_in_B(W, B)))
```

The function `w_in_B` as well as many other functions of the toolbox all specification of multiple trajectories

$$\mathscr{W} = \{w^1, \dots, w^N\}, \quad \text{where } w^i \in (\mathbb{R}^q)^{T_i} \tag{W$_\mathrm{d}$}$$

by a cell array as well as a single trajectory w_d.

Note 2 (Two representations of a signal $w \in (\mathbb{R}^q)^T$). A q-variate, T-samples long signal $w := \big(w(1), \dots, w(T)\big)$ is represented in the toolbox by:

$$T \times q \text{ matrix} \begin{bmatrix} w^\top(1) \\ \vdots \\ w^\top(T) \end{bmatrix} \qquad \text{or} \qquad qT \times 1 \text{ vector} \begin{bmatrix} w(1) \\ \vdots \\ w(T) \end{bmatrix}.$$

The matrix format has the advantage that the number of variables q can be deduced from the size. However, it is incompatible with (B_T), which uses the vector format. The conversion from the matrix to the vector format is done by `vec(w')`, where `vec` is column-wise vectorization, and the conversion from the vector to the matrix format is done by `reshape(w, q, T)'`. Functions accepting B_T as an input require q to be passed as an extra input argument.

When the $T \times q$ matrix representation is used, multiple trajectories (W$_\mathrm{d}$) are stored in a cell array with N elements, the ith element of which is a $T_i \times q$ matrix representing w^i. When the $Tq \times 1$ vector representation is used and $T_1 = \cdots = T_N =: T$, multiple trajectories (W$_\mathrm{d}$) are stored in a $qT \times N$ matrix, the ith column of which represents w^i.

Note 3 (Computing the dimension of $\mathscr{B}|_T$). The computation of a basis B_T from a trajectory $w_\mathrm{d} \in \mathscr{B}|_{T_\mathrm{d}}$ requires finding the dimension of $\mathscr{B}|_T$. Under standard conditions on w_d (see Section 3.7) that are almost certainly satisfied for a sufficiently long random trajectory $w_\mathrm{d} \in \mathscr{B}|_T$,

$$\dim \mathscr{B}|_T = \operatorname{rank} \mathscr{H}_T(w_\mathrm{d}). \tag{dim-rank}$$

Thus, $\dim \mathscr{B}|_T$ can be found by rank computation. A robust way of computing $\operatorname{rank} \mathscr{H}_T(w_\mathrm{d})$ is thresholding the singular values of $\mathscr{H}_T(w_\mathrm{d})$. The functions of the toolbox that construct B_T from data therefore have an optional threshold parameter `tol` with default value:

```
if ~exist('tol', 'var') || isempty(tol), tol = 1e-8; end % <default-tol>
```

Alternatively, the user can specify the model's complexity, in which case the dimension is computed from (dim $\mathscr{B}|_T$).

3.2 Input/output partitionings

A partitioning of the variables $w(t) \in \mathbb{R}^q$ into inputs $u(t) \in \mathbb{R}^m$ and outputs $y(t) \in \mathbb{R}^p$ is defined by a permutation matrix $\Pi \in \mathbb{R}^{q\times q}$ as follows:

$$w \mapsto (u,y): \begin{bmatrix} u \\ y \end{bmatrix} := \Pi w \qquad \text{and} \qquad (u,y) \mapsto w: \; w = \Pi^\top \begin{bmatrix} u \\ y \end{bmatrix}. \tag{I/O}$$

Πw reorders the variables w, so that the first m variables are the inputs and the remaining $p := q - m$ variables are the outputs. The permutation matrix Π is specified in the functions of the toolbox by a vector `io`, such that `w(io)` $\mapsto$ `[u; y]`. The restricted behavior with permuted variables is created with the function `BT2BT`:

```
io = flip(1:q); BTp = BT2BT(BT, q, io);
```

Indeed, the trajectory $w' := \Pi w$ with permuted variables belongs to the image of `BTp`:

```
check(w_in_B(w(:, io), BTp))
```

The function `BT2UYT` extracts from the basis of the restricted behavior $\mathscr{B}|_T$ its input and output components:

```
[UT, YT] = BT2UYT(BT, m, p);
```

Note 4 (Default input/output partitioning $w = \left[\begin{smallmatrix} u \\ y \end{smallmatrix}\right]$). `BT2UYT` and other functions of the toolbox that require an input/output partitioning but do not accept a specification for it assume $\Pi = I_q$:

```
if ~exist('io', 'var') || isempty(io), io = 1:q; end % <default-io>
```

For a general partitioning (I/O), defined by `io`, call the functions with input `BT2BT(BT, q, io)` instead of `BT`.

The inverse transformation `UYT2BT` reconstructs $\mathscr{B}|_T$ from its input and output components:

```
check(norm(BT - UYT2BT(UT, YT, m, p)) == 0)
```

The number of inputs m is uniquely defined by the behavior. However, an input/output partitioning of the variables is in general not unique. The currently available methods in the literature for finding an input/output partitioning of a system $\mathscr{B}$ are based on parametric representations of the system (*e.g.*, the kernel representation). Next, we describe a data-driven method for finding an input/output partitioning directly from the finite-horizon behavior $\mathscr{B}|_T$. The key observation is that (I/O) is an input/output partitioning of $\mathscr{B}$ if and only if it is possible to simulate any trajectory of $\mathscr{B}$ by choosing the input u and the initial conditions, specified by a past trajectory w_{ini}. Therefore, for any $w_{\text{ini}} \in \mathscr{B}|_{T_{\text{ini}}}$ with $T_{\text{ini}} \geq \ell$ and $u_{\text{f}} \in (\mathbb{R}^m)^{T_{\text{f}}}$ with $T_{\text{f}} \geq \ell$, there should exist a unique $y_{\text{f}} \in (\mathbb{R}^p)^{T_{\text{f}}}$, such that

$$w_{\text{ini}} \wedge \Pi^\top \left[\begin{smallmatrix} u_{\text{f}} \\ y_{\text{f}} \end{smallmatrix}\right] \in \mathscr{B}|_{T_{\text{ini}}+T_{\text{f}}}.$$

Let $B_{T_{\text{ini}}+T_{\text{f}}}$ be the matrix of basis vectors for $\mathscr{B}|_{T_{\text{ini}}+T_{\text{f}}}$ and let $\left[\begin{smallmatrix} W_{\text{ini}} \\ U_{\text{f}} \end{smallmatrix}\right]$ be the the submatrix of $B_{T_{\text{ini}}+T_{\text{f}}}$, corresponding to the initial trajectory w_{ini} and the input u_{f}. Then, (I/O) is an input/output partitioning of $\mathscr{B}$ if and only if

$$\operatorname{rank} \begin{bmatrix} W_{\text{ini}} \\ U_{\text{f}} \end{bmatrix} = m(T_{\text{ini}} + T_{\text{f}}) + n.$$

The method described above is implemented in `is_io`, which checks if a given partitioning of the variables is a possible input/output partitioning, and `BT2IO`, which finds all possible input/output partitionings of the system. In order to test `is_io` and `BT2IO`, consider the following single-input single-output system

```
Bp = ss(tf([0 1], [1 1], 1));
```

that has an input/output partitioning $w = \left[\begin{smallmatrix} u \\ y \end{smallmatrix}\right]$ but not $w = \left[\begin{smallmatrix} y \\ u \end{smallmatrix}\right]$.

```
BpT = B2BT(Bp, T);
check(is_io(BpT, 2, [1 2]) == true)
check(is_io(BpT, 2, [2 1]) == false)
check(all(BT2IO(BpT, 2) == [1 2]))
```

3.3 Subbehaviors

The following subbehaviors of $\mathscr{B}$ are of special interest:

- $\mathscr{Y}_0$ — *zero-input subbehavior*, *i.e.*, the set of transient responses,
- $\mathscr{U}_0$ — *zero-output subbehavior*, *i.e.*, the set of inputs blocked by the system,
- $\mathscr{B}_0$ — *zero initial conditions subbehavior*, *i.e.*, the set of zero initial conditions trajectories,
- $\mathscr{B}_\mathrm{c}$ — *controllable subbehavior*, *i.e.*, the set of trajectories that are patchable with zero past trajectory, and
- $\mathscr{B}_\mathrm{p}$ — *periodic subbehavior*, *i.e.*, the set of periodic trajectories.

Orthonormal bases for their restrictions to $[1,T]$ are computed from $\mathscr{B}|_T$ by the following functions:

```
Y0 = BT2Y0(BT, q);
U0 = BT2U0(BT, q);
B0 = BT2B0(BT, q);
BC = BT2BC(BT, q);
%BP = BT2BP(BT, q); TODO
```

Let's verify that a free response of $\mathscr{B}$ is in the image of `Y0`:

```
y0 = initial(B, rand(n, 1), T-1); check(w_in_B(y0, Y0))
```

Next, we verify that $w = \left[\begin{smallmatrix} u \\ 0 \end{smallmatrix}\right]$ with u in the image of `U0` is a trajectory of the system:

```
u0 = reshape(U0 * rand(size(U0, 2), 1), m, T)';
w0 = [u0 zeros(T, p)]; check(w_in_B(w0, BT))
```

Next, we verify that the initial conditions of a random trajectory in the zero initial conditions subbehavior are zero

```
T0 = size(B0, 1) / q;
w0 = reshape(B0 * rand(size(B0, 2), 1), q, T0)';
xini = w2xini(w0, B); check(norm(xini) < tol)
```

Using the zero initial conditions subbehavior $\mathscr{B}_0 \subset \mathscr{B}|_T$, we can find the zero initial conditions input-to-output map $H_T : u|_T \mapsto y|_T$. Let B_0 be the matrix of the basis vectors for $\mathscr{B}_0$ and let U_0, Y_0 be the submatrices of B_0 corresponding to the inputs and the outputs. Then, $H_T = Y_0 U_0^{-1}$. The method is implemented in the function `BT2HT`:

```
[HT, T] = BT2HT(BT, q);
```

The $pT \times mT$ matrix H_T is the finite-horizon representation of the transfer function $H(z)$ of the system $\mathscr{B}$ corresponding to the input/output partitioning (I/O). In particular, it has lower-triangular block-Toeplitz structure:

```
check(norm(HT - convm(B, T)) < tol)
```

3.4 Analysis

In this section, we find properties of the system directly from its restricted behavior, rather than from parametric representations as done by classical analysis methods. The properties considered are: complexity, controllability, H_∞-norm, and distance between systems.

System's complexity

The complexity $\boldsymbol{c}(\mathscr{B})$ of a linear time-invariant system $\mathscr{B}$ is defined as the triple: number of inputs $\boldsymbol{m}(\mathscr{B})$, lag $\boldsymbol{\ell}(\mathscr{B})$, and order $\boldsymbol{n}(\mathscr{B})$. Although in the classical setting the order $\boldsymbol{n}(\mathscr{B})$ is defined via a minimal state-space representation, it is a property of the system $\mathscr{B}$ and can be computed directly from the restricted behavior $\mathscr{B}|_T$ (provided $T \geq \boldsymbol{\ell}(\mathscr{B})+1$). Also, the number of inputs $\boldsymbol{m}(\mathscr{B})$ can be computed from $\mathscr{B}|_T$ without reference to a particular input/output representation. The method for finding $\boldsymbol{m}(\mathscr{B})$ and $\boldsymbol{n}(\mathscr{B})$ from $\mathscr{B}|_T$ is based on (dim $\mathscr{B}|_T$). Evaluating dim $\mathscr{B}|_{t_i}$ (see Note 3) for $t_1 \neq t_2 \geq \boldsymbol{\ell}(\mathscr{B})$, *e.g.*, $t_1 = T$ and $t_2 = T-1$, we obtain the system of equations

$$\begin{bmatrix} T & 1 \\ T-1 & 1 \end{bmatrix} \begin{bmatrix} m \\ n \end{bmatrix} = \begin{bmatrix} \dim \mathscr{B}|_T \\ \dim \mathscr{B}|_{T-1} \end{bmatrix},$$

from which $m = \boldsymbol{m}(\mathscr{B})$ and $n = \boldsymbol{n}(\mathscr{B})$ can be found. The resulting method is implemented in the function `BT2c`:

```
[ch, mh, ellh, nh] = BT2c(BT, q); check(all([mh nh] == [m n]))
```

A model-based method for computing the lag of the system is implemented in the function `lag`:

```
ell = lag(B); check(ellh == ell)
```

Note 5. `BT2c` finds dim $\mathscr{B}|_{T-1}$ and dim $\mathscr{B}|_T$ by rank computation, *i.e.*, thresholding of the singular values using a user defined threshold `tol` (see Note 3).

The structure of a bounded complexity linear time-invariant system is fully characterized by the integer invariants $(\ell_1, \dots, \ell_p)$ [8, Section III]. The function `BT2ells` finds the integer invariants using a basis for the finite-horizon behavior.

```
ells = BT2ells(BT, q); check(sum(ells) == n && max(ells) == ell)
```

As discussed in the next section, the complexity connects nonparameteric and parametric representations and is a critical first step in finding a parametric representation of a bounded complexity system.

Distance between systems

The function `Bdist` computes distance between systems. The distance is defined as the principal angle between the finite-horizon behaviors of the systems with default horizon $T = 100$. Here is an example:

```
Bp = B; Bp.a = Bp.a + 0.01 * randn(n); d = Bdist(B, Bp);
```

The distance measure `Bdist` is used by the function `equal` for checking if two systems are equal:

```
Bp = ss2ss(B, rand(n)); check(equal(B, Bp))
```

Controllability

The controllable subbehavior $\mathscr{B}_c \subset \mathscr{B}|_T$ for $T \geq \boldsymbol{\ell}(\mathscr{B})$ is $mT+n$-dimensional if and only if $\mathscr{B}$ is controllable. This leads to a data-driven controllability test that is implemented in the function `isunctr`:

```
Bp = ss([0.5 1; 0 0.25], [1; 0], [1 1], 1, -1);
n_unctr = isunctr(B2BT(Bp, 10), 2); check(n_unctr == 1)
```

Moreover, $mT + n - \dim \mathscr{B}_c|_T$ is the number of uncontrollable modes of $\mathscr{B}$. Based on $\mathscr{B}_c$, a quantitative test for controllability—a distance to uncontrollability—is implemented in the function `distunctr`. Here is an example:

```
Bp = ss([0.5 1; 0 0.25], [1; 1e-5], [1 1], 1, -1);
d = distunctr(B2BT(Bp, 10), 2); % -> 1e-7
```

H_∞-norm

Using the zero initial conditions input-to-output map H_T of $\mathscr{B}$ for a given input/output partitioning allows us to compute the finite-horizon H_∞-norm of $\mathscr{B}$. The method is implemented in the function `BT2Hinf`:

```
[Hinf, ~, uinf] = BT2Hinf(BT, q); C = convm(B, T);
check(abs(Hinf - norm(C)) / norm(C) < tol)
```

3.5 Parametric representations

So far, the tutorial reviewed functions related to and based on the nonparametric representation (B_T) of the restricted behavior $\mathscr{B}|_T$. This section shows an application of the approach for computing parametric representations.

Computing a kernel representation from $\mathscr{B}|_T$ is essentially applying the `null` function on $B_{\ell+1}^\top$. The method is implemented in the functions `B2R` and `BT2R` (`B2R` constructs `BT` from `B` and calls `BT2R`):

```
R = B2R(B); R = BT2R(BT, q);
```

The inverse operation—finding the restricted behavior from a kernel or a state-space representation—is done by the functions `R2BT` and `ss2BT`, which implement the model-based approach, *i.e.*, they construct B_T from the model parameters R and (A,B,C,D), respectively.

```
BT_ = R2BT(B2R(B), q, T); check(equal(B, BT_))
BT_ = ss2BT(B, T);         check(equal(B, BT_))
```

The basis computed by `ss2BT` is not orthonormal. It consists of observability and convolution matrices, see ($(A,B,C,D) \mapsto B_T$) on page 25.

Contrary to `BT2R` (which is essentially Matlab's `null` function), computing a state-space representation from the restricted behavior is nontrivial. Indeed, it requires to do 1) state construction and 2) detect a possible input/output partitioning of the variables. The data-driven approach for these operations is implemented in the function `BT2ss`:

```
check(equal(B, BT2ss(BT, q)))
```

Similarly, the transformation from a kernel representation to a state-space representation is nontrivial. A data-driven method for this operation is implemented in the function `R2ss`:

```
check(equal(B, R2ss(R, q)))
```

For multi-output systems, the function `B2R` computes a *nonminimal* kernel representation. Computing a minimal kernel representation or converting a nonminimal kernel representation to a minimal one are also nontrivial operations as they require rank computation over the ring of polynomials. Data-driven methods for computing a special minimal representation, called shortest-lag [**W86a**], are implemented in the functions `BT2Rmin` and `R2Rmin`:

```
Rmin = BT2Rmin(BT, q); check(equal(BT, R2BT(Rmin, q, T), q))
Rmin = R2Rmin(R, q);   check(equal(BT, R2BT(Rmin, q, T), q))
```

Note 6 (Parametric vs non-parametric representations). The dichotomy of parametric vs nonparametric representations is misleading—there is a range of nonminimal parametric representations that cover the gap between minimal parametric and nonparametric representations. The problem of detecting when a parametric representation is minimal is equivalent to the one of finding the model's complexity, see Note 3. It is also essential in the case of identification from noisy data where the key issue is the one of achieving an accuracy–complexity trade-off.

Note 7 (The `ctol` parameter). All functions of the toolbox computing model's complexity accept as an optional input argument a tolerance for thresholding the singular values. Alternatively, the user may specify the model's complexity `[m, ell, n]` by `ctol`. If `ctol` is a scalar, it is used as a threshold `tol` for rank estimation as explained in Note 3. Otherwise, `ctol` should be the vector of integers specifying the model's complexity.

3.6 Signal processing and open-loop control

This section collects operations on signals by a dynamical system. The operations implemented are: projection of a signal on a system, computing initial conditions for a signal, simulation, and inference of one variable from another. They are special cases of a signal processing problem, called *interpolation and approximation of trajectories* [9].

Projection

The basic operation (KS) of projection of a signal $w \in (\mathbb{R}^q)^T$ on the behavior $\mathscr{B}|_T$ of a system $\mathscr{B} \in \mathscr{L}^q$ is equivalent to computing the *distance from w to $\mathscr{B}$*. The same operation is equivalent to (errors-in-variables) Kalman smoothing [11]: the projection $\widehat{w}$ is the *smoothed version* of w. Problem (KS) is solved by the function `dist`:

```
[d, wh] = dist(w, B);
```

The distance `d` from w to $\mathscr{B}|_T$ is zero if and only if $w \in \mathscr{B}|_T$, so that `dist` is a robust way of checking if w is an exact trajectory of $\mathscr{B}$. (It is used in the implementation of `w_in_B`.)

Initial conditions estimation

For $w_d \in \mathscr{B}|_T$, there is a corresponding initial conditions. If $\mathscr{B}$ is defined by an input/state/output representation, the initial conditions can be specified by the initial state $x_{\text{ini}} = x(1)$. The initial state x_{ini} corresponding to a trajectory w can be found with the function `w2xini`:

```
[xini, e] = w2xini(w, B); check(norm(w - B2w(B, T, xini, w(:, 1:m))) < tol)
```

The output argument `e` is zero when $w \in \mathscr{B}|_T$ and nonzero otherwise.

The initial conditions can be specified in a representation free way by an initial trajectory w_{ini} of length $T_{\text{ini}} \geq \boldsymbol{\ell}(\mathscr{B})$. An initial trajectory w_{ini} of corresponding to a given trajectory w can be computed by the function `w2wini`:

```
BT = B2BT(B, ell + size(w,1));
[wini, e] = w2wini(w, BT, ell); check(w_in_B([wini; w], B))
```

Simulation

The classical simulation problem: given initial conditions and input, find the corresponding output is solved by `u2y`:

```
yf = u2y(BT, q, w(:, 1:m), wini); check(w(:, m+1:end) - yf)
```

As another example, next, we use `u2y` in order to find the first `Tf` samples of the impulse response of the system:

```
Tf = 3; u1 = [[1; zeros(q * Tf - 1, 1)] zeros(q * Tf, m - 1)];
h1 = u2y(BT, q, u1); h = lsim(B, u1); check(h1 - h < tol)
```

Observer

Consider a linear time-invariant system $\mathscr{B} \in \mathscr{L}^q$. The variables w of $\mathscr{B}$ are partitioned into given/observed variables w_{given} with dimension q_{given} and missing/to-be-estimated variables w_{missing} with dimension $q_{\text{missing}} = q - q_{\text{given}}$. Without loss of generality, we assume that $w = \left[\begin{smallmatrix} w_{\text{given}} \\ w_m \end{smallmatrix}\right]$. The problem considered is: Given a system $\mathscr{B} \in \mathscr{L}^q$ and observed part w_{given} of a trajectory $w \in \mathscr{B}|_T$, find the missing part w_{missing} of w.

The solution is based on the finite-horizon nonparameteric representation (B_T) of $\mathscr{B}$. Let B_{given} be the submatrix of B_T corresponding to w_{given} and B_{missing} be the submatrix corresponding to w_{missing}. Then, there is a g, such that

$$w_{\text{given}} = B_{\text{given}} g \quad \text{and} \quad w_{\text{missing}} = B_{\text{missing}} g.$$

Solving for g the first equation and substituting into the second one, we have the following estimate of w_{missing}:

$$\widehat{w}_{\text{missing}} := B_{\text{missing}} B_{\text{given}}^{+} w_{\text{given}}.$$

It can be shown that, when $\operatorname{rank} B_{\text{given}} = \boldsymbol{m}(\mathscr{B})T + \boldsymbol{n}(\mathscr{B})$, $\widehat{w}_{\text{missing}}$ is exact, *i.e.*, of $\widehat{w}_{\text{missing}} = w_{\text{missing}}$.

Special cases of the observer problem are estimation of the input given the output (which is also a system inversion problem) and estimation of a second input, *e.g.*, disturbance, given the input and the output.

The solution of the observer problem is implemented in the function `wgiven2wmissing`. Here is an example verifying the method for disturbance estimation:

```
qg = 2; qm = 2; q = qm + qg; m = 1; p = q - m; n = 5; B = drss(n, p, m);
T = 20; w = B2w(B, T); wg = w(:, 1:qg); wm = w(:, qg+1:q);
wmh = wgiven2wmissing(wg, B); check(norm(wm - wmh) / norm(wm) < tol)
```

Open-loop control

The control problem considered is open-loop linear quadratic tracking: Given a system $\mathscr{B} \in \mathscr{L}^q$, to-be-tracked trajectory $w_{\text{r}} \in (\mathbb{R}^q)^{T_{\text{r}}}$, and tracking criterion $\|e\|_v := \|v \otimes e\|$, where $v \in (\mathbb{R}^q_+)^{T_{\text{r}}}$ and $\otimes$ is the element product,

$$\text{minimize} \quad \text{over } \widehat{w} \in (\mathbb{R}^q)^{T_{\text{r}}} \quad \|w_{\text{r}} - \widehat{w}\|_v \quad \text{subject to} \quad \widehat{w} \in \mathscr{B}|_{T_{\text{r}}}.$$

A variation of the problem is to specify initial and final conditions for the control trajectory $\widehat{w}$ via "past" and "future" trajectories $w_{\text{p}} \in (\mathbb{R}^q)^{T_{\text{p}}}$ and $w_{\text{f}} \in (\mathbb{R}^q)^{T_{\text{f}}}$:

$$\text{minimize} \quad \text{over } \widehat{w} \in (\mathbb{R}^q)^{T_{\text{r}}} \quad \|w_{\text{r}} - \widehat{w}\|_v \quad \text{subject to} \quad w_{\text{p}} \wedge \widehat{w} \wedge w_{\text{f}} \in \mathscr{B}|_{T_{\text{p}}+T_{\text{r}}+T_{\text{f}}}. \tag{LQCTR}$$

This allows us to solve optimal state transfer problems, *i.e.*, find a trajectory of the system that achieves optimal transition from the given initial state to the given final state. The solution exists for any initial and final states if and only if the system $\mathscr{B}$ is controllable and the state transfer is at least $\boldsymbol{\ell}(\mathscr{B})$-samples long.

The function in the toolbox solving the open-loop linear quadratic tracking control problem (LQCTR) is

```
wh = lqctr(B, wr, v, wp, wf);
```

The input parameters `v`, `wp`, and `wf` are optional. The default value of `v` is `ones(Tr, q)`, *i.e.*, uniform weights for all variables and all moments of time. If `wp` and/or `wf` is not specified, then the corresponding constraint in (LQCTR) is dropped, *i.e.*, the corresponding initial or final condition is free.

Here is an example of linear quadratic tracking with free initial and final condition:

```
m = 1; p = 1; n = 3; q = m + p; B = drss(n, p, m);
Tr = 10; wr = [zeros(Tr, m) ones(Tr, p)]; ell = lag(B);
v = [1e-2 * ones(Tr, m), ones(Tr, p)]; wh = lqctr(B, wr, v);
```

In order to solve the optimal state transfer problem with specified initial and final states, first we check controllability of the system. The solution is verified by showing that $w_{\text{p}} \wedge \widehat{w} \wedge w_{\text{f}} \in \mathscr{B}|_{T_{\text{p}}+T_{\text{r}}+T_{\text{f}}}$.

```
if ~isunctr(B2BT(B, 10), q)
  Tr = ell; wr = zeros(Tr, q);
  wp = B2w(B, ell); wf = zeros(ell, q);
  wh = lqctr(B, wr, [], wp, wf);
  check(w_in_B([wp; wh; wf], B))
end
```

Control as interconnection

In the following example the plant $\mathscr{B}$ is a 4th order single-input single-output system, defined by the transfer function

$$H(z) = \frac{0.2826z + 0.5067z^2}{1 - 1.4183z + 1.5894z^2 - 1.3161z^3 + 0.8864z^4}.$$

```
Q = [0 0 0 0.28261 0.50666]; P = [1 -1.41833 1.58939 -1.31608 0.88642];
B = ss(tf(Q, P, -1));
```

The controller $\mathscr{B}_c$ is obtained with the function `h2syn` from the Robust Control Toolbox of Matlab

```
Bc = h2syn(B, 1, 1);
```

It leads to the closed-loop system $\mathscr{B}_{cl}$

```
B_ = ss(B.a, B.b, [B.c; B.c], [B.d; B.d], 1); Bcl = lft(B_, Bc);
```

The closed-loop system $\mathscr{B}_{cl}$ can be obtained alternatively by projecting on the y-variable the interconnection of the plant $\mathscr{B}$ and the control $\mathscr{B}_c$ with flipped inputs and outputs:

$$\mathscr{B}_{cl} = \Pi_y\big(\mathscr{B} \cap \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right] \mathscr{B}_c\big).$$

```
T = 10; BT = B2BT(B, T); BcT = B2BT(Bc, T);
Bcl_wT = BTintersect(BT, BT2BT(BcT, q, [2 1]));
[Bcl_uT, Bcl_yT] = BT2UYT(Bcl_wT, 1, 1);
equal(Bcl, Bcl_yT, q) % -> 1
```

3.7 Identification

The functions presented in the previous sections of the tutorial use the nonparametric representation (B_T) of a linear time-invariant system $\mathscr{B} \in \mathscr{L}_c^q$. In this section, the system $\mathscr{B}$ is implicitly specified by a trajectory $w_d \in \mathscr{B}|_{T_d}$

```
Td = 100; wd = B2w(B, Td);
```

A necessary and sufficient condition for w_d to define $\mathscr{B} \in \mathscr{L}_c^q$ is the *identifiability condition* [8]:

$$\operatorname{rank} \mathscr{H}_{\boldsymbol{\ell}(\mathscr{B})+1}(w_d) = \boldsymbol{m}(\mathscr{B})\big(\boldsymbol{\ell}(\mathscr{B})+1\big) + \boldsymbol{n}(\mathscr{B}). \tag{GPE-ID}$$

(GPE-ID) is a *generalized persistency of excitation* condition on the data w_d and requires prior knowledge of the complexity $\boldsymbol{c}(\mathscr{B})$ of the data-generating system $\mathscr{B}$.

When (GPE-ID) is satisfied, $\mathscr{B}$ can be recovered from the data w_d. This is done by finding the *most powerful unfalsified model* $\mathscr{B}_{\text{MPUM}}(w_d)$ for w_d. $\mathscr{B}_{\text{MPUM}}(w_d)$ is the least complicated exact model for w_d in the class of linear time-invariant systems $\mathscr{L}^q$. Note that $\mathscr{B}_{\text{MPUM}}(w_d)$ is well defined (*i.e.*, it exists and is unique) for any $w_d \in (\mathbb{R}^q)^{T_d}$, independently of whether or not (GPE-ID) holds. Also, finding $\mathscr{B}_{\text{MPUM}}(w_d)$ from w_d does not require priori knowledge of $\boldsymbol{c}(\mathscr{B})$: $\mathscr{B}_{\text{MPUM}}(w_d)$ is a computational construct that has theoretical guarantees to recover the data-generating system under the condition of (GPE-ID).

For $T \geq \boldsymbol{\ell}(\mathscr{B})$, under the stronger generalized persistency of excitation condition

$$\operatorname{rank} \mathscr{H}_T(w_d) = \boldsymbol{m}(\mathscr{B})T + \boldsymbol{n}(\mathscr{B}), \tag{GPE}$$

(B_T) holds. A necessary condition for (GPE) is that the length T_d of w_d must satisfy the lower bound

$$T_d \geq T_{\min} := (m+1)T + n - 1, \tag{$T_{\min}$}$$

```
Tmin = @(T) (m + 1) * T + n - 1; %% <define-Tmin>
```

Thus, ($T_{\min}$) quantifies the statement "w_d is long enough for recovering the data-generating system".

An analog of `BT2c` for complexity computation from data w_d is `c_mpum`, which computes $\boldsymbol{c}\big(\mathscr{B}_{\text{MPUM}}(w_d)\big)$ without explicitly identifying $\mathscr{B}_{\text{MPUM}}(w_d)$:

```
ch = c_mpum(wd); check(all(ch == [m, lag(B), n]))
```

The function `w2BT` computes a basis B_T for $\mathscr{B}_{\text{MPUM}}(w_d)|_T$ from data w_d

```
BhT = w2BT(wd, T);
```

Once $\widehat{B}_T$ is obtained from w_{d}, the non-parametric analysis functions from the previous sections can be used.

The second parameter of `w2BT` allows specification of an upper bound `[m, ell, n]` on the model's complexity, in which case unstructured low-rank approximation is used to enforce the bound. Alternatively, the second parameter of `w2BT` can be a tolerance `tol` for the estimation of the complexity by singular values thresholding (see Notes 6 and 3). Unlike the function `slra`, which imposes both rank and structure constraints, `w2BT` doesn't impose shift-invariance structure on `BT`.

Note 8. `c_mpum` and `w2BT` detect, respectively, $\boldsymbol{c}\big(\mathscr{B}_{\mathrm{MPUM}}(w_{\mathrm{d}})\big)$ and $\dim \mathscr{B}_{\mathrm{MPUM}}(w_{\mathrm{d}})$ from the data w_{d} by rank computation with a user defined tolerance (see Note 3 and Appendix **??**). Alternatively, the user may specify directly a complexity bound. When the complexity is wrongly estimated or misspecified, there may be no $\widehat{\mathscr{B}} \in \mathscr{L}^q_{(m,\ell,n)}$ that corresponds to image B_T, see Appendix **??**.

The following functions compute parametric representations of $\mathscr{B}_{\mathrm{MPUM}}(w_{\mathrm{d}})$ and thus by construction impose both the bounded complexity and LTI structure. A kernel representation of $\mathscr{B}_{\mathrm{MPUM}}(w_{\mathrm{d}})$ is obtained by the function `w2R`:

```
R = w2R(wd); check(w_in_B(wd, R2BT(R, q, Td)))
```

and a state-space representation by the function `w2ss`:

```
Bh = w2ss(wd); check(w_in_B(wd, Bh))
```

A lag recursive method for computing a *minimal* kernel representation of $\mathscr{B}_{\mathrm{MPUM}}(w_{\mathrm{d}})$ ([8, Algorithm 1]) is implemented in the function `mpum`:

```
R_ = mpum(wd); check(w_in_B(wd, R2BT(R_, q, Td)))
```

Note 9. `c_mpum`, `w2R`, `w2ss`, and `mpum` use the same methods as `BT2c`, `BT2R`, `BT2ss`, and `BT2Rmin`, respectively. The former operate on B_T while the latter on $\mathscr{H}_T(w_{\mathrm{d}})$. For exact data satisfying the (GPE) condition, image $B_T =$ image $\mathscr{H}_T(w_{\mathrm{d}})$ and the functions yield the same result. For inexact data, an approximation of image $\mathscr{H}_{T_{\mathrm{d}}}(w_{\mathrm{d}})$ by a low-dimensional linear time-invariant structured subspace is needed, see Note 6. This is a nontrivial operation that makes system identification problems harder than corresponding analysis problems.

4 Implementation

- Signals are represented in two different formats—matrix and vector format, see Note 2.
- For input/state/output representations we use the Control System Toolbox's `ss` object.
- In the naming of the functions and in the code, LTI systems are denoted by `B` when represented by an `ss` object and by `BT` when represented by (B_T). In the `BT` representation, the number of variables q has to be passed along with `BT`. Thus, functions accepting (B_T) as an input also require q as an extra input argument.
- The polynomial matrix
$$R(z) = R_0 + R_1 z + \cdots + R_\ell z^\ell \in \mathbb{R}^{p\times q}[z]$$
is represented by the matrix of its coefficients stacked in a row
$$R = \begin{bmatrix} R_0 & R_1 & \cdots & R_\ell \end{bmatrix} \in \mathbb{R}^{p\times q(\ell+1)}.$$
Contrary to Matlab's convention of ordering the coefficients in descending degrees, we order them in ascending degrees (which is consistent with the multiplication of R with the Hankel matrix $R\mathscr{H}_{\ell+1}(w_{\mathrm{d}}) = 0$). As in the vector format for the representation of a signal, where the number of variables q has to be passed along with w, functions accepting R as an input also require q as an extra input argument.
- We use literate programming [15, 16]. This document is written in Emacs Org mode [17] and the code is automatically extracted from the source file.
- Sections 4.1–4.6 follow Sections 3.1–3.6 of the tutorial.

Handling `ctol`

The variable `ctol` is either complexity `c = [m ell n]` or tolerance `tol`. It is used for computing (from `c`) or estimating (from `tol`) the rank of the Hankel matrix $\mathscr{H}_T(w_d)$, which indicates the dimension of the restricted behavior $\mathscr{B}|_T$, see (**??**).

```
function r = c2r(c, T, opt)
if ~exist('opt', 'var'), r = c(1) * T + c(3);
else r = c(1) * T; end % needed for BT2B0

function [m, ell, n] = unpack_c(c), m = c(1); ell = c(2); n = c(3);
```

When `ctol` is specified as a tolerance, the conversion from `ctol` to `c`, requires complexity estimation using `R`, `BT`, or `wd`, whichever is given. This is done by the functions `BT2c` and `c_mpum`:

```
%% <ctol2c>
if isscalar(ctol) % estimate c from BT or wd
  % if exist('R', 'var') && ~exist('BT', 'var'), BT = null(R); end
  if exist('BT', 'var'), [c, m, ell, n] = BT2c(BT, q, ctol);
  elseif exist('wd', 'var'), [c, m, ell, n] = c_mpum(wd, ctol); end
else % ctol should specify complexity
  c = ctol; [m, ell, n] = unpack_c(c);
end
```

The computation of the model's complexity in the «`ctol2c`» chunk is the most fragile operation in the toolbox. On the other hand, «`ctol2c`» is used in almost all functions. In some cases it causes infinite recursion. This issue is still investigated and «`ctol2c`» will be subject to change.

If `ctol` is not specified, its default value is the default tolerance `tol`:

```
<<default-tol>>
if ~exist('ctol', 'var') || isempty(ctol), ctol = tol; end % <default-ctol>
```

4.1 Nonparametric representation of the restricted behavior

`B2w` — select a random trajectory w from a system $\mathscr{B}$

The system $\mathscr{B}$ is specified by an `ss` object, *i.e.*, an input/state/output representation. The Control System Toolbox's function `lsim` is used to simulate a trajectory $w \in \mathscr{B}|_T$. By default, random input and random initial conditions are used. If the simulated trajectory w is used in data-driven methods to represent the system $\mathscr{B}$, the initial conditions should be random. With zero initial conditions, the uncontrollable subbehavior of $\mathscr{B}$ is not included in $\mathscr{B}_{\text{MPUM}}(w_d)$.

```
function w = B2w(B, T, xini, u)
[p, m] = size(B); n = order(B);
if ~exist('xini', 'var') || isempty(xini), xini = rand(n, 1); end
if isscalar(xini) && xini == 0, xini = zeros(n, 1); end
if ~exist('u', 'var') || isempty(u), u = rand(T, m); end
if isscalar(u) && u == 0, u = zeros(T, m); end
y = lsim(B, u, [], xini); w = [u y];
```

`R2w` — select a random bounded trajectory w from a system specified by a kernel representation

```
function [w, BT] = R2w(R, q, T, ctol)
<<default-ctol>>
BT = R2BT(R, q, T, ctol); w = BT * rand(size(BT, 2), 1); w = reshape(w, q, T)';
```

`B2BT` — construct an orthonormal basis for the restricted behavior $\mathscr{B}|_T$

`B2BT` implements the indirect "data-driven" approach:

$$(A,B,C,D) \xmapsto{\texttt{lsim}} w_\mathrm{d} \xmapsto{\texttt{hank}} \mathscr{H}_T(w_\mathrm{d}) \xmapsto{\texttt{orth,lra}} B_T$$

The simulated random trajectory w_d has the minimal length (T_min) in order for the Hankel matrix $\mathscr{H}_T(w_\mathrm{d})$ to reach its maximal rank $mT+n$. Since the complexity of $\mathscr{B}$ is a priori known, it is enforced using `lra` with rank `m*T + n`.

```
function BT = B2BT(B, T);
[p, m] = size(B); n = order(B);
<<define-Tmin>>
BT = lra(hank(B2w(B, Tmin(T)), T), m * T + n);
```

`BT2W` — convert the basis `BT` into a set of trajectories (cell array)

```
function W = BT2W(BT, q);
[qT, N] = size(BT); T = qT / q;
for i = 1:N, W{i} = reshape(BT(:, i), q, T)'; end
```

`BT2BT` — change horizon or permute variables

```
function BT = BT2BT(BT, q, Tio, ctol)
T = size(BT, 1) / q;
if isscalar(Tio), Tnew = Tio; % horizon restriction/extension
  if T == Tnew, return, end    % nothing to do
  <<default-ctol>>
  if T < Tnew, % extension
    R = BT2R(BT, q, ctol); BT = R2BT(R, q, T, ctol);
  else % restriction
    BTnew = [];
    for i = 0:(T - Tnew)
      BTnew = [BTnew BT(i * q + 1:(i + Tnew) * q, :)];
    end
    if length(ctol) == 3
      BT = lra(BTnew, c2r(ctol, Tnew));
    else, BT = orth(BTnew, ctol); end
  end
else, io = Tio;  % variables permutation
  BT = BT(q * kron(0:T-1, ones(1, q)) + kron(ones(1, T), io), :);
end
```

It was empirically observed that `BT(1:q * Tnew, :)` may fail. Most probably the reason is the finite precision computation. A fix is to complement `BT(1:q * Tnew, :)` with all possible shifts of a window with the new horizon over the old horizon.

After changing the horizon, in general, the dimension of the subspace changes. An orthonormal bases is recomputed by `orth` if `ctol` is tolerance and `lra` if `ctol` is complexity.

Extending the horizon requires obtaining a parametric model from `BT` and using it for construction of the new finite-horizon behavior.

`BTappend` — append one restricted behavior to another

```
function BT = BTappend(BT1, q1, BT2, q2, ctol)
<<default-ctol>>
```

```
[q1T1, nc1] = size(BT1); T1 = q1T1 / q1;
[q2T2, nc2] = size(BT2); T2 = q2T2 / q2;

T = min(T1, T2); q = q1 + q2;
BT1 = BT2BT(BT1, q1, T, ctol);
BT2 = BT2BT(BT2, q2, T, ctol);

BT = zeros(q * T, nc1 + nc2);
I = kron(ones(1, T), 1:q1)   + kron(0:T-1, q * ones(1, q1));
BT(I, 1:nc1)       = BT1;
I = kron(ones(1, T), q1+1:q) + kron(0:T-1, q * ones(1, q2));
BT(I, nc1+1:end) = BT2;
<<ctol2c>>
BT = lra(BT, c2r(c, T));
```

4.2 Input/output partitioning

`BT2UYT` — extract the restricted behaviors of the input and the output variables

```
function [UT, YT] = BT2UYT(BT, m, p)
q = m + p; [qT, N] = size(BT); T = qT / q;
UT = zeros(m * T, N); for i = 1:m, UT(i:m:end, :) = BT(i:q:end, :); end
YT = zeros(p * T, N); for i = 1:p, YT(i:p:end, :) = BT(m+i:q:end, :); end
```

`BTproject` — projection of the behavior on a subset of variables

```
function BvarT = BTproject(BT, q, var)
BT = BT2BT(BT, q, [var, setdiff(1:q, var)]);
BvarT = BT2UYT(BT, length(var), q - length(var));
```

`UYT2BT` — reconstruct $\mathscr{B}|_T$ from the input and the output restricted behaviors

```
function BT = UYT2BT(UT, YT, m, p)
q = m + p; [mT, N] = size(UT); T = mT / m; BT = zeros(q * T, N);
for i = 1:m, BT(i:q:end, :) = UT(i:m:end, :); end
for i = 1:p, BT(m+i:q:end, :) = YT(i:p:end, :); end
```

`is_io` — check if a partitioning of the variables is a possible input/output partitioning

```
function ans = is_io(BT, q, io, ctol);
<<default-ctol>>
[c, m, ell, n] = BT2c(BT, q, ctol);
p = q - m; T = size(BT, 1) / q;
BTp = BT(1:q * ell, :); BTf = BT(q * ell + 1:end, :); % selection from BT!
UTf = BT2UYT(BT2BT(BTf, q, io), m, p);
ans = rank([BTp; UTf], tol) == m * T + n;
```

Note: the horizon should be $T \geq 2\ell + 1$.

`BT2IO` — find all input/output partitionings of the variables

```
function [IO_possible, IO_impossible] = BT2IO(BT, q, ctol);
<<default-ctol>>
```

```
IO = flipud(perms(1:q)); possible = []; impossible = [];
for i = 1:size(IO, 1)
  if is_io(BT, q, IO(i, :), ctol),
    possible = [possible; i];
  else
    impossible = [impossible; i];
  end
end
IO_possible   = IO(possible, :);
IO_impossible = IO(impossible, :);
```

Note: the horizon should be $T \geq 2\ell + 1$.

4.3 Subbehaviors

`BT2Y0` and `ss2Y0` — zero input subbehavior

```
function Y0 = BT2Y0(BT, q, ctol);
<<default-ctol>>
[c, m] = BT2c(BT, q, ctol); p = q - m;
[UT, YT] = BT2UYT(BT, m, p);
Y0 = YT * null(UT, tol);
```

- Note: the horizon should be $T \geq \ell + 1$.

```
function Y0 = ss2Y0(B, T); Y0 = obsvm(B, T);
```

`BT2U0` and `ss2U0` — zero output subbehavior

```
function U0 = BT2U0(BT, q, ctol);
<<default-ctol>>
[c, m] = BT2c(BT, q, ctol); p = q - m;
[UT, YT] = BT2UYT(BT, m, p);
U0 = UT * null(YT, tol);
```

- Note: the horizon should be $T \geq \ell + 1$.

```
function U0 = ss2U0(B, T);
% TODO
```

`BT2B0` and `B2B0` — zero initial conditions subbehavior

```
function [B0, T] = BT2B0(BT, q, ctol);
<<default-ctol>>
<<ctol2c>>
Bini = BT(1:q * ell, :); T = size(BT, 1) / q - ell; % selection from BT!
B0 = lra(BT(q * ell + 1:end, :) * null(Bini, tol), c2r(c, T, 0));
```

Note: the horizon should be $T \geq 2\ell$.

```
function B0 = B2B0(B, T, ctol)
<<default-tol>>
[p, m] = size(B); q = m + p; ell = lag(B);
B0 = BT2B0(B2BT(B, T+ell), q, ctol);
```

BT2HT and B2HT — zero initial conditions finite-horizon input-to-output map

```
function [HT, T] = BT2HT(BT, q, ctol)
<<default-ctol>>
[c, m] = BT2c(BT, q, ctol); p = q - m;
[U0, Y0] = BT2UYT(BT2B0(BT, q, ctol), m, p);
HT = Y0 * pinv(U0, tol); T = size(U0, 1) / m;

function HT = B2HT(B, T), HT = convm(B, T);
```

BT2BC and B2BC — controllable subbehavior

```
function [BC, c] = BT2BC(BT, q, ctol);
<<default-ctol>>
[c, m, ell, n] = BT2c(BT, q, ctol); ell_ctr = n; % use an upper bound
Bini = BT(1:q * ell, :); % selection from BT!
BC = orth(BT((q * (ell + ell_ctr) + 1):end, :) * null(Bini, tol), tol);
```

Note: the horizon should be $T \geq 2\ell + \ell_c tr$, where $\ell_c tr$ is the controllability index (upper bound n).

```
function BC = B2BC(B, T, ctol),
<<default-ctol>>
[p, m] = size(B); ell = lag(B); ell_ctr = n;
BT = B2BT(B, ell + ell_ctr + T);
BC = BT2BC(BT, p + m, ctol);
```

BT2BP and B2BP — periodic subbehavior

```
function BP = BT2BP(BT, q, ctol);
<<default-ctol>>
[c, m, ell, n] = BT2c(BT, q, ctol); p = q - m;

T = size(BT, 1) / q; t = (1:T)';
[UT, YT] = BT2UYT(BT, m, p); P = [UT; YT];

%% ??? TODO
z = exp(i * linspace(0, 2 * pi, T));
for k = 1:length(z)
  A  = [[zeros(m * T, p); -kron(z(k) .^ t, eye(p))] P];
  hg = pinv(A, tol) * [kron(z(k) .^ t, eye(m)); zeros(p * T, m)];
  Hh(:, :, k) = hg(1:p, :);
end

function BP = B2BP(B, T)
% TODO
```

4.4 Analysis

lag — find the lag of a system, given as an ss object

```
function ell = lag(B, tol)
<<default-tol>>
[p, m] = size(B); O = obsv(B); r = rank(O, tol);
for ell = 0:r, if rank(O(1:p * ell, :), tol) == r, break, end, end
```

`B2c` — find the system's complexity from an `ss` object

```
function [c, m, ell, n] = B2c(B, tol);
<<default-tol>>
[p, m] = size(B); C = ctrb(B); n = rank(C, tol); ell = lag(B, tol); c = [m, ell, n];
```

`BT2c` — find the system's complexity from its restricted behavior

```
function [c, m, ell, n] = BT2c(BT, q, ctol);
<<default-ctol>>
if isscalar(ctol),
  tol = ctol; T = size(BT, 1) / q;
  r1 = rank(BT, tol); r2 = rank(BT2BT(BT, q, T - 1, ctol), ctol);
  mn = [T 1; T-1 1] \ [r1; r2];
  m = round(mn(1)); n = round(mn(2));
  if n > 0
    for ell = 1:n, if rank(BT2BT(BT, q, ell, ctol)) == m * ell + n, break, end, end
  else, ell = 0; end
  c = [m, ell, n];
else,
  c = ctol; [m, ell, n] = unpack_c(c);
end
```

Note: the horizon should be $T \geq \ell + 1$.

`BT2ells` — find the system's lag structure from its restricted behavior

```
function ells = BT2ells(BT, q, tol);
<<default-tol>>
T = size(BT, 1) / q; p = 0; m_ = q; ells = []; d_ = 0;
for t = 1:T
  d = rank(BT(1:q*t, :), tol); m = d - d_; dm = m_ - m;
  if dm, ells = [ells, (t-1) * ones(1, dm)]; m_ = m; p = p + dm; end
  d_ = d;
end
```

`is_lti` — check if B_T represents a bounded complexity LTI system

```
function ans = is_lti(BT, q, c, tol)
<<default-tol>>
ans = all(dist_lti(BT, q, c) <= tol);
```

`dist_lti` — distance of to the set of bounded complexity LTI systems

```
function [d, BTh] = dist_lti(BT, q, c)
Rh = BT2R(BT, q, c);
BTh = R2BT(Rh, q, size(BT, 1) / q);
d = Bdist(BT, BTh, q);
```

Complexity selection from a complexity–accuracy trade-off: $e(d)$

```
function e = error_tradeoff(BT, q, m, n_range)
np = length(d_range); p = q - m;
for i = 1:np, e(i) = dist_lti(BT, q, [m, ceil(n_range(i) / p), n_range(i)]); end
```

`Bdist` — find the distance between two systems

```
function [d, B1T, B2T] = Bdist(B1, B2, q, T, ctol)
<<default-horizon>>
if isa(B1, 'ss'), B1T = B2BT(B1, T); q = sum(size(B1)); else, B1T = B1; end
if isa(B2, 'ss'), B2T = B2BT(B2, T); q = sum(size(B2)); else, B2T = B2; end
if exist('q') && ~isempty(q)
  T = min(size(B1T, 1), size(B2T, 1)) / q;
  <<default-ctol>>
  B1T = BT2BT(B1T, q, T, ctol); B2T = BT2BT(B2T, q, T, ctol);
end
d = subspace(B1T, B2T);
```

The distance depends on the horizon T. The default horizon is:

```
if ~exist('T', 'var') || isempty(T), T = 100; end % <default-horizon>
```

When `B1` and/or `B2` are not `ss`, they should represent $\mathscr{B}^1|_{T_1}$ and $\mathscr{B}^2|_{T_2}$. In this case, $T = \min\{T_1, T_2\}$.

`equal` — check if two systems are equal

```
function ans = equal(B1, B2, q, tol, T)
<<default-tol>>
<<default-horizon>>
if ~exist('q', 'var'), q = []; end
[d, B1T, B2T] = Bdist(B1, B2, q, T);
ans = d < tol & size(B1T, 2) == size(B2T, 2);
```

`isunctr` — check controllability using

The function implements two methods—one based on (B_T) and one based a kernel representation—see [**van-dooren**].

```
function n_unctr = isunctr(BTorR, q, ctol)
<<default-ctol>>
if size(BTorR, 2) < size(BTorR, 1)
  BT = BTorR; % the given representation is BT
  [BC, c] = BT2BC(BT, q, ctol);
  n_unctr = c(1) * size(BC, 1) / q + c(3) - size(BC, 2);
else
  R = BTorR; % the given representation is R
  [R, ells] = R2Rp(R, q, ctol); n = sum(ells); ell = max(ells);
  M = multmat(R, q, 2 * ell + n); Mc = M(:, q * ell + 1:(q + n) * ell);
  n_unctr = size(Mc, 2) - rank(Mc, tol);
end
```

`distunctr` — distance to uncontrollability

Computes the distance to the set of uncontrollable systems with at least n_c uncontrollable modes. The default value for n_c is 1.

```
function d = distunctr(BT, q, nc, ctol)
<<default-ctol>>
if ~exist('nc', 'var') || isempty(nc), nc = 1; end
[c, m, ell, n] = BT2c(BT, q, ctol);
Bini = BT(1:q * ell, :); % selection from BT!
```

```
[~, NullBini] = lra(Bini', m * ell + n);
BC = BT((q * 2 * ell + 1):end, :) * NullBini';
[P, R, hatBC] = lra(BC, m * size(BC, 1) / q + n - nc);
d = norm(BC - hatBC, 'fro') / norm(BC, 'fro');
```

`BT2Hinf` — computation of the H_∞-norm

```
function [Hinf, HT, uinf] = BT2Hinf(BT, q, ctol)
<<default-ctol>>
HT = BT2HT(BT, q, ctol);
[u, s, v] = svd(HT); Hinf = s(1); uinf = v(:, 1);
```

`BT2Hz` — evaluation of the transfer function

```
function Hz = BT2Hz(BT, q, z, ctol)
<<default-ctol>>
[c, m] = BT2c(BT, q, ctol); p = q - m;
T = size(BT, 1) / q; t = (1:T)';
[UT, YT] = BT2UYT(BT, m, p); P = [UT; YT];
for k = 1:length(z)
  A  = [[zeros(m * T, p); -kron(z(k) .^ t, eye(p))] P];
  hg = pinv(A, tol) * [kron(z(k) .^ t, eye(m)); zeros(p * T, m)];
  Hz(:, :, k) = hg(1:p, :);
end
```

4.5 Parametric representations

`B2R` — find a kernel representation of a system

$$(A,B,C,D) \xmapsto{\texttt{lsim}} w_\mathrm{d} \xmapsto{\texttt{null}} R$$

```
function R = B2R(B, tol)
<<default-tol>>
[c, m, ell] = B2c(B, tol);
[~, R] = lra(B2BT(B, ell + 1), c2r(c, ell + 1));
```

`R2BT` — construct a basis for the restricted behavior from a kernel representation

$$R \xmapsto{\texttt{R2Rmin}} R_\mathrm{p} \xmapsto{\texttt{multmat}} \mathscr{M}_T(R_\mathrm{p}) \xmapsto{\texttt{null}} \mathscr{B}|_T = \ker \mathscr{M}_T(R_\mathrm{p})$$

```
function BT = R2BT(R, q, T, ctol);
<<default-ctol>>
<<ctol2c>>
MT = multmat(R2Rp(R, q, ctol), q, T);
if length(ctol) == 3
   [~, BT] = lra(MT', q * T - c2r(c, T)); BT = BT';
else
   [~, BT] = lra(MT', ctol); BT = BT';
end
```

```
function BT = R2BT(R, q, T, ctol);
<<default-ctol>>
<<ctol2c>>
R = orth(R', tol)'; Rs = R(1:(q-m), :); Ms = multmat(Rs, q, T - 1);
```

```
GT = [R zeros(size(R, 1), q * (T - ell - 1)); zeros(size(Ms, 1), q) Ms];
if length(ctol) == 3
  [~, BT] = lra(GT', q * T - c2r(c, T)); BT = BT';
else
  [~, BT] = lra(GT', ctol); BT = BT';
end

clear all

q = 3; m = 1; p = q - m; ell = 4; T = 2 * ell + 1;
ells = [ell * ones(1, p-1) 0];
n = sum(ells); c = [m, ell, n];

[B, Rc] = randB(q, ells, 'ells'); BT0 = B2BT(B, T); % eig(B)
R = Rcell2Rmat(Rc, q);
R_ = multmat(Rc, q, ell+1);
R__ = w2R(B2w(B, 500), c);

[BT, MT]  = R2BT(R_, q, T, c); [BT_, GT] = R2BT_(R_, q, T, c);

[Bdist(B, BT) Bdist(B, BT_)]
```

`BT2R` — find a kernel representation from a basis of the restricted behavior

```
function R = BT2R(BT, q, ctol)
<<default-ctol>>
<<ctol2c>>
[~, R] = lra(BT2BT(BT, q, ell + 1, ctol), c2r(c, ell + 1));
```

Note: the horizon should be $T \geq \ell + 1$.

`ss2BT` — construct a basis for the restricted behavior from a state-space representation

$$(A,B,C,D) \mapsto B_T = \Pi_T \begin{bmatrix} 0_{mT\times n} & I_{mT} \\ \mathscr{O}_T(A,C) & \mathscr{T}_T(H) \end{bmatrix} \qquad ((A,B,C,D) \mapsto B_T)$$

```
function BT = ss2BT(B, T);
[p, m] = size(B); n = order(B);
BT = uy2w([zeros(m * T, n) eye(m * T); obsvm(B, T) convm(B, T)], m, p);
```

`BT2ss` — find a state-space representation from a basis of the restricted behavior

```
function [Bh, io] = BT2ss(BT, q, io, ctol);
<<default-ctol>>
<<ctol2c>>
p = q - m;

%% compute state by past/future intersection
Wpf = BT(1:q * (2 * ell + 1), :); R = null(Wpf', tol)'; % selection from BT!
Wp = BT(1:q * ell, :); X = R(:, 1:q * ell) * Wp;         % selection from BT!

%% compute the shifted state
sigmaWp = BT(q + 1:q * (ell + 1), :);
sigmaX = R(:, 1:q * ell) * sigmaWp;
```

```
%% make X and sigmaX minimal by LRA
[u, s, ~] = svd(X); P = u(1:n, :);
Xmin = P * X; sigmaXmin = P * sigmaX;

%% find an input/output partition
if ~exist('io', 'var') || isempty(io)
  IO = BT2IO(BT, q); io = IO(1, :);
end
if q > 1, BT = BT2BT(BT, q, io); end
[UT, YT] = BT2UYT(BT, m, p);
U = UT(m * ell + 1:m * (ell + 1), :);
Y = YT(p * ell + 1:p * (ell + 1), :);

%% compute the state-space parameters
abcd = [sigmaXmin; Y] * pinv([Xmin; U], tol);
Bh = ss(abcd(1:n,      1:n), abcd(1:n,      n+1:end), ...
        abcd(n+1:end, 1:n), abcd(n+1:end, n+1:end), 1);
```

- Notes: the horizon should be $T \geq 2\ell + 1$.

`R2ss` — find an input/state/output representation from a given kernel representation

```
function [B, io] = R2ss(R, q, io, ctol)
<<default-ctol>>
ell1 = size(R, 2) / q; Tmin = 2 * ell1;
BT = R2BT(R, q, Tmin, ctol);
if ~exist('io', 'var') || isempty(io)
  IO = BT2IO(BT, q); io = IO(1, :);
end
[B, io] = BT2ss(BT, q, io, ctol);
```

`R2Rmin` — make a kernel representation shortest-lag

```
function [Rmin, ells] = R2Rmin(R, q, ctol)
<<default-ctol>>
ell1 = size(R, 2) / q; Tmin = 3 * ell1;
[Rmin, ells] = BT2Rmin(R2BT(R, q, Tmin, ctol), q, ctol);
```

- why `Tmin = ell + 1` is not sufficient?

`BT2Rmin` — find a minimal kernel representation from the restricted behavior

```
function [R, ells] = BT2Rmin(BT, q, ctol)
<<default-ctol>>
<<ctol2c>> % [c, m, ell] = BT2c(BT, q, ctol);
R = {}; ells = []; m = q; i = 0;
for L = 1:ell+1
  BL = BT2BT(BT, q, L, ctol); r = rank(BL, tol);
  if (r < m * L + sum(ells))
    N = null(BL', tol)'; Np = multmat(R, q, L);
    if ~isempty(Np), Nnew = rspan_diff(N, Np); else, Nnew = N; end
    for j = 1:size(Nnew, 1) R{i + j, 1} = Nnew(j, :); ells(i + j) = (L - 1); end
    i = i + size(Nnew, 1);
  end
end
```

Notes: the horizon should be $T \geq 2\ell + 1$.

`Rappend` — append one kernel representation to another

```
function R = Rappend(R1, q1, R2, q2)
[p1, ncol] = size(R1); ell1 = ncol / q1 - 1;
[p2, ncol] = size(R2); ell2 = ncol / q2 - 1;

ell = max(ell1, ell2); q = q1 + q2;
R = zeros(p1 + p2, q * (ell + 1));
I = kron(ones(1, ell+1), 1:q1)    + kron(0:ell, q * ones(1, q1));
R(1:p1, I)      = R1;
I = kron(ones(1, ell+1), q1+1:q) + kron(0:ell, q * ones(1, q2));
R(p1+1:end, I) = R2;
```

4.6 Signal processing and open-loop control

`dist` — find the distance from a trajectory to a system

```
function [d, wh] = dist(w, B, ctol)
<<default-ctol>>
if isa(w, 'ss') % distance between systems
  d = Bdist(w, B, ctol);
else            % distance from signal to system
  if iscell(w)
    for i = 1:length(w),
      [d(i) wh{i}] = dist(w{i}, B, ctol);
    end
  else
    [T, q] = size(w);
    if isa(B, 'ss'), BT = B2BT(B, T); else, BT = BT2BT(B, q, T, ctol); end
    wh_vec = ProjBT(vec(w'), orth(BT)); wh = reshape(wh_vec, q, T)';
    d = norm(w - wh, 'fro');
  end
end
```

Exploiting orthonormality of the basis B_T, the projection on $\mathscr{B}|_T$ can be computed by matrix multiplication: $\widehat{w} = B_T B_T^\top w$. For multiple trajectories $w^1, \dots, w^N$ of length-T stack next to each other in a matrix $W := \begin{bmatrix} w^1 & \cdots & w^N \end{bmatrix}$, their simultaneous projection on $\mathscr{B}|_T$ is $\widehat{W} = B_T B_T^\top W$.

```
function wh = ProjBT(w, BT)
wh = BT * BT' * w;
```

`w_in_B` — check if a given signal is a trajectory of a system

The system can be specified either by a state-space representation or its finite horizon behavior.

```
function ans = w_in_B(w, B, tol)
<<default-tol>>
if iscell(w)
```

```
  for i = 1:length(w), ans(i) = w_in_B(w{i}, B, tol); end
else
  ans = dist(w, B) < tol;
end
```

Instead of using `dist`, another possible implementation for checking if $w \in \mathscr{B}|_T$ is rank $\begin{bmatrix} w & B_T \end{bmatrix} = \operatorname{rank} B_T$.

```
[T, q] = size(w);
ans = rank([vec(w') BT(1:q*T, :)], tol) == rank(BT(1:q*T, :), tol); % selection from
```

`w_in_kerR` — check if a given signal is a trajectory of a system defined by $\ker R(\sigma)$

```
function ans = w_in_kerR(w, R, tol)
<<default-tol>>
ans = norm(R * hank(w, size(R, 2) / size(w, 2))) < tol;
```

`w_in_ss` — check if a given signal is a trajectory of a system defined by $\mathscr{B}_{\mathrm{i/s/o}}(A,B,C,D)$

```
function ans = w_in_ss(w, B, tol)
<<default-tol>>
[xini, e] = w2xini(w, B);  ans = e < tol;
```

`w2xini` — find initial state

```
function [xini, e, x] = w2xini(w, B)
[p, m] = size(B); n = order(B); [T, q] = size(w);
if m > 0,
  u = w(:, 1:m); y = w(:, m+1:end); y0 = y - lsim(B, u);
else, y0 = w; end
O = obsvm(B, T); xini = O \ vec(y0');
if nargout > 1, e = norm(vec(y0') - O * xini); end
if nargout > 2,
  B_ = ss(B.a, B.b, eye(n), zeros(n, m), -1);
  x = reshape(convm(B_, T) * vec(u') + obsvm(B_, T) * xini, n, T)';
end

function [Xini, e] = W2Xini(W, B)
[p, m] = size(B); q = m + p; T = size(W, 1) / q;
[U, Y] = BT2UYT(W, m, p);
if m > 0,
  Y0 = Y - convm(B, T) * U;
else, Y0 = Y; end
O = obsvm(B, T); Xini = O \ Y0;
if nargout > 1
  E = Y0 - O * Xini; e = sqrt(sum(E .^ 2));
end
```

`w2wini` — find initial trajectory

```
function [wini, e] = w2wini(wf, BT, Tini, tol)
<<default-tol>>
[Tf, q] = size(wf); T = size(BT, 1) / q;
if ~exist('Tini', 'var') || isempty(Tini), c = BT2c(BT, q); Tini = c(2); end
Bini = BT(1:q * Tini, :); Bf = BT(q * Tini + 1:end, :); % selection from BT!
```

```
g = pinv(Bf, tol) * vec(wf'); wini = reshape(Bini * g, q, Tini)';
if nargout > 1, e = norm(vec(wf') - Bf * g); end
```

- note: q is inferred from the size of w_f

`u2y` — simulation

```
function yf = u2y(BT, q, uf, wini)
T = size(BT, 1) / q; [Tf, m] = size(uf); p = q - m;
if ~exist('wini'), Tini = T - Tf; wini = zeros(Tini, q);
else, Tini = size(wini, 1); end
Bini = BT(1:q * Tini, :); % selection from BT!
[Uf, Yf] = BT2UYT(BT(q * Tini + 1:q * (Tini + Tf), :), m, p);
yf = Yf * (pinv([Bini; Uf]) * [vec(wini'); vec(uf')]);
yf = reshape(yf, p, Tf)';
```

- notes:
  - the horizon T should be $\geq T_{ini} + T_f$
  - if `wini` is not specified, by default it is taken as a zero trajectory of length $T - T_{ini}$

`wgiven2wmissing` — observer

```
function wm = wgiven2wmissing(wg, B, tol)
<<default-tol>>
[p, m] = size(B); q = m + p;
[T, qg] = size(wg); qm = q - qg;

BT = B2BT(B, T); [Bg, Bm] = BT2UYT(BT, qg, qm);
wm = Bm * pinv(Bg, tol) * vec(wg');
wm = reshape(wm, qm, T)';
```

`lqctr` — linear-quadratic tracking control

```
function wh = lqctr(B, wr, v, wp, wf, ctol)
<<default-ctol>>
[Tr, q] = size(wr);
if ~exist('wp'), wp = []; Tp = 0; else, Tp = size(wp, 1); end
if ~exist('wf'), wf = []; Tf = 0; else, Tf = size(wf, 1); end
if ~exist('v') || isempty(v), v = ones(Tr, q); end
vec_v = vec(v'); vec_wr = vec(wr'); T = Tp + Tr + Tf;
if isa(B, 'ss'), BT = B2BT(B, T); else BT = BT2BT(BT, q, T, ctol); end
Bpf = BT([1:q * Tp, q * (Tp + Tr) + 1:end], :);
Br  = BT(q * Tp + 1:q * (Tp + Tr), :);
Br  = vec_v(:, ones(1, size(Br, 2))) .* Br; vec_wr = vec_v .* vec_wr;
if ~isempty(Bpf)
  N = null(Bpf); gp = pinv(Bpf) * [vec(wp'); vec(wf')];
  z = pinv(Br * N) * (vec_wr - Br * gp);
  wh = Br * (gp + N * z);
else
  wh = Br * pinv(Br) * vec_wr;
end
wh = reshape(wh, q, Tr)';
```

4.7 System identification

c_mpum — find the complexity of the most powerful unfalsified model

```
function [c, m, ell, n] = c_mpum(wd, ctol)
<<default-ctol>>
if length(ctol) == 3 % nothing to do
  c = ctol; [m, ell, n] = unpack_c(c); return
end
if ~iscell(wd)
  [Td, q] = size(wd); ell_max = floor((Td + 1) / (q + 1)) - 1;
else
  N = length(wd); for i = 1:N, [Td(i), q] = size(wd{i}); end
  %% find ell_max (different from the formula in the paper)
  Td = [sort(Td, 'descend') 0];
  for i = 1:N
    ell_max = floor((sum(Td(1:i)) + i) / (q + i)) - 1;
    if (Td(i + 1) < ell_max + 1), break, end
  end
end
r1 = rank(hank(wd, ell_max), tol);
r2 = rank(hank(wd, ell_max + 1), tol);
mn = [ell_max 1; ell_max+1 1] \ [r1; r2];
m = round(mn(1)); n = round(mn(2));
for ell = 0:n, if rank(hank(wd, ell), tol) == m * ell + n, break, end, end
c = [m, ell, n];
```

- in case of one trajectory: `ell_max = floor((Td + 1) / (q + 1)) - 1` — see [8]

For consistency with a general naming convention used in the toolbox, we add an alias of `c_mpum`:

```
function [c, m, ell, n] = w2c(wd, ctol)
<<default-ctol>>
[c, m, ell, n] = c_mpum(wd, ctol);
```

w2BT — finite horizon behavior from data

$$\mathscr{W}_\mathrm{d} \xmapsto{\texttt{hank}} \mathscr{H}_T(\mathscr{W}_\mathrm{d}) \xmapsto{\texttt{orth/lra}} \mathscr{B}|_T = \text{image}\, \mathscr{H}_T(\mathscr{W}_\mathrm{d})$$

```
function BT = w2BT(wd, T, ctol)
<<default-ctol>>
<<ctol2c>>
BT = lra(hank(wd, T), m * T + n);
```

w2R — from data to kernel representation

```
function R = w2R(wd, ctol)
<<default-ctol>>
<<ctol2c>>
[~, R] = lra(hank(wd, ell + 1), m * (ell + 1) + n);
```

w2ss — state-space representation of the most powerful unfalsified model

```
function Bh = w2ss(wd, io, ctol)
<<default-ctol>>
```

```
<<ctol2c>>
q = size(wd, 2);
<<default-io>>
BT = w2BT(wd, 2  * ell + 1, c);
Bh = BT2ss(BT, q, io, c);
```

mpum — minimal kernel representation of the most powerful unfalsified model

```
function [R, ells] = mpum(w, ell_max, tol)
<<default-tol>>
if ~iscell(w), [T, q] = size(w); else, [T, q] = size(w{1}); end
if ~exist('ell_max', 'var') || isempty(ell_max)
  ell_max = floor((T + 1) / (q + 1));
end
m = q; p = 0; ells = []; R = {};
for L = 1:ell_max
  H = hank(w, L);
  if rank(H, tol) < m * L + sum(ells)
    N = null(H', tol)'; Np = multmat(R, q, L);
    if isempty(Np), Rnew = N; else, Rnew = perp(Np / N) * N; end
    if ~isempty(Rnew),
        dp = size(Rnew, 1);
        for i = 1:dp, R{p+i} = Rnew(i, :); ells(p+i) = L - 1; end
        p = p + dp; m = m - dp;
    end
  end
end
```

4.8 Auxiliary functions

vec — column-wise vectorization of a matrix

```
function a = vec(A), a = A(:); end
```

uy2w and w2uy — row permutation reordering variables from (u, y) to w and back

```
function w = uy2w(uy, m, p)
q = m + p; T = size(uy, 1) / q; w = zeros(size(uy));
for i = 1:m, w(i:q:end, :) = uy(i:m:m*T, :); end
for i = 1:p, w(m+i:q:end, :) = uy(m*T+i:p:end, :); end
```

```
function [u, y] = w2uy(w, m, p)
q = m + p; T = size(w, 1) / q;
for i = 1:m, u(i:m:m*T, :) = w(i:q:end, :); end
for i = 1:p, y(i:p:p*T, :) = w(m+i:q:end, :); end
```

obsvm — extended observability matrix

$$\mathscr{O}_T(A, C) := \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{T-1} \end{bmatrix} \in \mathbb{R}^{pT \times n}$$

```
function O = obsvm(B, T)
[p, m] = size(B);
O = B.c; for i = 2:T, O = [O; O(end-p+1:end, :) * B.a]; end
```

`convm` — convolution matrix

$$\mathscr{C}_T(H) := \begin{bmatrix} H(0) & 0 & \cdots & 0 \\ H(1) & H(0) & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ H(T-1) & \cdots & H(1) & H(0) \end{bmatrix} \in \mathbb{R}^{pT \times mT}$$

```
function C = convm(B, T)
if isa(B, 'ss')
  [p, m] = size(B); if m == 0, return, end; h = impulse(B, T-1);
else,
  h = B; if length(size(h)) == 3, [TT, p, m] = size(h); else, p = 1; m = 1, end
end
C = zeros(p * T, m * T);
if m > 1
  H = zeros(T * p, m);
  for i = 1:T, H((i - 1) * p + 1:i * p, :) = reshape(h(i, :, :), p, m); end
else
  H = vec(h');
end
for i = 1:T
  C((i - 1) * p + 1:end, (i - 1) * m + 1:i * m) = H(1:(T - i + 1) * p, :);
end
```

`blkhank` — block Hankel matrix

$$\mathscr{H}_L(w) := \begin{bmatrix} w(1) & w(2) & \cdots & w(T-L+1) \\ w(2) & w(3) & \cdots & \\ \vdots & \vdots & & \vdots \\ w(L) & w(L+2) & \cdots & w(T) \end{bmatrix} \in \mathbb{R}^{qL \times (T-L+1)}$$

```
function H = blkhank(w, L)
[q, T] = size(w); if T < q, w = w'; [q, T] = size(w); end, j = T - L + 1;
if j <= 0, H = []; else,
  H = zeros(L * q, j);
  for i = 1:L, H(((i - 1) * q + 1):(i * q), :) = w(:, i:(i + j - 1)); end
end
```

`Pblkhank` — block Hankel matrix projector

```
function wh = Pblkhank(D, L)
[m, n] = size(D); q = m / L; T = L + n - 1; np = T * q;
I = blkhank(reshape(1:np, q, T)', L);
for k = 1:np, ph(k) = mean(D(I == k)); end
wh = reshape(ph, q, T)';
```

`hank` — mosaic Hankel matrix

$$\mathscr{H}_L(w) := \begin{bmatrix} \mathscr{H}_L(w^1) & \cdots & \mathscr{H}_L(w^N) \end{bmatrix} \in \mathbb{R}^{qL \times \sum_{i=1}^{N}(T_i - L+1)}$$

```
function H = hank(w, L)
if ~iscell(w), H = blkhank(w, L); return, end
N = length(w); H = [];
for k = 1:N, H = [H blkhank(w{k}, L)]; end
```

`Phank` — mosaic Hankel matrix projector

```
function wh = Phank(D, L, n)
if ~exist('n'), wh = Pblkhank(D, L); return, end
for k = 1:length(n), wh{k} = Pblkhank(D(:, 1:n(k)), L); D(:, 1:n(k)) = []; end
```

`multmat` — multiplication matrix

The product $C(z) = A(z)B(z)$ of the matrix polynomials $A \in \mathbb{R}^{g\times q_a}[z]$ and $B \in \mathbb{R}^{q_a\times q_b}[z]$ with degrees $\ell_a := \deg A$ and $\ell_b := \deg B$ is

$$C = A\mathscr{M}_{\ell_a+\ell_b+1}(B)$$

where for a matrix polynomial

$$R(z) = R_0 + R_1 z + \cdots + R_\ell z^\ell \in \mathbb{R}^{g\times q}[z]$$

and with degree ℓ, the multiplication matrix $\mathscr{M}_T(R)$ with $T \geq \ell$ block-columns is defined as

$$\mathscr{M}_T(R) := \begin{bmatrix} R_0 & R_1 & \cdots & R_\ell & & & \\ & R_0 & R_1 & \cdots & R_\ell & & \\ & & \ddots & \ddots & & \ddots & \\ & & & R_0 & R_1 & \cdots & R_\ell \end{bmatrix} \in \mathbb{R}^{g(T-\ell)\times qT}.$$

For a set of polynomials $\mathscr{R} := \{R^1, \ldots, R^N\}$, where $R^i \in \mathbb{R}^{g_i\times q}[z]$ and has degree ℓ_i, the multiplication matrix $\mathscr{M}_T(\mathscr{R})$ with $T \geq \max(\ell_1, \ldots, \ell_N)$ block-columns is defined as

$$\mathscr{M}_T(\mathscr{R}) := \begin{bmatrix} \mathscr{M}_T(R^1) \\ \vdots \\ \mathscr{M}_T(R^N) \end{bmatrix} \in \mathbb{R}^{(gTN-\sum_{i=1}^N \ell_i)\times qT}.$$

```
function M = multmat(R, q, T)
if iscell(R)
  N = length(R); M = [];
  for i = 1:N, M = [M; multmat(R{i}, q, T)]; end
else
  [g, nc] = size(R); ell = nc / q - 1;
  M = zeros(g * (T - ell), T * q);
  for i = 1:(T- ell)
    M(g * (i - 1) + 1:g * i, (1:(ell + 1) * q) + (i - 1) * q) = R;
  end
end
```

`Pmultmat` — multiplication matrix projector

```
function Rh = Pmultmat(D, p, q)
[m, n] = size(D); L = m / p; T = n / q; ell = T - L;
I = multmat(reshape(1:(p*(ell + 1)), p, ell + 1), q, T);
for k = 1:p*(ell + 1), ph(k) = mean(D(I == k)); end
Rh = reshape(ph, p, ell + 1);
```

`rspan_intersect` — intersection of subspaces defined by the row-spans of matrices

- row span C is the intersection of the subspaces row span A and row span B

```
function C = rspan_intersect(A, B, tol)
<<default-tol>>
Ap = null(A, tol); Bp = null(B, tol); C = null([Ap Bp]', tol)';
```

`in_rspan` — check if vector(s) are in the rowspan of a matrix

- $a_i \in$ row span B, where a_i is the ith row of A

```
function [ans, E, U] = in_rspan(A, B, tol)
<<default-tol>>
U = A * pinv(B); E = A - U * B; ans = norm(E) < tol;
```

`is_subspace` — check if $(A) \subset (B)$

```
function ans = is_subspace(A, B, tol)
<<default-tol>>
ans = norm(A - B * pinv(B) * A) < tol;
```

`BTintersect` — intersection of subspaces defined by the column-spans of matrices

- image B is intersection of the subspaces image B^1 and image B^2

```
function BT = BTintersect(BT1, BT2, tol)
<<default-tol>>
BT = rspan_intersect(BT1', BT2', tol)';
```

`perp` — row span orthogonal complement

- row span $A' = (\text{row span } A)^\perp$

```
function Ap = perp(A, tol)
<<default-tol>>
[u, s, v] = svd(A); r = sum(diag(s) > tol); Ap = v(:, (r + 1):end)';
```

`rspan_diff` — difference of subspaces

- row span C is the difference of the subspaces row span A and row span B

```
function C = rspan_diff(A, B, tol)
<<default-tol>>
C = perp(B / A) * A;
```

`lra` — low-rank approximation

```
function [P, R, dh] = lra(d, rtol)
<<default-tol>>
if ~exist('rtol', 'var') || isempty(rtol), rtol = tol; end
[u, s, v] = svd(d);
if rtol < 1, r = sum(diag(s) > rtol); else r = rtol; end
P = u(:, 1:r); R = u(:, (r + 1):end)';
if nargout > 2, dh = u(:, 1:r) * s(1:r, 1:r) * v(:, 1:r)'; end
```

`check` — print PASS / FAIL

```
function check(a), if a, disp('PASS'), else, disp('FAIL'), end
```

`randB` — random LTI systems with a specified complexity

```
function [B, R, BT] = randB(q, c_or_ells, opt)
if exist('opt') | length(c_or_ells) ~= 3 % specification of the lag structure
    ells = c_or_ells; [c, m, ell, n] = ells2c(ells, q); p = q - m;
else % complexity specification
    c = c_or_ells; [m, ell, n] = unpack_c(c); p = q - m; ells = c2ells(c, q);
end
for i = 1:p, R{i} = rand(1, q * (ells(i) + 1)); end
% B = R2ss(R, q, 1:q, c);
BT = R2BT(R, q, 2 * (ell + 1), c); B = BT2ss(BT, q, 1:q, c);
```

`ells2c` and `c2ells` — conversion between complexity and lag structure

```
function [c, m, ell, n] = ells2c(ells, q)
n = sum(ells); ell = max(ells); p = length(ells);
m = q - p; c = [m, ell, n];
```

The complexity does not fully specify the lag structure. Thus, the transformation $c \mapsto (\ell_1,\dots,\ell_p)$ is not well defined. The function `c2ells` chooses a particular lag structure that matches the given complexity (m,ℓ,n): $\lfloor n/\ell \rfloor$ annihiltors have degree ℓ and if $n < p\ell$ and the remainder of the division n/ℓ is nonzero, one annihilator has degree equal to the remainder, while rest of the annihilators if any have degree zero.

```
function ells = c2ells(c, q)
[m, ell, n] = unpack_c(c); p = q - m;
if (n < 0) | (n > p * ell), error('incorrect complexity'); end
ells = zeros(1, p); k = floor(n / ell); ells(1:k) = ell; ells(k + 1) = mod(n, ell);
```

References

[1] J. C. Willems. “From time series to linear system—Part I. Finite dimensional linear time invariant systems, Part II. Exact modelling, Part III. Approximate modelling”. In: *Automatica* 22, 23 (1986, 1987), pp. 561–580, 675–694, 87–115.

[2] J. C. Willems. “The behavioral approach to open and interconnected systems: Modeling by tearing, zooming, and linking”. In: *Control Systems Magazine* 27 (2007), pp. 46–99.

[3] I. Markovsky and F. Dörfler. “Behavioral systems theory in data-driven analysis, signal processing, and control”. In: *Annual Reviews in Control* 52 (2021), pp. 42–64.

[4] I. Markovsky, L. Huang, and F. Dörfler. “Data-driven control based on behavioral approach: From theory to applications in power systems”. In: *IEEE Control Systems Magazine* 43 (5 2023), pp. 28–68.

[5] I. Markovsky and P. Rapisarda. “Data-driven simulation and control”. In: *Int. J. Control* 81.12 (2008), pp. 1946–1959.

[6] I. Markovsky. “A software package for system identification in the behavioral setting”. In: *Control Eng. Practice* 21 (2013), pp. 1422–1436.

[7] I. Markovsky. “A missing data approach to data-driven filtering and control”. In: *IEEE Trans. Automat. Contr.* 62 (4 2017), pp. 1972–1978. ISSN: 1558–2523.

[8] I. Markovsky and F. Dörfler. “Identifiability in the behavioral setting”. In: *IEEE Trans. Automat. Contr.* 68 (3 2023), pp. 1667–1677.

[9] I. Markovsky and F. Dörfler. “Data-driven dynamic interpolation and approximation”. In: *Automatica* 135 (2022), p. 110008.

[10] T. Söderström. *Errors-in-Variables Methods in System Identification*. Springer, 2018.

[11] I. Markovsky and B. De Moor. “Linear dynamic filtering with noisy input and output”. In: *Automatica* 41.1 (2005), pp. 167–171.

[12] M. Bisiacco, M.-E. Valcher, and J. C. Willems. “A Behavioral Approach to Estimation and dead-beat observer design with applications to state-space models”. In: *IEEE Trans. Automat. Contr.* 51.11 (2006), pp. 1787–1797.

[13] I. Markovsky. *Low-Rank Approximation: Algorithms, Implementation, Applications*. Springer, 2019.

[14] B. De Moor et al. “DAISY: A Database for Identification of Systems”. In: *Journal A* 38.3 (1997). Available from `http://homes.esat.kuleuven.be/~smc/daisy/`, pp. 4–5.

[15] D. Knuth. *Literate programming*. Cambridge University Press, 1992.

[16] N. Ramsey. “Literate programming simplified”. In: *IEEE Software* 11 (1994), pp. 97–105.

[17] C. Dominik. *The org mode 7 reference manual*. Network theory, 2010.

[18] I. Markovsky. “Structured low-rank approximation and its applications”. In: *Automatica* 44.4 (2008), pp. 891–909.

[19] I. Markovsky and K. Usevich. “Software for weighted structured low-rank approximation”. In: *J. Comput. Appl. Math.* 256 (2014), pp. 278–292.

A Auxiliary functions

The functions presented in this section are used internally for the implementation of the main functions of the toolbox, presented in Section 3, and are not needed for using the toolbox however they may be useful for extending the functionality of the toolbox.

The function `vec` is the column-wise vectorization operator $x = \text{vec}(X)$. The map $w \mapsto \left[\begin{smallmatrix} u \\ y \end{smallmatrix}\right]$, where w is the column vector with block elements $w(1),\ldots,w(T)$ and similarly u and y are the column vectors of the sequential input and output samples is implemented in the function `w2uy`:

```
w = reshape(1:q*T, q, T)'; vec_w = vec(w');
[vec_u, vec_y] = w2uy(vec_w, m, p);
check(norm(vec_w - uy2w([vec_u; vec_y], m, p)) == 0)
```

Applied on $W := \begin{bmatrix} w^1 & \cdots & w^N \end{bmatrix}$, `w2uy` gives $U := \begin{bmatrix} u^1 & \cdots & u^N \end{bmatrix}$ and $Y := \begin{bmatrix} y^1 & \cdots & y^N \end{bmatrix}$.

The extended observability matrix $\mathscr{O}_T(A,C)$ and convolution matrix $\mathscr{T}_T(H)$ are constructed by `obsvm` and `convm`, respectively:

```
check(norm(obsvm(B, n) - obsv(B)) < tol)

C = convmtx(impulse(B(1, 1), n-1), n);
check(norm(convm(B(1, 1), n) - C(1:n, 1:n)) < tol)
```

The Hankel matrix $\mathscr{H}_T(w)$ constructor is `blkhank`:

```
H = blkhank([1 2 3 4 5], 3);
```

and the projection on the set of Hankel matrices is implemented in `Pblkhank`:

```
w = Pblkhank([1 2 3; 2 3 4; 3 4 5], 3);
```

More generally `hank` accepts as an input a cell array and constructs the mosaic Hankel matrix $\mathscr{H}_T(\mathscr{W})$:

```
H = hank({[1 2 3 4], [5 6 7]}, 3);
```

and `Phank` implements the projection on the set of mosaic Hankel matrices:

```
w = Phank([1 2 5; 2 3 6; 3 4 7], 3);
```

The time-series multiplication matrix $\mathscr{M}_T(R)$ is implemented in `multmat`:

```
M = multmat([1 2 3], 1, 5);
```

and the corresponding projector in `Pmultmat`:

```
R = Pmultmat([1 2 3 0 0; 0 1 2 3 0; 0 0 1 2 3], 1, 1);
```

Matrix polynomial product is done by

```
A = [1 2 , 5 6; 3 4 , 7 8]; q = 2; ell = 1;
C = A * multmat2(A, q, 2 * ell + 1);
```

Next, we introduce operations with subspaces. The intersection of two subspaces, defined by the row-spans of given matrices, is implemented in `rspan_intersect`:

```
A = rand(2, 5); B = [rand(1, 5); rand(2) * A];
C = rspan_intersect(A, B); check(rank([A; C]) == 2)
```

The orthogonal complement of a subspace, defined as the image of a given matrix, is implemented in `perp`:

```
A = rand(2, 5); Ap = perp(A);
check(size(rspan_intersect(A, B), 1) == 0)
```

Finally, the difference of subspaces, defined as row-spans of given matrices, is implemented in `rspan_diff`:

```
A = rand(2, 5); B = [rand(1, 5); rand(2) * A];
C = rspan_diff(B, A); check(rank([B(1, :); C]) == 1)
```

Unstructured low-rank approximation is implemented in `lra`:

```
D = rand(5, 5); [R, P, dh] = lra(D, 2);
```

and generalized low-rank approximation is implemented in `glra`:

```
dh_ = glra(D, 4, 1);
```

Hankel structured low-rank approximation in implemented in the function `slra`:

```
[ph, info] = slra(p, s, r);
```

of the SLRA package [18, 19], which is not included in the Behavioral Toolbox and should be installed separately. The `slra` function is used for optimization based approximate system identification, see [6].