Exercises for the SOCN course "Behavioral approach to system theory"

Ivan Markovsky

This document illustrates and extends the material presented in the lectures of the course by providing computational examples and 40 exercises for independent work. As Richard Hamming said "the purpose of computing is insight no numbers" [5]. The value of the exercises is not the obtained solutions but the thought process of coming up with them: the ideas and attempted ideas that one has when doing the exercises. Thus, although sample solutions are provided, it is important to attempt the exercises independently before consulting with others or reading the solutions.

The exercises are organized thematically and gradually build knowledge and tools. Therefore it is recommended to do them sequentially. Section 1 gives examples of alternative direct data-driven solutions to model-based ones. The examples—representation of the restricted behavior $\mathscr{B}|_T$, computing the lag of the system, and finding a kernel representation of a given system $\mathscr{B}$—are meaningful problems on their own. Section 3 develops a fully featured implementation of the data-driven interpolation and approximation algorithm presented in the lectures. Sections 4–8 show applications of to algorithm to simulation, missing data estimation, errors-in-variables smoothing, classical Kalman smoothing, state transition, and least-squares optimal tracking control. Appendix A introduces the Hankel, mosaic-Hankel, and polynomial multiplication matrices, which are used in the exercises.

1 A first glimpse of data-driven methods

The problems considered involve a true data-generating system (sometimes called "plant" in the context of control). This system may be explicitly given by a parametric representation or it may be implicitly specified by data. The data consists of one or more trajectories of the system and prior knowledge, such as linear time-invariant dynamics, the number of inputs, and order. Problems and corresponding solution methods starting from a representation of the system are called *model-based*. Problems and corresponding solution methods starting from data are called *data-driven*. A data-driven problem can be reduced to a corresponding model-based problem by model identification. Methods that solve the data-driven problem by first identifying a model and then using a model-based method are called *indirect*. Methods that solve the data-driven problem directly without parametric model identification are called *direct*. Before diving into data-driven problems, this section shows that some model-based problems have simple data-driven solution, *i.e.*, even though a parametric model is given, it may be easier to solve the model-based problem by simulating data and using a direct data-driven method. "Easier" means requiring less human effort (less thinking and coding). The approach of using simulation and direct data-driven method is computationally inefficient.

Representation of the restricted behavior $\mathscr{B}|_T$

Exercise 1 (From (A,B,C,D,Π) to a basis of $\mathscr{B}|_T$). Given an input/state/output representation $\mathscr{B} = \mathscr{B}_{\rm ss}(A,B,C,D,\Pi)$ of a linear time-invariant system $\mathscr{B}$ and a natural number T, find a basis for the restricted behavior $\mathscr{B}|_T$ and implement the solution in a function `B = ss2B(ss)`. Do the exercise in two different ways:

1. *model-based* — using the parameters A,B,C,D,Π to obtain an explicit formula for the basis, and
2. *data-driven* — using a simulated trajectory $w_{\rm d} \in \mathscr{B}$.

Model-based solution SOL

For any trajectory $w \in \mathscr{B}|_T$, there is an initial state $x(1) = x_{\rm ini} \in \mathbb{R}^n$, such that

$$w(t) = \Pi \begin{bmatrix} u(t) \\ y(t) \end{bmatrix}, \quad x(t+1) = Ax(t) + Bu(t), \quad y(t) = Cx(t) + Du(t), \quad \text{for } t = 1,2,\ldots,T.$$

This system of equations can be written more compactly as

$$w = \underbrace{\Pi_T \begin{bmatrix} 0_{mT\times n} & I_{mT} \\ \mathscr{O}_T(A,C) & \mathscr{C}_T(H) \end{bmatrix}}_{M_T(A,B,C,D,\Pi)\in\mathbb{R}^{qT\times(n+mT)}} \begin{bmatrix} x_{\text{ini}} \\ u \end{bmatrix}, \tag{1}$$

where $\Pi_T \in \mathbb{R}^{qT\times qT}$ is a permutation matrix (determined by Π), $\mathscr{O}_T(A,C) \in \mathbb{R}^{pT\times n}$ is the extended observability matrix

$$\mathscr{O}_T(A,C) := \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{T-1} \end{bmatrix} \in \mathbb{R}^{pT\times n} \tag{2}$$

with T block rows and

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

is the *convolution matrix* with T block rows constructed from the impulse response of the system

$$H(0) = D, \quad H(t) = CA^{t-1}B, \quad \text{for } t = 1,2,\ldots \tag{3}$$

It follows that

$$\mathscr{B}|_T = \text{image}\, M_T(A,B,C,D,\Pi). \tag{4}$$

The state-space representation of the restricted behavior (4) is a map from the initial condition x_{ini} and the input u to a trajectory $w \in \mathscr{B}|_T$. Vice versa, to every $w \in \mathscr{B}|_T$, $T \geq \ell$, correspond unique x_{ini} and u.

```
function BT = basis_BT_mb(B, T)
[p, m] = size(B); n = order(B);
O = B.c; for i = 2:T, O = [O; O(end, :) * B.a]; end
h = impulse(B, T); C = convmtx(h, T); C = C(1:T, :);  % SISO case only
M = [zeros(m * T, n) eye(m * T); O C]; BT = zeros(size(M));
BT(1:2:end) = M(1:T, :); BT(2:2:end) = M(T+1:end, :); % SISO case only
```

In order to validate `basis_BT_mb` numerically, we take an example system

```
n = 5; p = 1; m = 1; B = drss(n, p, m); % random example system
```

simulate a random trajectory of the system

```
Ts = 10; us = rand(Ts, m); xs_ini = rand(n, 1);
ys = lsim(B, us, [], xs_ini); ws = [us ys];
```

and verify that `ws` is in the image of `BT`

```
BT_mb = basis_BT_mb(B, Ts); wst = ws';
rank(BT_mb) == rank([BT_mb wst(:)]) % -> 1
```

Remark 1. The representation (4) shows that, for $T \geq \ell$

$$\dim \mathscr{B}|_T = \text{rank}\, M_T(A,B,C,D,\Pi) = mT + n. \tag{5}$$

Formula (5) explains why $c = (m,\ell,n)$ is a measure of the model's complexity. Intuitively, the system's complexity is related to its size—the more trajectories $\mathscr{B}$ allows, the more complex it is. For LTI systems, the complexity is then determined by the dimension of $\mathscr{B}$. Formula (5) shows that the dimension of $\mathscr{B}|_T$ is an affine function of T with offset n and slope m. For large T, meaning $T \geq \ell$, the complexity is therefore dominated by the number of inputs m. For systems with equal number of inputs, the more complex system is the one with the larger order.

Data-driven solution **SOL**

In the data-driven solution, first we simulate a random sufficiently long trajectory w_d of the given model $\mathscr{B}$.

```
function wd = B2wd(B, Td)
[p, m] = size(B); n = order(B);
ud = rand(Td, m); xini = rand(n, 1);
yd = lsim(B, ud, [], xini); wd = [ud yd];
```

Then, we form the Hankel matrix $\mathscr{H}_T(w_\mathrm{d})$ and compute a basis of image $\mathscr{H}_T(w_\mathrm{d})$. The "sufficiently long" is determined by the condition that $\mathscr{H}_T(w_\mathrm{d})$ has at least as many columns as rows. This gives

$$T_\mathrm{d} \geq T_\mathrm{min} := (m+1)T + n - 1, \tag{6}$$

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

where m is the number of inputs, n is the order of the system, and T is the length of the trajectory w. Then, the randomness of w_d guarantees that almost surely image $\mathscr{H}_T(w_\mathrm{d}) = \mathscr{B}|_L$. In order to obtain a basis for image $\mathscr{H}_T(w_\mathrm{d})$, we use the `orth` function of Matlab. This leads to the following function:

```
function BT = basis_BT_dd(B, T)
[p, m] = size(B); n = order(B); <<define-Tmin>>
BT = orth(blkhank(B2wd(B, Tmin(T)), T);
```

where `blkhank` is a Hankel matrix constructor, see Appendix A.

As in the model-based solution, we validate `basis_BT_dd` numerically

```
BT_dd = basis_BT_dd(B, Ts);
rank(BT_dd) == rank([BT_dd wst(:)]) % -> 1
```

Moreover, we verify that `BT_mb` and `BT_dd` have the same image:

```
rank(BT_mb) == rank([BT_mb BT_dd])  % -> 1
```

Computing the lag of a given system

Exercise 2 (Finding $\ell(\mathscr{B})$). Given an input/state/output representation $\mathscr{B} = \mathscr{B}_\mathrm{ss}(A,B,C,D,\Pi)$, find the lag $\ell(\mathscr{B})$ and implement the solution in a function `ell = lag(B)`. Again, do the exercise in two different ways:

1. *model-based* — using the parameters A,B,C,D,Π, and
2. *data-driven* — using a simulated trajectory $w_\mathrm{d} \in \mathscr{B}$.

Model-based solution **SOL**

For the computation of $\ell(\mathscr{B})$ using the state-space representation, we construct the extended observability matrix $\mathscr{O}_T(A,C)$ for $T = 1,\ldots,n$. Then, $\ell(\mathscr{B})$ is equal to the minimal T for which $\operatorname{rank}\mathscr{O}_T(A,C) = n$.

```
function ell = lag_mb(B)
[p, m] = size(B); n = order(B); O = B.c;
for ell = 1:n
  if rank(O) == n, break, end
  O = [O; O(end - p + 1:end, :) * B.a];
end
```

Data-driven solution **SOL**

For the computation of $\ell(\mathscr{B})$ using a trajectory w_d, we construct the Hankel matrix $\mathscr{H}_T(w_\mathrm{d})$ for $T = 1,\ldots,n$. Then, $\ell(\mathscr{B})$ is equal to the minimal T for which $\operatorname{rank}\mathscr{H}_T(w_\mathrm{d}) = Tm+n$.

```
function ell = lag_dd(B)
[p, m] = size(B); n = order(B); <<define-Tmin>>
wd = B2wd(B, Tmin(T));
for ell = 1:n,
  if rank(blkhank(wd, ell)) == ell * m + n, break, end
end
```

As a numerical check

```
lag_mb(B) == lag_dd(B) % -> 1
```

Finding a kernel representation

Exercise 3 (From an input/state/output to a kernel representation `ss2ker`). Given a linear time-invariant system $\mathscr{B}$, specified by an input/state/output representation $\mathscr{B}_{\mathrm{ss}}(A,B,C,D,\Pi)$, find a kernel representation $\mathscr{B} = \ker R(\sigma)$ and implement the solution in a function `R = ss2ker(ss)`. Again, do the exercise in two different ways:

1. *model-based* — using the parameters A,B,C,D,Π to obtain an explicit formula for R, and

2. *data-driven* — using a simulated trajectory $w_\mathrm{d} \in \mathscr{B}$.

Model-based solution **SOL**

The model-based solution is simpler in the single-input single-output case. First, we convert the state-space representation to a transfer function. In the single-input single-output case $H = q/p$, where q and p are polynomials. Then, R is constructed from the coefficients of q and p, using the fact that $R = \begin{bmatrix} q & -p \end{bmatrix}$. Note, however, that by default Matlab stores the coefficients of a polynomial in a decreasing order, while we choose to store them in ascending order, so that a conversion by flipping them is also needed.

```
function R = ss2R_mb(B) % SISO case
[q, p] = tfdata(tf(B), 'v'); n = order(B);
R = zeros(1, 2 * (n + 1)); R(1:2:end) =  fliplr(q); R(2:2:end) = -fliplr(p);
```

Data-driven solution **SOL**

Again, we start by simulating a random sufficiently long trajectory w_d. The kernel representation manifest itself in a basis for the left kernel of the Hankel matrix $\mathscr{H}_{\ell+1}(w_\mathrm{d})$, where ℓ is the lag of the system. Let the rows of

$$R =: \begin{bmatrix} R_0 & R_1 & \cdots & R_\ell \end{bmatrix} \in \mathbb{R}^{g\times q(ell+1)}$$

form a basis for the left kernel of $\mathscr{H}_{\ell+1}(w_\mathrm{d})$. The kernel representation $R(\sigma)w = 0$ of the system is given by the polynomial

$$R(z) := R_0 + R_1 z + \cdots + R_\ell z^\ell.$$

This leads to the function:

```
function R = ss2R_dd(B)
<<define-Tmin>>
[p, m] = size(B); n = order(B); <<define-Tmin>>
ell = lag_dd(B); wd = B2wd(B, Tmin(ell + 1));
R = w2R(wd, ell);

function R = w2R(wd, ell)
R = null(blkhank(wd, ell + 1)')';
```

In the single-input single-output case $\ell = n$ and the dimension of the left kernel of $\mathscr{H}_{\ell+1}(w_\mathrm{d})$ is one. Indeed, $\operatorname{rank}\mathscr{H}_{\ell+1}(w_\mathrm{d}) = 2n+1$ and $\mathscr{H}_{\ell+1}(w_\mathrm{d})$ has $2n+2$ rows. We validate numerically that the results of `ss2R_mb` and `ss2R_dd` are equivalent:

```
R_mb = ss2R_mb(B); R_mb = R_mb / R_mb(1);
R_dd = ss2R_dd(B); R_dd = R_dd / R_dd(1);
norm(R_mb - R_dd) % -> 1e-08
```

Summary and discussion

In the data driven solutions of both problems—representation of the restricted behavior $\mathscr{B}|_T$ and finding a kernel representation—the main tool is the Hankel matrix. In the former problem, we used the image of $\mathscr{H}_T(w_\mathrm{d})$. In the latter problem, we used the left kernel of $\mathscr{H}_{\ell+1}(w_\mathrm{d})$. Note also the correspondence between a basis vector R for the left kernel of $\mathscr{H}_{\ell+1}(w_\mathrm{d})$ and the parameter R of the kernel representation of the system.

In Exercise 3, we are computing what is called the *most powerful unfalsified model (MPUM)* for the data w_d. Under identifiability assumptions (satisfied in the exercise almost surely due to the "sufficiently long" condition and the randomness of the data), the MPUM coincides with the true data-generating system. For details about the MPUM and its computation, see [12]. A mini-project explores the recursive computation of the MPUM.

2 Most powerful unfalsified model

MPUM's complexity

Exercise 4 (Finding the MPUM's complexity). Given trajectory $w_\mathrm{d} \in (w_\mathrm{d}^q)^{T_\mathrm{d}}$, find the complexity (m,ℓ,n) of the most powerful unfalsified model $\mathscr{B}_{\mathrm{MPUM}}(w_\mathrm{d})$.

The condition that $\mathscr{H}_T(w_\mathrm{d})$ has at least as many columns as rows imposes a limit on T

$$T_{\max} := \left\lfloor \tfrac{T_\mathrm{d}+1}{q+1} \right\rfloor. \tag{$T_{\max}$}$$

```
Tmax = @(wd) floor((size(wd, 1) + 1) / (size(wd, 2) + 1)); % <define-Tmax>
```

```
function c = c_mpum(wd)
<<define-Tmax>>
T = Tmax(wd); r = rank(blkhank(wd, T));
n = mod(r, T); m = (T   n) / k;
for ell = 1:n,
  if rank(blkhank(wd, ell)) == ell * m + n, break, end
end
c = [m, ell, n];
```

Kernel representation

```
function R = w2R(wd, ell)
if ~exist('ell'), c = c_mpum(wd); ell = c(2); end
R = null(blkhank(wd, ell + 1)')';
```

3 Implementation of the algorithm for data-driven interpolation/approximation

The exercises in this section lead to a fully featured implementation of the method for interpolation/approximation of trajectories of [11]. The interpolation/approximation of trajectories problem is defined as follows:

$$\text{minimize} \quad \text{over } \widehat{w} \quad \|w-\widehat{w}\|_S \quad \text{subject to} \quad \widehat{w} \in \mathscr{B}|_T, \tag{7}$$

where the data-generating system $\mathscr{B}$ is implicitly specified by a *data trajectory* $w_\mathrm{d} \in \mathscr{B}|_{T_\mathrm{d}}$ and the cost function

$$\|w-\widehat{w}\|_S := \|S \odot (w-\widehat{w})\| \tag{8}$$

is the 2-norm of the element-wise product $\odot$ between the weights $S_i(t) \geq 0$ and the approximation errors $w_i(t) - \widehat{w}_i(t)$. Zero weights $S_i(t) = 0$ specify missing data, while infinite weights $S_i(t) = \infty$ specify exact interpolation points.

The method for solving (7) uses the data-driven representation

$$\mathscr{B}|_T = \text{image } \mathscr{H}_T(w_\text{d}), \tag{9}$$

of the system $\mathscr{B}$, where $\mathscr{H}_T(w_\text{d})$ is the Hankel matrix with T-block-rows, constructed from the data w_d. A necessary and sufficient condition for (9) is that w_d is a trajectory of a linear time-invariant system $\mathscr{B}$ and

$$\text{rank } \mathscr{H}_T(w_\text{d}) = \boldsymbol{m}(\mathscr{B})T + \boldsymbol{n}(\mathscr{B}). \tag{10}$$

The result of doing the exercises is a Matlab function `ddint` with the following interface:

```
function [wh, N, g] = ddint(wd, w, S, m, n, l, L, ell)
```

The input argument `wd` corresponds to the data trajectory w_d and `w` corresponds to the to-be-approximated/interpolated trajectory w. Specifying `wd` and `w` is compulsory. The other input arguments are optional. The output argument `wh` corresponds to a solution $\widehat{w}$ of (7). In addition to solving (7), `ddint` implements the following functionality:

- model-based data-driven interpolation/approximation,
- data w_d consisting of multiple trajectories $w_\text{d}^1, \ldots, w_\text{d}^N$,
- specification of the set of all solutions in case of a nonunique solution $\widehat{w}$,
- preprocessing of the data w_d via low-rank approximation, and
- ℓ_1-norm regularization.

The extra features are accessible via the optional arguments as described in the exercises.

Exercise 5 (Basic algorithm `wh = ddint(wd, w)`). Implement and test on simulation examples the basic algorithm for data-driven interpolation and approximation. The first dimension `Td` of `wd` is the number of samples and the second dimension `q` is the number of variables. The number of samples `Td` must be bigger than `q`. The argument `w` specifies the to-be-interpolated/approximated trajectory w. Again, the first dimension represents time and the second dimension corresponds to variables. The missing values of w are specified by `NaN`'s. If an exact interpolant does not exist, an approximate one is computed using the unweighted least-squares cost function (8), *i.e.*, $S_{ij} = 1$, for all i, j.

Exercise 6 (Nonuniqueness of the solution). The interpolation condition is that the interpolant $\widehat{w}$ of w is a trajectory of $\mathscr{B}$. If the interpolant is not unique the set of all interpolants is given by $\widehat{\mathscr{W}} = \widehat{w} + \text{image } N$, where $\widehat{w}$ (returned in the output argument `wh`) is a particular solution and the matrix N specifies the nonuniqueness via its image. Modify `ddint` to compute N and return in the output argument `N`.

Exercise 7 (Model-based interpolation/approximating). In the basic algorithm, the argument `wd` implicitly specifies the data-generating system $\mathscr{B}$ by a trajectory. Implement and test on simulation examples an option of `ddint` to accept as `wd` the model $\mathscr{B}$ specified by an `ss` or `tf` object instead of a trajectory.

Exercise 8 (Data consisting of multiple trajectories). Implement and test on simulation examples an option of `ddint` to accept as `wd` a cell array containing multiple trajectories $w_\text{d}^1, \ldots, w_\text{d}^N$ of the system $\mathscr{B}$.

Exercise 9 (Element-wise weighted cost function). Implement and test on simulation examples an option of `ddint` to accept an element-wise positive matrix $S \in \mathbb{R}^{T \times q}$ that specifies a weighted cost function (8). The weight matrix is passed to `ddint` via the optional argument `S`. The default value of `S` is unit weights (`ones(T, q)`), which corresponds to the ordinary least-squares approximation.

Exercise 10 (Zeros and infinite weights). Modify `ddint` so that

1. interpolation conditions can specified by infinite weights `S(t, i) = inf`, and
2. missing values can be specified by zero weights `S(t, i) = 0`.

Exercise 11 (Noise filtering via low-rank approximation). Include in `ddint` the optional input parameters `m` and `n` that specify the complexity of the data-generating system $\mathscr{B}$. Use the parameters `m` and `n` are, when passed to `ddint`, for checking the condition (10) for the data-driven representation. If the condition is not satisfied, issue a warning message. In case of inexact data `wd` (*i.e.*, $\text{rank } \mathscr{H}_T(w_\text{d}) > mT + n$) use the parameters `m` and `n` for preprocessing of the data matrix $\mathscr{H}_T(w_\text{d})$ by unstructured rank-$mT + n$ approximation.

Exercise 12 (ℓ_1-norm regularization). By default, $\widehat{w}$ is computed by solving the weighted least-squares problem $\|w - Dg\|_S$ with the pseudo-inverse. Modify `ddint` to allow for specification of the optional parameter `l` that adds the regularization term $\mathtt{l} \cdot \|g\|_1$ in the cost function.

Exercise 13 (Recursive computation). Modify the code of `ddint` to allow for the optional parameters `L` (block size) and `ell` (lag of the system) that trigger a recursive computation of $\widehat{w}$ in blocks of `L` samples, where $1 \leq \mathtt{L} \leq T$. The recursive algorithm matches the final conditions of a block with the initial conditions of the following block.

Solutions SOL

The solutions piece together a complete version of the function `ddint`.

```
function [wh, N, g] = ddint(wd, w, S, m, n, l, L, ell)
[T, q] = size(w);
<<default-S>>
<<wd-lti-model>>
<<recursive-computation>>
<<form-the-data-matrix>>
<<given-complexity>>
<<construct-the-system>>
<<equality-constraints-pre-processing>>
<<solve-the-system>>
<<equality-constraints-post-processing>>
<<define-wh>>
<<nonuniqueness-of-solution>>
```

We show the actual Matlab code in a literature programming style [6]. A keyword `<<label>>` is replaced by its definition given elsewhere and marked with `%% <label>`. The replacement of the keywords is done automatically by the `org-babel` package of the Emacs editor [15].

Exercise 5: Basic algorithm `wh = ddint(wd, w)`

In the basic algorithm, we first construct the Hankel matrix $\mathscr{H}_T(w_\mathrm{d})$ of the data w_d

```
Hwd = blkhank(wd, T); % <form-the-data-matrix>
```

and extract in a matrix A the rows of $\mathscr{H}_T(w_\mathrm{d})$ corresponding to the given samples in the to-be-interpolated trajectory w

```
%% <construct-the-system>
w_vec = vec(w'); Ia = find(~isinf(S_vec) & (S_vec ~= 0));
b = w_vec(Ia); A = Hwd(Ia, :);
```

Then, we solve the system of linear equations $Ag = b$, where $b := w(I_\mathrm{given})$, using the pseudo-inverse

```
%% <solve-the-system-for-g>
if ~exist('l') || isempty(l) || l == 0 || isa(wd, 'lti')
  g = pinv(A) * b;
else
  <<solve-the-l1-norm-regularized-problem>>
end
```

In general, we can not use the backslash operator to solve the system $Ag = b$ because A is rank deficient.

Finally, we reconstruct the interpolant $\widehat{w}$ from its vectorized version $\mathscr{H}_T(w_\mathrm{d})g$

```
wh = reshape(Hwd * g, q, T)'; % <define-wh>
```

Exercise 6: Nonuniqueness of the solution

The nonuniqueness of g, given by the null space of $\mathscr{H}_T(w_\mathrm{d})|_{I_\mathrm{given}}$, is due to two sources:

1. the fact that $\mathscr{H}_T(w_\mathrm{d})$ may not be a basis for $\mathscr{B}_T$ (*i.e.*, $\mathscr{H}_T(w_\mathrm{d})$ is not full column rank), and

2. some rows of $\mathscr{H}_T(w_\mathrm{d})$ may be removed in forming $\mathscr{H}_T(w_\mathrm{d})|_{I_\mathrm{given}}$.

Even if the solution $\widehat{w}$ of (7) is unique, g may be nonunique due to the first source of nonuniqueness. The nonuniqueness of $\widehat{w}$ is caused by the second reason — the removed equations from $\mathscr{H}_T(w_\mathrm{d})g = w$ due to the missing elements in w. In order to characterize the "essential" nonuniqueness of g, first we compute a basis for $\mathscr{B}_T$. Let B_T be a matrix formed by stacking next to each other the basis vectors. The essential nonuniqueness of g is given by the nullspace $\mathscr{N}_g$ of B_T. The nonuniqueness of $\widehat{w}$ is then given by $\mathscr{H}_T(w_\mathrm{d})\mathscr{N}_g$.

```
%% <nonuniqueness-of-solution>
if nargout > 1,
  if ~eq
    BT = orth(Hwd); N = Hwd * null(BT(Ia, :));
  else
    <<nonuniqueness-of-solution-with-equality-constraints>>
  end
end
```

Exercise 7: Model-based interpolation/approximating

As we've seen in Section 1 direct data-driven solutions can be used also for solving problems where a model is given by simulating data from the model. Using this idea, we solve the model-based interpolation/extrapolation problem the the following modification of the code:

```
%% <wd-lti-model>
if isa(wd, 'lti')
  m_ = size(wd, 2); n_ = order(wd); Tmin = (m_ + 1) * T + n_;
  ud = rand(Tmin, m_); yd = lsim(wd, ud, [], rand(n_, 1)); wd = [ud yd];
end
```

The number of data points T_d is chosen as the minimal one (see (6) in Section 4). This choice is justified by the exactness of the data w_d and the fact that it minimizes the computational cost.

Exercise 8: Data consisting of multiple trajectories

The only modification needed for the case is multiple trajectories is to use the mosaic-Hankel matrix instead of the Hankel matrix. We could have used `moshank` instead of `blkhank` on the first place (the basic algorithm) because `moshank` is a proper generalization of `blkhank`. However, we allow for a third option: the user can pass as the input argument `wd` their own data matrix (*e.g.*, a trajectory matrix or a preprocessed Hankel matrix). The case (single trajectory, multiple trajectories, or a data matrix) is recognized by the type and size of the input argument `wd`. Then, the modification of `ddint` is:

```
%% <form-the-data-matrix>
if iscell(wd)
  Hwd = moshank(wd, T);
elseif size(wd, 2) == q
  Hwd = blkhank(wd, T);
end
```

Exercise 9: Element-wise weighted cost function

By default, the approximation cost function is the ordinary least-squares, which corresponds to the element-wise weight matrix S with all elements corresponding to the given elements of w equal to 1.

```
%% <default-S>
if ~exist('S') || isempty(S), S = ones(size(w)); end, S(isnan(w)) = 0;
eq = any(isinf(S(:))); % infinite weights <=> equality constraints
```

In the general case of element-wise weighted cost function, we incorporate the weights in the matrices A and b, defining the system of linear equations of the basic algorithm (see Exercise 5).

```
%% <construct-the-system>
w_vec = vec(w'); S_vec = vec(S');
Ia = find(~isinf(S_vec) & (S_vec ~= 0));
b = S_vec(Ia) .* w_vec(Ia);
A = S_vec(Ia, ones(1, size(Hwd, 2))) .* Hwd(Ia, :);
```

The modified ordinary least-squares problem $Ag = b$ gives us then the solution of the original element-wise weighted least-squares problem. This approach has the advantage of keeping the rest of `ddint` (that deals with solving the system of equations) the same as in the unweighted case. This is important later on when regularization, equality constraint, and recursive computation are added.

Exercise 10: Zeros and infinite weights

The solution in Exercise 9 for taking into account element-wise weighted cost function assumes finite weights S_{ij}. It is convenient, however, to specify exact interpolation points, *i.e.*, equality constraints in the weighted least-squares problem, by infinite weights. In this exercises we take into account the infinite weights.

The solution (not shown here) requires a basis for $\mathscr{B}_T$. This is done on the preprocessing step, see code chunk `<<given-complexity>>`. Then, taking into account the equality constraints boils down to pre-processing A and b

```
%% <equality-constraints-pre-processing>
if eq
  Ie = find(isinf(S_vec)); Ae = Hwd(Ie, :); be = w_vec(Ie);
  N = null(Ae); ge = pinv(Ae) * be;
  if isempty(N), wh = reshape(Hwd * ge, q, T)'; return; end
  b = b - A * ge; A = A * N;
end
```

and post-processing g

```
%% <equality-constraints-post-processing>
if eq, g = ge + N * g; end
```

The postprocessing is needs also in case of computation of the set of solutions

```
%% <nonuniqueness-of-solution-with-equality-constraints>
N = Hwd * N * null(A);
```

Exercise 11: Noise filtering via low-rank approximation

Both the numerical rank and the low-rank approximation of the data matrix are computed from the singular value decomposition.

```
%% <given-complexity>
if exist('m') && exist('n') && ~isempty(m) && ~isempty(n)
  r = T * m + n; tol = 1e-12; [U, S, V] = svd(Hwd);
  if s(r) < tol, warning('wd not informative.'); end
  if s(r + 1) > tol, Hwd = U(:, 1:r) * S(1:r, 1:r) * V(:, 1:r)'; end
elseif eq    % in case of equality constraints
  Hwd = orth(Hwd); % a basis for B_T is needed
end
```

Exercise 12: ℓ_1-norm regularization

There are different methods and packages for solving the ℓ_1-norm regularized problem $\min_g \|Ag-b\|+\lambda\|g\|_1$, *e.g.*, Matlab's `lasso` function and the CVX package [**cvx**]. Here we use the CVX package:

```
%% <solve-the-l1-norm-regularized-problem>
cvx_begin quiet
   variable g(size(A, 2), 1)
   minimize(norm(b - A * g) + l * norm(g, 1))
cvx_end
```

Exercise 13: Recursive computation

```
%% <recursive-computation>
if exist('L') && ~isempty(L),
  k = floor((T - ell) / L);
  if k <= 1, wh = ddint(wd, w, S, m, n, l); return, end
  wh = ddint(wd, w(1:(ell+L), :), S(1:(ell+L), :), m, n, l);
  for i = 2:k
    I = ell + (((i - 1) * L + 1):(i * L));
    whi = ddint(wd, [wh(end-ell+1:end, :); w(I, :)], ...
                    [inf(ell, q); S(I, :)], m, n, l);
    wh(I, :) = whi(ell+1:end, :);
  end
  N = []; g = []; return
end
```

Summary and discussion

In this section, we've build the foundation for the experiments in the following sections. Next, we will apply `ddint` for solving well-known problems, such as simulation, smoothing, and tracking control, as well as less well-known problems, such as input estimation and missing data estimation. The mini-projects compare `ddint` with alternative model-based methods in case of noisy data, disturbances, and nonlinear system dynamics.

A quick and dirty way of dealing with equality constraints is to replace the infinite weights in S by a large value (*e.g.*, 10^8). As an optional exercise, compare the solutions obtain by infinite and large values (but finite) values for weights corresponding to exact interpolation points in test examples.

In the stochastic setting of *errors-in-variables* estimation $w=\overline{w}+\widetilde{w}$, where $\overline{w}\in\mathscr{B}|_T$ is the true value of w and the measurement noise $\widetilde{w}$ is zero mean, uncorrelated, white, Gaussian, with element-wise standard deviations `1 / S(t, i)`, finite positive weights `S` specify the noise standard deviation of to-be-approximated "noisy" elements of w. The weighted least-squares cost function corresponds then to the maximum-likelihood estimation criterion, *i.e.*, the solution $\widehat{w}$ of (7) is the maximum-likelihood estimator of w.

4 Simulation

A basic operation a model is used for is simulation. It is defined for a system $\mathscr{B}$ with an input/output partitioning of the variables $w=\begin{bmatrix}u\\y\end{bmatrix}$ as follows: given the system $\mathscr{B}$, the input u, and the "initial condition", find the output y. The "initial condition" can be specified by a "prefix" trajectory w_{ini} of w.

Lemma 2 (Initial condition specification ddctr)**.** *Let $\mathscr{B}\in\mathscr{L}^q$ admit an input/output partition $w=(u,y)$. Then, for any given $w_{ini}\in\mathscr{B}_{T_{ini}}$ with $T_{ini}\geq\boldsymbol{\ell}(\mathscr{B})$ and $u\in(\mathbb{R}^m)^L$, there is a unique $y\in(\mathbb{R}^p)^L$, such that $w_{ini}\wedge(u,y)\in\mathscr{B}|_{T_{ini}+L}$.*

Proof. Two proofs of Lemma 2 are given in **??**. The first one is based on an input/state/output representation of $\mathscr{B}$. The second one is based on a generic basis of $\mathscr{B}|_{T_{\text{ini}}+L}$. □

From the behavioral point of view, simulation is a particular way of parametrizing the elements of $\mathscr{B}$. Equivalently, simulation is the problem of selecting an element $w \in \mathscr{B}$ of the behavior.

Formally, the simulation problem is defined as follows: Given a system $\mathscr{B}$, an input/output partitioning~(**??**), input $u \in (\mathbb{R}^m)^L$, and initial condition $w_{\text{ini}} \in (\mathbb{R}^q)^{T_{\text{ini}}}$,

$$\text{find } y \in (\mathbb{R}^p)^L, \text{ such that } w_{\text{ini}} \wedge \Pi(u,y) \in \mathscr{B}|_{T_{\text{ini}}+L}, \tag{11}$$

see Figure~1 for a graphical illustration.

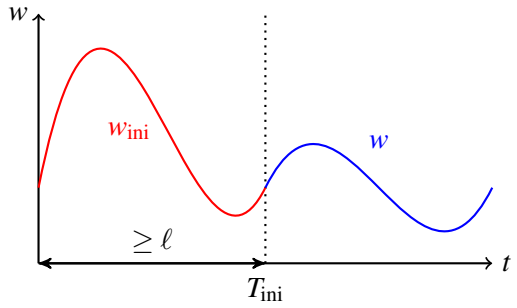

Figure 1: Initial condition for a trajectory $w \in \mathscr{B}$ are specified in the behavioral setting by a prefix trajectory w_{ini} of length $T_{\text{ini}} \geq \boldsymbol{\ell}(\mathscr{B})$. The condition that w is generated from the initial condition specified by w_{ini} is then $w_{\text{ini}} \wedge w \in \mathscr{B}$.

The assumption that w_{ini} is a trajectory of $\mathscr{B}$ guarantees existence of a solution to the simulation problem (11). However, in general, the solution may not be unique. In order to render the solution unique, w_{ini} has to be "sufficiently" long. A sufficient condition for this is that $T_{\text{ini}} \geq \boldsymbol{\ell}(\mathscr{B})$.

The trajectory w_{ini} plays the role of the initial state in the state-space setting. It can be shown that for $T_{\text{ini}} \geq \boldsymbol{\ell}(\mathscr{B})$, the vector w_{ini} of sequential samples is a state vector of the system RW97. Then, problems of estimation of w_{ini} can also be understood as state estimation problems. %{It turns out that quite many papers in Section 6 construct a non-minimal state-space realization using w_{ini}.}

The example given in this section applies the data-driven interpolation and approximation method for solving a data-driven version of the classical model-based simulation problem. The key observation is that simulation is a special interpolation problem in which the given data is "past" inputs and outputs (specifying the initial conditions) and "future" inputs, while the missing values are the to-be-simulated "future" outputs.

Model-based simulation

In order to illustrate the idea on a numerical example, first, we choose a data-generating system $\mathscr{B}$. For this purpose we use Matlab's function `drss`

```
n = 5; p = 1; m = 1;
B = drss(n, p, m);
```

which generates a discrete-time random stable state-space system with a specified order `n`, number of inputs `p`, and number of outputs `m`. In the example, $\mathscr{B}$ is a 5th order (`n = 5`) single-input (`m = 1`) single-output (`p = 1`) linear time-invariant system

Then, we choose an input `us` and initial conditions `xs_ini` for the simulation problem

```
Ts = 10; us = rand(Ts, m); xs_ini = zeros(n, 1);
```

In the example, the simulation horizon is $T_{\text{s}} = 10$ samples, the input is a uniform random process, and the system is initially at rest (zero initial conditions).

Once the system, the input, and the initial conditions are chosen, we obtain the corresponding response y_{s} by

```
ys = lsim(B, us, [], xs_ini); ws = [us ys];
```

Data-driven simulation

In order to do data-driven simulation, we need a "data" trajectory `wd` of the system $\mathscr{B}$. In the example, `wd` is a random trajectories of $\mathscr{B}$ with length $T_\mathrm{d} = 100$ samples

```
Td = 100;
ud = rand(Td, m); xd_ini = rand(n, 1);
yd = lsim(B, ud, [], xd_ini); wd = [ud yd];
```

The length $T_\mathrm{d} = 100$ is chosen arbitrarily, however, it must satisfy the lower bound (6).

The trajectory w represents the to-be-simulation trajectory w_s of the system. In order to account for the initial conditions, w is split into two parts, called "past" and "future". The past is of length n and specifies the initial conditions. The future is of length T_s and contains the simulated response. In the example, the initial conditions are zero, so that the past is set to zero. The future is the given input u_s and `NaN`'s for the to-be-computed output y_s.

```
w = [zeros(n, m + p); [us NaN(Ts, p)]];
```

With `wd` and `w` specified, we are ready to solve the data-driven simulation problem by calling the function `ddint`

```
wh = ddint(wd, w);
```

The simulation output $\widehat{y}_\mathrm{s}$ is obtained from the interpolated samples in `wh`

```
ysh = wh(n+1:end, m+1:end);
```

Finally, we verify that the model-based and the data-driven simulation methods yield the same result

```
norm(ys - ysh) % -> 1e-15
```

Exercises

Exercise 14 (Impulse response computation). An important special case of simulation is impulse response computation. Apply the data-driven simulation method implemented in `ddint` to compute the first $T_\mathrm{s} = 10$ samples of the impulse response of the system directly from the data w_d. Verify that the result obtained by `ddint` is exact by comparing it with the true impulse response computed by model-based simulation.

The System Identification Toolbox of Matlab provides a function `impulseest` for estimation of impulse response from data. Apply it to the problem at hand using the data w_d. Compare the estimation accuracy and computational efficiency of `ddint` and `impulseest`.

Exercise 15 (Step response computation). Another important special case of simulation is step response computation. Apply `ddint` to compute the first $T_\mathrm{s} = 10$ samples of the step response of the system directly from the data w_d. Verify that the result obtained by `ddint` is exact by comparing it with the true step response computed by model-based simulation.

Exercise 16 (Minimum number of data samples T_d). Verify that the minimum number of samples T_d for which `ddint` yields correct (*i.e.*, exact up to the numerical precision) result is T_min, given in (6).

Exercise 17 (Multivariable systems). Compare the model-based (`lsim`) and data-driven (`ddint`) simulation methods on multivariable systems.

Exercise 18 (Multiple data trajectories $w_\mathrm{d}^1, \ldots, w_\mathrm{d}^N$). Try out the feature of `ddint` to use as data `wd` multiple trajectories. This allows one to reduce the length T_d of the experiments. How small can T_d be made by using multiple experiments? How many experiments are needed then?

Exercise 19 (Computational time as a function of the simulation horizon T). Increase the number of samples T_s (using the minimum number of data samples T_d, see Exercise 16) and observe how the computational time grows. Compare the result with the one of model-based methods.

Exercise 20 (Nonzero initial conditions). Explain how to deal with nonzero initial conditions. Compare the model-based (`lsim`) and the data-driven (`ddint`) simulation methods in case of nonzero initial conditions.

Exercise 21 (Autonomous systems). Compare the model-based and data-driven simulation methods in case of autonomous systems.

Exercise 22 (Recursive data-driven simulation). Experiment with the option of `ddint` to do recursive computation in blocks of L samples, where $\ell < L \leq T$. Show that the recursive computation allows to reduce the data limit (6). What is the minimum number of data samples T_d needed in case of recursive computation of $\widehat{w}$?

Exercise 14: Impulse response simulation

For impulse response simulation, the only modification of the example code needed is to define `us` as a unit pulse:

```
us = [1; zeros(Ts-1, 1)]; ys = lsim(B, us);
```

The result obtained by completion of the missing data with `ddint`

```
w = [zeros(n, m + p); [us NaN(Ts, p)]];
tic, wh = ddint(wd, w); toc                % -> 0.01
```

then is exact up to the numerical precision

```
ysh = wh(n+1:end, m+1:end); norm(ys - ysh) % -> 1e-15
```

The alternative direct data-driven method `impulseest` does not yield the correct result:

```
tic, ysh2 = impulseest(iddata(yd, ud)); toc % -> 0.34
ysh2 = ysh2.num(1:Ts)';
norm(ys - ysh2) % -> non zero
```

The nonzero error of `impulseest` is due to the finite impulse response (FIR) representation used by the method. The rational for using an FIR representation is that, in theory, a stable linear time-invariant system can be approximated arbitrarily well by an FIR system at the price of increasing the length of the impulse response. Therefore, the error incurred by the FIR representation, which depends on the length of the impulse response, can be controlled. The length of the FIR representation is determined automatically by `impulseest`, presumably for minimization of the error. The finite length of the data w_d, however, limits the number of impulse response coefficients that can be estimated from data. Thus, in practice, the finiteness of the data imposes a limit on the accuracy achievable by the FIR representation. The issue of finite data becomes more pronounced when the system is lightly damped. In the extreme of marginally stable as well as for unstable systems, the FIR approximation fails to capture the system dynamics (the error is infinite).

The data-driven representation, used by `ddint`, does not suffer from the limitation of the FIR representation, it can be exact even for a finite w_d and "works" for unstable systems (see Section **??**). Thus, as observed empirically, it allows exact computation of a finite number of samples of the impulse response, using a finite amount of data w_d. The key question becomes:

> What is the minimum amount of data needed?

Exercises 16 and 22 investigate this question.

Exercise 15: Step response simulation

The modification needed for simulation of the step response is again trivial:

```
us = ones(Ts, 1); ys = lsim(B, us); ws = [us ys];
```

The result obtained by completion of the missing data with `ddint`

```
w = [zeros(n, m + p); [us NaN(Ts, p)]]; wh = ddint(wd, w);
```

is exact up to the numerical precision

```
ysh = wh(n+1:end, m+1:end); norm(ys - ysh) % -> 1e-15
```

Exercise 16: Minimum number of data samples T_{d}

In order to verify empirically that (6) is the minimal number of samples needed for data-driven simulation, first we check that the result obtained by `ddint` is correct for $T_{\mathrm{d}} = T_{\mathrm{min}}$:

```
<<define-Tmin>> % Tmin = (m + 1) * T + n - 1;
T = size(w, 1);
wh = ddint(wd(1:Tmin(T), :), w); norm(ws - wh(n+1:end, :)) % -> 0
```

Then, we check that the result obtained by `ddint` is incorrect for $T_{\mathrm{d}} = T_{\mathrm{min}} - 1$:

```
wh = ddint(wd(1:Tmin(T) - 1, :), w); norm(ws - wh(n+1:end, :)) % -> not zero
```

Therefore, with less that T_{min} data points `ddint` can not simulate the response.

Exercise 17: Multivariable systems

Methods based on parametric representations are often derived first in the single-input single-output case. Depending on the method, subsequent generalizations to multivariable systems may be cumbersome, nontrivial, or even impossible. An advantage of `ddint` is that it naturally applies to general multivariable systems. The simulation example below demonstrates this.

We take random stable `n = 5`-th order system with `m = 3` inputs and `p = 2` outputs:

```
n = 5; m = 3; p = 2; B = drss(n, p, m);
```

The to-be-simulated trajectory is a random zero initial conditions trajectory with `Ts = 10` samples:

```
Ts = 10; us = rand(Ts, m); xs_ini = zeros(n, 1);
ys = lsim(B, us, [], xs_ini); ws = [us ys];
```

The data trajectory w_{d} is also randomly chosen (observing the fact that it should satisfy (6))

```
Td = 1000; ud = rand(Td, m); xd_ini = rand(n, 1);
yd = lsim(B, ud, [], xd_ini); wd = [ud yd];
```

The code for the computation of the response itself is identical to the code in the single-input single-output case:

```
w = [zeros(n, m + p); [us NaN(Ts, p)]];
wh = ddint(wd, w); ysh = wh(n+1:end, m+1:end);
```

Finally, we verify that the result is correct

```
norm(ys - ysh) % -> 1e-15
```

Exercise 18: Multiple data trajectories $w_{\mathbf{d}}^1, \ldots, w_{\mathbf{d}}^N$

Using multiple trajectories, the length of the trajectories can be reduced to $T_{\mathrm{d}} = T$, where T is the simulation time (including the specification of the initial conditions). As shown in Exercise 20, the shortest length of the past block of w that is sufficient to specify the initial conditions is the lag of the system ℓ. Thus, T_{d} can be reduced to as few as $T_{\mathrm{s}} + \ell$ samples. With $T_{\mathrm{d}} = T$, the minimum number of trajectories needed is $N = mT + n$.

```
T = size(w, 1); N = m * T + n; Td = T; clear wd
for i = 1:N
  ud = rand(Td, m); xd_ini = rand(n, 1);
  yd = lsim(B, ud, [], xd_ini); wd{i} = [ud yd];
end
```

We call `ddint` with the cell-array `wd` as "data" and `w`, defined for solving the simulation problem

```
wh = ddint(wd, w); ysh = wh(n+1:end, m+1:end);
```

Finally, we verify that the result `ysh` is correct:

```
norm(ys - ysh) % -> 1e-15
```

Note 3 ($T_{\mathrm{d}} = \ell + 1$ and $N = m(\ell+1) + n$ via recursive computation). Using `ddint`'s option for recursive computation of the response, T_{d} can be reduced to $\ell + 1$ and N to $m(\ell+1) + n$ (see Exercise 22). It can be shown that this is the fundamental lower limit for any data-driven simulation method.

Exercise 19: Computational time as a function of the simulation horizon T

The data generating system is random stable single-input, `p = 100` output, of order `n = 1000`

```
n = 1000; m = 1; p = 100; B = drss(n, p, m); ell = ceil(n / p);
```

The high order and large number of outputs are chosen in order to make the computation time for the model-based simulation method `lsim` non-negligibly small. The maximum simulation horizon is `Ts = 1500`

```
Ts = 1500; us = rand(Ts, m); xs_ini = zeros(n, 1);
ys = lsim(B, us, [], xs_ini); ws = [us ys];
```

The data trajectory w_d is long enough for the maximum `Ts`

```
Td = 5000; ud = rand(Td, m); xd_ini = rand(n, 1);
yd = lsim(B, ud, [], xd_ini); wd = [ud yd];
```

We do the data-driven simulation for increasing length of the simulation horizon providing as data w_d to `ddint` the minimal number of samples (6):

```
<<define-Tmin>> % Tmin = @(T) (m + 1) * T + n - 1;
np = 10; T = round(linspace(100, Ts, np));
for i = 1:np
  w = [zeros(ell, m + p); [us(1:T(i), :) NaN(T(i), p)]];
  tic, wh = ddint(wd(1:Tmin(size(w, 1)), :), w); t(i) = toc;
  tic, ys_ = lsim(B, us(1:T(i))); t_(i) = toc;
end
```

Figure 2 shows the computation time of `ddint` and `lsim` as a function of the simulation horizon for

```
plot(T, t, '-'), axis([T(1) T(end) 0 max(t)]), box off
title('ddint'), xlabel('simulation time'), ylabel('computation time')
plot(T, t_, '-'), axis([T(1) T(end) 0 max(t_)]), box off
title('lsim'), xlabel('simulation time'), ylabel('computation time')
```

The result indicates that the computation time of `ddint` grows quadratically as a function of =T= while for the model-based simulation methods, implemented in `lsim`, the time grows linearly.

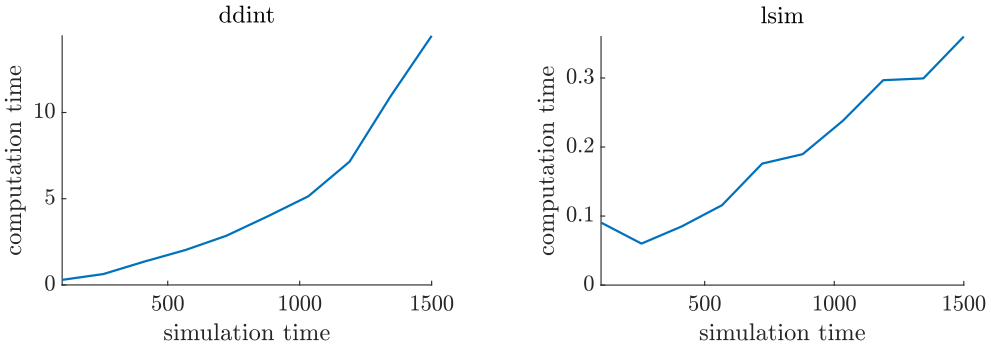

Figure 2: Left plot: The computation time needed for `ddint` to simulate a response of length T is quadratic in T. Right plot: The computation time needed for `lsim` to simulate a response of length T is linear in T.

Exercise 20: Nonzero initial conditions

Initial conditions in the data-driven setting are specified by a sufficiently long past trajectory w_ini, such that the concatenation $w_\mathrm{ini} \wedge w_\mathrm{s}$ of w_ini and w_s is a valid trajectory. In the single-output case, "sufficiently long" means at least n-sample, where n is the order of the system. Zero initial conditions are specified by setting $w_\mathrm{ini} = 0$.

The key question for setting nonzero initial conditions is how to choose w_ini. One option is to simulate $w_\mathrm{ini} \wedge w_\mathrm{s}$

```
u = rand(Ts + n, m); x_ini = rand(n, 1);
y = lsim(B, u, [], x_ini); w = [u y];
wini = w(1:n, :); ws = w(n+1:end, :);
us = ws(:, 1:m); ys = ws(:, m+1:end);
```

The question of fining w_{ini} when w_{s} is given, however, leads to an important problem on its own. In the classical setting, finding the initial conditions corresponding to a given trajectory w_{s} is the initial state estimation problem that is solved by an *observer design*. In the data-driven setting the observer problem becomes: given w_{d} and w_{s}, find w_{ini}. This problem can also be solved with `ddint`:

```
wh = ddint(wd, [NaN(n, m + p); ws]); wini_ = wh(1:n, :);
```

Note 4 (Nonuniqueness of w_{ini}). `wini_` need not be equal to `wini`. In general, there are infinitely many trajectories w_{ini} of length ℓ that can be concatenated with w_{s} resulting in a valid trajectory of $\mathscr{B}$ (see Exercise **??** where all of them are compared).

Next, we demonstrate that using both `wini` and `wini_` as initial conditions for the data-driven simulation

```
wh  = ddint(wd, [wini;  [us NaN(Ts, p)]]); ysh  = wh(n+1:end, m+1:end);
wh_ = ddint(wd, [wini_; [us NaN(Ts, p)]]); ysh_ = wh(n+1:end, m+1:end);
```

yields the correct result

```
norm(ys - ysh)  % -> 1e-15
norm(ys - ysh_) % -> 1e-15
```

Exercise 21: Autonomous systems

Since the data-driven simulation method based on `ddint` is applicable to general linear time-invariant systems under nonzero initial conditions, it can be used for simulation of autonomous systems. The only modification of the code given earlier is that now we set `m = 0`.

```
n = 5; p = 1; m = 0; B = drss(n, p, m);
Ts = 10; us = rand(Ts, m); xs_ini = rand(n, 1);
ys = lsim(B, us, [], xs_ini); ws = [us ys];

Td = 100; ud = rand(Td, m); xd_ini = rand(n, 1);
yd = lsim(B, ud, [], xd_ini); wd = [ud yd];

wh = ddint(wd, [NaN(n, m + p); ws]); wini = wh(1:n, :);
wh = ddint(wd, [wini; [us NaN(Ts, p)]]);
ysh = wh(n+1:end, m+1:end); norm(ys - ysh) % -> 1e-10
```

Exercise 22: Recursive data-driven simulation

By choosing $L = 1$ (its minimum value for the block computation), we need $T_{\text{d}} \geq (m+1)\ell + m + n$ data samples. Next, this is verified empirically.

```
n = 5; p = 1; m = 1; B = drss(n, p, m); ell = round(n / p);
Ts = 10; us = rand(Ts, m); xs_ini = zeros(n, 1);
ys = lsim(B, us, [], xs_ini); ws = [us ys];

Td = (m+1) * ell + m + n; % = Tmin(ell + 1)
ud = rand(Td, m); xd_ini = rand(n, 1);
yd = lsim(B, ud, [], xd_ini); wd = [ud yd];

w = [zeros(ell, m+p); [us NaN(Ts, p)]];
wh = ddint(wd, w, [], [], [], [], 1, ell);
ysh = wh(ell+1:end, m+1:end); norm(ys - ysh) % -> 1e-15
```

Or using $N = m(\ell+1)+n$ trajectories, their length can be reduced to $T_\mathrm{d} = \ell+1$

```
%% Recursive simulation (using multiple trajectories)
Td = ell + 1; N = m * (ell + 1) + n;
for i = 1:N
  ud = rand(Td, m); xd_ini = rand(n, 1);
  yd = lsim(B, ud, [], xd_ini); wd{i} = [ud yd];
end

w = [zeros(ell, m+p); [us NaN(Ts, p)]];
wh = ddint(wd, w, [], [], [], [], 1, ell);
ysh = wh(ell+1:end, m+1:end); norm(ys - ysh) % -> 1e-15
```

Summary and discussion

This section shows that `ddint` is applicable to the simulation problem and results in a general and practical data-driven simulation method. Exercises 14, 15, and 21 demonstrate the data-driven simulation method based on `ddint` in the special cases of impulse response computation, step response computation, and simulation of an autonomous system. For impulse response computation of a finite impulse response system, the System Identification Toolbox of Matlab provides an alternative direct data-driven method `impulseest`. For all other cases, the available data-driven simulation methods are indirect, *i.e.*, they involve an explicit model identification. Exercise 17 demonstrates that `ddint` works correctly in case of general multivariable systems. Exercise 20 shows how initial conditions can be obtained also by using `ddint` for solving a data-driven version of the classical observer problem. Exercise 18 demonstrates the possibility of using data from multiple experiments. Exercise 16 verifies formula (6) for the minimal amount of data samples T_d and Exercise 19 shows empirically the computation time of `ddint` as a function of the simulation horizon T. In comparison with model-based methods `ddint` is less efficient, however, on a modern computer the difference is noticeable only at relatively large simulation horizons.

The idea of data-driven interpolation and approximation as well as the method of [11] are inspired by [13], where data-driven methods for simulation and control are presented. Apart from generalization of the problem, an important new development compared to [13] is the use of low-rank approximation [8] and regularization for dealing with inexact data [2]. These developments allowed practical application of the methods.

5 Missing data estimation

Simulation is a special interpolation problem. In this section, we consider two other interpolation problems: input estimation and estimation of periodically missing values. The latter example demonstrates the full flexibility of `ddint` to treat missing values among inputs as well as outputs at arbitrary moments of time.

Input estimation

Consider a system $\mathscr{B}$ with two inputs—u_1 and u_2—and one output:

```
n = 5; p = 2; m = 1;
<<random-data-generating-system-B>>
```

implicitly specified by a trajectory w_d

```
Td = 100;
<<random-data-trajectory-wd>>
```

The goal is to estimate u_1, given u_2 and y.

```
T = 20; u0 = rand(T, m); x_ini = zeros(n, 1);
y0 = lsim(B, u0, [], x_ini); w0 = [u0 y0];
```

In order to solve the data-driven input estimation with `ddint`, we pose it as a missing data estimation problem

```
w  = [NaN(T, 1) u0(:, 2:end) y0];
wh = ddint(wd, w); u1h = wh(:, 1);
```

Finally, we verify that the estimate `u1h` matches the true input u_1

```
norm(u0(:, 1) - u1h) % -> 1e-14
```

Periodic missing values

The given values `w` are sampled from `w0` with a period of $n+1$ samples:

```
w = w0; w(3:n+1:end, :) = NaN;
```

The interpolated trajectory by `ddint`

```
wh = ddint(wd, w);
```

matches exactly the true trajectory `ws`

```
norm(w0 - wh) % -> 1e-12
```

A model-based method for missing data estimation is implemented in the function `misdata` of the System Identification toolbox of Matlab. Applying it to the data in the example

```
wh_mb = misdata(iddata(w(:, 2:3), w(:, 1)), idss(B));
wh_mb = [wh_mb.InputData wh_mb.OutputData];
norm(w0 - wh_mb) % -> 1e-15
```

it also recovers exactly the missing data.

Exact recovery of the missing data, however, is not guaranteed. General necessary and sufficient conditions are not available. Necessary conditions and special cases are explored in the exercises.

Exercises

Exercise 23 (Necessary condition for uniqueness). Argue that a necessary condition for unique solution of the interpolation problem is that at least $mT+n$ elements in w are given. Is it also sufficient?

Exercise 24 (Condition for uniqueness of the input estimation problem). When does the input estimation problem have a unique solution? Give separately the cases of known initial conditions and unknown initial conditions. Start with the single-input single-output case.

The following problems explore the impact of the missing elements location on the uniqueness of the completion.

Exercise 25 (Smallest number of given elements in w guaranteeing unique interpolation). Give examples of interpolation problems with as few given elements in w as possible that result in unique completion. Consider separately the cases of scalar autonomous system, single-input single-output system, and general multivariable system.

Exercise 26 (Smallest number of missing elements in w leading to nonuniqueness). Give examples of interpolation problems with as few missing values as possible that result in nonunique completion. Consider separately the cases of scalar autonomous system, single-input single-output system, and general multivariable system.

Solutions SOL

Exercise 23: Necessary condition for uniqueness

For well-posedness of the interpolation problem in w at least $mT_\mathrm{s}+n$ elements should be given. This condition is necessary but not sufficient. Finding general necessary and sufficient conditions is an open problem.

Exercise 24: Condition for uniqueness of the input estimation problem

The to-be-estimated elements are the initial conditions and the input. This makes $n+mT_\mathrm{s}$ unknowns. The given elements are the outputs. This makes pT_s equations. Necessary condition for uniqueness are then

$$p > m \quad \text{and} \quad T_\mathrm{s} > \frac{n}{p-m}.$$

Exercise 25: Smallest number of given elements in w guaranteeing unique interpolation

In general, for unique completion at least $mT_s + n$ elements have to be given. Their distribution however also matters. Specialized for the scalar autonomous case, the minimal number of given elements is n. When the n given elements are the first samples, they specify the initial conditions and therefore guarantee uniqueness of the completion.

In the single-input single-output case, the minimal number of given elements is $T_s + n$. By choosing them to be the T_s input samples and the first n samples of the output, the completion is guaranteed to be unique because the interpolation problem becomes simulation.

In the general case, the minimal number of given elements is $mT_s + n$. Let's choose mT_s of the given elements as the input samples. The remaining n given elements should be chosen so that they specify the initial conditions, so that they should be chosen among the first ℓ samples of the output, where ℓ is the lag of the system. For a general multivariable system this task is nontrivial. However, when $n = p\ell$ the choice is evident—we can choose all output elements in the first ℓ samples. The special case $n = p\ell$ is generic when the system is randomly chosen.

Exercise 26: Smallest number of missing elements in w leading to nonuniqueness

In the scalar autonomous case, $T_s - n + 1$ missing elements (*i.e.*, less than n given elements) guarantee nonuniqueness. In the case of open systems (systems with inputs), the easiest way to break the uniqueness of the completion is to have at a ℓ (lag of the system) consecutive samples completely missing, *i.e.*, $q\ell$ elements missing with the specific distribution described above results in nonunique completion.

Summary and discussion

This section demonstrates the flexibility of `ddint` to estimate missing elements in arbitrary configuration of input as well as output variables. This is another viewpoint of the trajectory interpolation aspect of the basic problem (7) that `ddint` solves. The key question in missing data estimation is "under what conditions the completion is unique?" General necessary and sufficient conditions for uniqueness of the interpolant is an open problem. The exercises explored some special cases thus developing intuition about the general question.

Dealing with missing values in the data trajectory w_d (as well as in the to-be-interpolated trajectory w) makes the problem much harder. A subspace-type method is proposed in [9]. Other methods are based on local optimization [14] and convex relaxations [3].

6 Errors-in-variables smoothing

In the examples reviewed so far the given samples are exact and the goal is interpolation, or, equivalently, missing values recovery. Existence of an exact completion of the missing values is guaranteed by the exactness of the data. Moreover, uniqueness of the completion guarantees correct recovery of the missing values, so it is the critical property.

The new aspect in the following examples is that w is inexact. This can be due to

1. *errors-in-variables setup*—additive measurement noise on w,

2. *ARMAX setup*—unobserved disturbances acting on the system and measurement noise on the output y,

3. *nonlinear and/or time-varying dynamics*—the data-generating system is not linear time-invariant,

or a combination of the three. The data trajectory w_d however is still exact. (Inexact w_d is considered in the mini-projects.)

In case of inexact data w, exact solution of (7) generically does not exist. The problem involves then an approximation. In case of additive measurement noise (errors-in-variables setup) and disturbances (ARMAX setup), which are modeled as stochastic processes, the goal is to obtain the maximum-likelihood estimator of the true value of w.

Model-based solution

The errors-in-variables smoothing problem is

$$\text{minimize} \quad \text{over } \widehat{w} \quad \|w-\widehat{w}\|_S \quad \text{subject to} \quad \widehat{w} \in \mathscr{B}|_T.$$

Using a state-space representation of the system, it can be rewritten as

$$\text{minimize} \quad \text{over } \widehat{w}, \widehat{u}, \text{ and } x_0 \quad \|w-\widehat{w}\|_S \quad \text{subject to} \quad \widehat{w} = M_T(A,B,C,D,\Pi) \begin{bmatrix} x_0 \\ \widehat{u} \end{bmatrix},$$

where $M_T(A,B,C,D,\Pi)$ is defined in (1).

```
function wh = eiv_smoothing(B, w) % SISO case only
[T, q] = size(w); M = basis_BT_mb(B, T);
wh = reshape(M * pinv(M) * w(:), T, q);
```

Data-driven solution

Next, we consider optimal smoothing in the errors-in-variables setup. As before, the data-generating system

```
n = 5; p = 1; m = 1;
<<random-data-generating-system-B>>
```

is represented by the data trajectory

```
Td = 100;
<<random-data-trajectory-wd>>
```

The to-be-smoothed trajectory `w` is generated in the errors-in-variables setup:

```
T = 10; s = 0.1;
u0 = rand(T, m); x0_ini = zeros(n, 1);
y0 = lsim(B, u0, [], x0_ini); w0 = [u0 y0];
wt = randn(T, m+p); w = w0 + s * norm(w0) * wt / norm(wt);
```

Calling `ddint` with the input data `wd` and `w`

```
wh = ddint(wd, w);
```

results in the least-squares approximation of `w` by a trajectory `wh` of the system defined by =wd=. The least-squares approximation $\widehat{w}$ is the maximum-likelihood estimator of the true value `w0` of w.

In order to verify the result of `ddint`, we compare it with the one of the model-based function `eiv_smoothing`

```
norm(wh - eiv_smoothing(B, w)) % -> 1e-14
```

Exercises

Exercise 27 (Monte-Carlo simulation). In order to reduce the randomness of the result, average the relative approximation errors over $N = 100$ noise realizations.

Exercise 28 (Approximation error as a function of the noise level). Plot the average relative approximation error as a function of the noise level for noise in the interval [0%, 50%]. Is the result what you expected? Can you explain it?

Exercise 29 (Smoothing with exactly known initial conditions). Do the errors-in-variables smoothing experiment with known zero initial conditions. Compare the accuracy of the solutions (with respect to the true value) with and without using the prior knowledge about the initial conditions.

Exercise 30 (Errors-in-variables smoothing with exact interpolation points). Redo the errors-in-variables smoothing experiment with unknown initial conditions for an autonomous system with $1,\ldots,n-1$ interpolation points that are randomly chosen and for which the given data points are noise free. Verify that the approximation error with respect to the true trajectory is decreasing as more exact interpolation points are added.

Exercise 27: Monte-Carlo simulation

In order to reduce the randomness of the result due to the randomness of the noise, we repeat the experiment and look at the average performance. The procedure is known as Monte-Carlo simulation. We set `N=100` repetitions with noise level `s = 0.1`

```
error = @(wh) norm(ws - wh) / norm(ws);
N = 100; s = 0.1; j = 1;
<<run-monte-carlo>>

%% <run-monte-carlo>
for i = 1:N
  wt = randn(Ts, m+p); w = ws + s * norm(ws) * wt / norm(wt);
  wh = ddint(wd, w); e(i, j) = error(wh);
end
```

The result is

```
[error(w)  mean(e)] % -> 0.1000    0.0887
```

Exercise 28: Approximation error as a function of the noise level

Empirical observations suggest that up to a threshold noise level the method works "well" and after the threshold, the performance of the method sharply deteriorates or becomes erratic. By "works well", we mean that the performance is acceptable, *i.e.*, made relative and is expressed in percents it is less than say 20%. Also, the behaviour of the error as a function of the noise level may gradually grow but is not erratic.

```
np = 10; s = linspace(0, 0.5, np);
for j = 1:np
  s = s(j);
  <<run-monte-carlo>>
end
e = mean(e); plot(s, e), box off, axis([s(1) 1.1 * s(end) 0 max(e)])
xlabel('noise level'), ylabel('relative estimation error')
```

The result is shown in Figure 3.

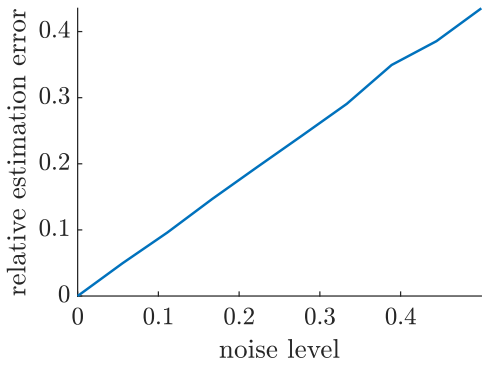

Figure 3: The approximation error grows linearly as a function of the noise level.

Exercise 32: Smoothing with exactly known initial conditions

Equality constraints are needed in the context of mixed interpolation/approximation problems in order to specify the interpolation points. For example in the errors-in-variables smoothing example imposing the prior knowledge that the to-be-interpolated trajectory `ws` is generated from zero initial conditions, can be taken into account by appending zero samples in the past

```
wt = randn(Ts, m+p); w = ws + 0.1 * norm(ws) * wt / norm(wt);
wext = [zeros(n, 2); w];
```

The past samples are known exactly, so that they are specified as exact interpolation points by infinite weights:

```
S = [inf(n, 2); ones(Ts, 2)];
```

The smoothing with known initial conditions is then done by calling `ddint` with the extended trajectory and weights

```
whext = ddint(wd, wext, S);
wh2 = whext(n+1:end, :);
```

We verify that the estimate taking into account the known initial conditions has lower relative approximation error with respect to the true trajectory than the estimate that does not use the prior knowledge about the initial conditions:

```
[norm(ws - wh) norm(ws - wh2)] / norm(ws) % -> 0.0985    0.0673
```

Exercise 30: Errors-in-variables smoothing with exact interpolation points

This exercise demonstrates the option of `ddint` to impose exact interpolation points when solving an approximation problem. Classical methods for incorporating the prior knowledge of exact elements do not have this flexibility although it is possible to develop specialized methods for this modification of the problem. The key point is that for this modification by using the data-driven approach a new method is not needed.

```
wext = [zeros(n, 2); ws];
S = [inf(n, 2); ones(Ts, 2)];

whext = ddint(wd, wext, S); wh = whext(n+1:end, :);

error = @(wh) norm(ws - wh) / norm(ws);
```

Summary and discussion

Errors-in-variables smoothing can be viewed alternatively as computing the distance, called *misfit*, of w to $\mathscr{B}$

$$\text{misfit}(w,\mathscr{B}) := \min_{\widehat{w}} \|w-\widehat{w}\|_S \quad \text{subject to} \quad \widehat{w} \in \mathscr{B}|_T.$$

The misfit quantifies the lack of fit between w and $\mathscr{B}$. It is the size (measured in the $\|\cdot\|_S$-norm) of the smallest perturbation on w that renders w an exact trajectory of $\mathscr{B}$. A recursive algorithm for model-based errors-in-variables smoothing is given in [10, Section 3].

The opposite of missing data is exactly known data. Exactly known data is imposed by equality constraints. They are the strongest type of prior knowledge about the data—complete confidence in the given values. Exercises 32 and 30 demonstrates how adding interpolation points in the smoothing problem (*i.e.*, adding prior knowledge) improves the estimation accuracy. The next section imposes the priori knowledge that the input is known exactly. The result is the ordinary Kalman smoother.

7 Kalman smoothing

In this section, we consider optimal smoothing in the ARMAX setup. The problem is known as Kalman smoothing because it was first solved by the celebrated Kalman smoother.

The Kalman smoothing problem differs from the errors-in-variables smoothing problem in to ways:

1. the input u is exactly known while the output y is still observed with additive noise, and
2. an unobserved input e, called *disturbance*, that is a zero-mean white Gaussian stochastic process acts on the system.

Let $\mathscr{B}_{\text{ext}}$ be the behavior of the extended variable $w_{\text{ext}} := \left[\begin{smallmatrix} w \\ e \end{smallmatrix}\right]$.

The disturbance e is not observed, however, the assumption that it is a zero-mean white Gaussian stochastic process makes it different from the missing input e considered before. The estimate $\widehat{e}$ of a missing input does not appear in the cost function and is determined only from the interpolation condition $(w,\widehat{e}) \in \mathscr{B}_{\text{ext}}$, while the estimate $\widehat{e}$ of a stochastic input appears in the cost function with the term $\|\widehat{e}\|_{S_e}$ and is estimated from the prior knowledge that it is as small as possible in the $\|\cdot\|_{S_e}$-norm. More pragmatically, using `ddint`, a missing input is specified by `ed = NaN`, while a stochastic input is specified by `ed = 0` and the block S_e of the weight matrix S that corresponds to e.

In order to illustrate the application of `ddint` to Kalman smoothing, first we choose an extended data-generating system $\mathscr{B}_{\text{ext}}$ with variables $w_{\text{ext}} = (u,y,e)$

```
n = 5; p = 1; mu = 1; me = 1; m = mu + me; sy = 0.1; se = 0.1;
Bext = drss(n, p, m); % Bext = Bext * diag(1 ./ dcgain(Bext));
```

and generate a data trajectory w_{d}

```
Td = 100;
ud = rand(Td, mu); ed = se * randn(Td, me); xd_ini = rand(n, 1);
yd = lsim(Bext, [ud ed], [], xd_ini); wextd = [ud yd ed];
```

and a to-be-smoothed trajectory w

```
T = 10; u = rand(T, mu); e = randn(T, me); x_ini = rand(n, 1);
y0 = lsim(Bext, [u e], [], x_ini); yt = randn(T, p);
y = y0 + sy * norm(y0) * yt / norm(yt);
```

The Kalman smoothing problem is specified as a data-driven interpolation/approximation problem (7) by setting

```
wext = [u y zeros(T, me)];
S = [inf(T, mu), (1 / norm(y - y0)) * ones(T, p), (1 / norm(e)) * ones(T, me)];
```

The smoothed output `yh` is obtained then by

```
wexth = ddint(wextd, wext, S); yh = wexth(:, mu+1:mu+p);
```

In order to verify the result, we use the relative estimation error

```
error = @(yh) norm(y0 - yh) / norm(y0);
```

and compare the estimate `yh` with the estimate obtained by a model-based method implemented in the function `predict` from the System Identification toolbox of Matlab

```
yh_mb = predict(idss(Bext(1,2)), iddata(y, u), T); yh_mb = yh_mb.OutputData;
```

The results are

```
[error(yh) error(yh_mb)] % -> 0.0956    0.3544
```

Exercises

Exercise 31 (Kalman smoothing without disturbance). Revise the solution of the Kalman smoothing problem for the case when w is not exact due to measurement noise on the output only, *i.e.*, when there is no disturbance signal.

Exercise 32 (Kalman smoothing with exactly known initial conditions). In the setup of Exercise 31 (measurement noise without disturbance), add the prior knowledge that the initial conditions are given and are exact. (You can assume zero initial conditions.)

Exercise 33 (Kalman smoothing without measurement noise). Revise the solution of the Kalman smoothing problem for the case when there is no measurement noise on the output.

Exercise 31: Kalman smoothing without disturbance

```
n = 5; p = 1; m = 1; sy = 0.1; B = drss(n, p, m);

Td = 100; ud = rand(Td, m); xd_ini = rand(n, 1);
yd = lsim(B, ud, [], xd_ini); wd = [ud yd];

T = 10; u = rand(T, m); x_ini = zeros(n, 1);
y0 = lsim(B, u, [], x_ini); yt = randn(T, p);
y = y0 + sy * norm(y0) * yt / norm(yt); w = [u y];
S = [inf(T, m), ones(T, p)];

wh = ddint(wd, w, S); yh = wh(:, m+1:end);
norm(y0 - yh) / norm(y0) % -> 0.0606
```

Exercise 32: Kalman smoothing with exactly known initial conditions

```
wh = ddint(wd, [zeros(n, m+p); w], [inf(n, m+p); S]);
yh = wh(n+1:end, m+1:end);
norm(y0 - yh) / norm(y0) % -> 1e-15
```

Exercise 33: Kalman smoothing without measurement noise

Since without measurement noise the given output y is exact, the original Kalman smoothing problem which aims at noise removal does not make sense. We can rephrase the problem however as estimation of the disturbance e or, equivalently, removal of the effect of the disturbance on y.

Consider first estimation of e

```
n = 5; p = 1; mu = 1; me = 1; m = mu + me; sy = 0.0;
Bext = drss(n, p, m);

Td = 100; ud = rand(Td, mu); ed = randn(Td, me); xd_ini = zeros(n, 1);
yd = lsim(Bext, [ud ed], [], xd_ini);
wextd = [ud yd ed];

T = 10; u = rand(T, mu); e = randn(T, me); x_ini = zeros(n, 1);
y0 = lsim(Bext, [u e], [], x_ini); yt = randn(T, p);
y = y0 + sy * norm(y0) * yt / norm(yt);

wext = [u y zeros(T, me)];
S = [inf(T, mu), (1 / norm(y - y0)) * ones(T, p), (1 / norm(e)) * ones(T, me)];

wexth = ddint(wextd, wext, S); eh = wexth(:, mu+p+1:end);
norm(e - eh) / norm(e) % -> 0.5882
```

Consider now removal of the effect of the disturbance on y.

```
%% Kalman smoothing without measurement noise (another approach)
y0d = lsim(Bext, [ud zeros(Td, me)]); % assuming zero initial conditions
dd  = lsim(Bext, [zeros(Td, mu) ed]); % norm(yd - (y0d + dd))
wextd = [ud y0d ed yd];

y0 = lsim(Bext, [u, zeros(T, me)]);
d  = lsim(Bext, [zeros(T, mu) e]);  % norm(y - (y0 + d))
wext = [u NaN(T, p) zeros(T, mu) y];
S = [inf(T, mu), zeros(T, p), ones(T, me), inf(T, p)];

wexth = ddint(wextd, wext, S);
```

```
y0h = wexth(:, mu+1:mu+p);
eh2 = wexth(:, mu+p+1:mu+p+me);
norm(y0 - y0h) / norm(y0) % -> check it!
norm(e - eh2) / norm(e)
```

Exercise 34: Mixed misfit–latency smoothing

```
n = 5; p = 1; mu = 1; me = 1; m = mu + me; s = 0.5;
Bext = drss(n, p, m);

Td = 100; ud = rand(Td, mu); ed = randn(Td, me); xd_ini = zeros(n, 1);
yd = lsim(Bext, [ud ed], [], xd_ini);
wextd = [ud yd ed];

T = 10; u0 = rand(T, mu); e = randn(T, me); x_ini = zeros(n, 1);
y0 = lsim(Bext, [u0 e], [], x_ini); w0 = [u0 y0];
wt = randn(T, mu+p); w = w0 + s * norm(w0) * wt / norm(wt);

wext = [w zeros(T, me)];
S = [(1 / norm(w - w0)) * ones(T, mu+p), (1 / norm(e)) * ones(T, me)];

wexth = ddint(wextd, wext, S);
wh = wexth(:, 1:mu+p); eh = wexth(:, mu+p+1:end);
[norm(w0 - w) norm(w0 - wh)] / norm(w0) % -> 0.1 0.099
```

Special cases:

- pure misfit (EIV smoothing)
- output error
- Kalman smoothing (ARMAX smoothing)
- pure latency
- exactly known disturbance
- completely unknown disturbance (input estimation)

Summary and discussion

In the errors-in-variables setup the lack of fit between w and $\mathscr{B}$ is due to the measurement errors. The appropriate estimation criterion in this case is the misfit. When the lack of fit between w and $\mathscr{B}$ is due to disturbance and the measurement error on the output, the appropriate estimation criterion is the *latency*, defined as

$$\text{latency}(w, \mathscr{B}_{\text{ext}}) := \min_{(w,\widehat{e}) \in \mathscr{B}_{\text{ext}}|_T} \|\widehat{e}\|.$$

The latency computation corresponds to the Kalman smoothing problem without measurement noise (Exercise 33).

In [7] a combined misfit–latency approach for system identification is proposed. The misfit–latency approach has as special cases the errors-in-variables and ARMAX setups and allows one to consider the most general problem where the uncertainty is due to both disturbance as well as measurement errors on both inputs and outputs. Data-driven smoothing in the combined misfit–latency setting can be done with `ddint`.

Exercise 34 (Mixed misfit–latency smoothing). Using `ddint`, solve the data-driven smoothing problem

$$\text{minimize} \quad \text{over } \widehat{w} \text{ and } \widehat{e} \quad \left\| \begin{bmatrix} w - \widehat{w} \\ \widehat{e} \end{bmatrix} \right\|_S \quad \text{subject to} \quad \begin{bmatrix} \widehat{w} \\ \widehat{e} \end{bmatrix} \in \mathscr{B}_{\text{ext}}|_T,$$

where the system $\mathscr{B}_{\text{ext}}$ is implicitly specified by a trajectory $w_{\text{d,ext}} \in \mathscr{B}_{\text{ext}}|_{T_{\text{d}}}$. Apply the solution on an example.

8 Data-driven control

The control problems considered are open-loop. The design specifications in the first two problems are constraints on the trajectory, while the design specification of the third problem is an optimality criterion.

State transition

A basic control problem is state transition: transition from a given "past" trajectory $w_p \in \mathscr{B}|_{T_p}$ to a given "future" trajectory $w_f \in \mathscr{B}|_{T_f}$ via a "control" trajectory $w_c \in \mathscr{B}|_{T_c}$, *i.e.*, find w_c, such that $w_p \wedge w_c \wedge w_f \in \mathscr{B}_{T_p+T_c+T_f}$, see Figure 4.

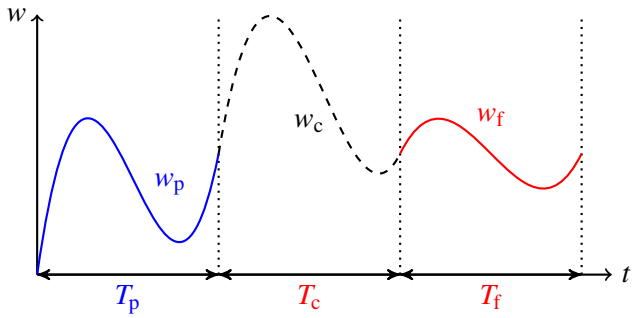

Figure 4: The state transition control problem aims to connect the given "past" w_p and "future" w_f trajectories via a "control" w_c trajectory.

In the simulation example we take a random stable system

```
n = 5; p = 1; m = 1; q = m + p; Tc = 10;
B = drss(n, p, m);
```

and a random data trajectory

```
Td = 100; ud = rand(Td, m); xd_ini = rand(n, 1);
yd = lsim(B, ud, [], xd_ini); wd = [ud yd];
```

The state transition problem is specified by the past and future trajectories

```
wp = rand(n, q); wf = zeros(n, q);
```

In order to cast the state transition problem as a missing data estimation problem, we define $w := w_p \wedge w_c \wedge w_f$ with w_p and w_f the given trajectories and w_c as missing/to-be-completed:

```
w = [wp; NaN(Tc, q); wf];
S = [inf(n, q); zeros(Tc, q); inf(n, q)];
[wh, N] = ddint(wd, w, S); wc = wh(n+1:n+Tc, :);
```

In the example there is a nonunique control trajectory w_c achieving the desired state transition. The set of possible control trajectories has dimension

```
size(N, 2) % -> 5
```

One way of utilizing the control freedom is by choosing the minimum energy control. The minimum energy control is computed by `ddint` as follows:

```
w = [wp; [zeros(Tc, m) NaN(Tc, p)]; wf];
S_me = [inf(n, q); [ones(Tc, m) zeros(Tc, p)]; inf(n, q)];
[wh_me, N_me] = ddint(wd, w, S_me); wc = wh(n+1:n+Tc, :);
```

Trajectory planning

The problem considered next is to find a trajectory that passes through given interpolation "points" $p_1, \ldots, p_K$ at given moments of time $t_1, \ldots, t_K$. By "points" we mean any subset of elements of the variables w (the whole w, the output y, or any subset of inputs and outputs). Moreover, the subset of variables need not be fixed for all moments of time.

Depending on the number of interpolation points and the number of variables involved, there may be infinitely many, one, or no feasible trajectory. This is explored in Exercise 35. In control problems, typically there are infinitely many feasible trajectories, in which case an optimal one is selected by specifying an optimality criterion.

In order to illustrate the application of `ddint` for trajectory planning, we choose a random stable system

```
n = 5; p = 2; m = 1; q = m + p; Tc = 10;
B = drss(n, p, m);
```

and a data trajectory

```
Td = 100; ud = rand(Td, m); xd_ini = rand(n, 1);
yd = lsim(B, ud, [], xd_ini); wd = [ud yd];
```

In addition to passing through the interpolation points, the desired trajectory is required to start and to end at the zero state, thus before adding the interpolation conditions the problem is defined by trajectory `w` and weight `S`

```
w = [zeros(n, q); NaN(Tc, q); zeros(n, q)];
S = [inf(n, q); zeros(Tc, q); inf(n, q)];
```

The interpolation points are $y(n+3) = \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ and $y(n+6) = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$. They are added to the problem specification as follows

```
Iy = (m+1):q
w(n + 3, Iy) = [1 0]; S(n + 3, Iy) = inf;
w(n + 6, Iy) = [0 1]; S(n + 6, Iy) = inf;
```

Now the set of all trajectories satisfying the constraints can be computed by evoking `ddint`

```
[wh, N] = ddint(wd, w, S);
```

It turns out that is in the particular example there is one dimensional solution set

```
size(N, 2) % -> 1
```

In order to find the minimum energy control that satisfies the constraints, we modify the problem as follows

```
w(n+1:n+Tc, 1:m) = 0; S(n+1:n+Tc, 1:m) = 1;
[wh_me, N_me] = ddint(wd, w, S);
```

Tracking control

The tracking control problem is mathematically equivalent to the errors-in-variables smoothing problem. The difference is that in the errors-in-variables smoothing problem `w` is a noise corrupted trajectory of the system, while in the tracking control problem `w` together with `S` specify the control objective and `w` need not be a trajectory of the system. In the example, we choose a random stable plant

```
n = 5; p = 1; m = 1; B = drss(n, p, m);
```

represented for the data-driven control by a trajectory

```
Td = 100; ud = rand(Td, m); xd_ini = rand(n, 1);
yd = lsim(B, ud, [], xd_ini); wd = [ud yd];
```

The to-be-tracked trajectory is zero input (minimum energy control) and constant output (step tracking)

```
Tc = 10; u = zeros(Tc, m); w = [u ones(Tc, p)];
```

The cost function minimized by the controller is

```
S = [ones(Tc, m) 100 * ones(Tc, p)];
e = @(wh) norm(S .* (w - wh), 'fro') / norm(w);
```

i.e., the control objective is minimum energy least-squares tracking of a constant unit output with time horizon $T_c = 10$.

Assuming zero initial conditions, the actual trajectory achieved applying the control input on the plant is

```
wha = @(wh) [wh(:, 1:m), lsim(B, wh(:, m+1:end))];
```

so that, the achieved performance is

```
ea  = @(wh) e(wha(wh));
```

Using `ddint` we solve the open-loop tracking control problem with zero initial conditions as follows:

```
wh = ddint(wd, [zeros(n, 2); w], [inf(n, 2); S]);
wh = wh(n+1:end, :);
```

The objective function and the actual achieved performance coincide:

```
[e(wh) ea(wh)] % -> 44.4719   44.4719
```

Exercises

Exercise 35 (Constraints and degrees of freedom). In the state transition and trajectory planning problems, what is the dimension of the set of solutions? How many interpolation constraints can be added in the trajectory planning problem retaining feasibility?

Exercise 36 (Uncertain initial conditions). Relax the prior knowledge of exactly known initial condition to uncertain initial conditions and observe the effect on the performance.

Exercise 37 (Unknown initial conditions). The opposite extreme of exactly known initial condition is unknown initial conditions. Compare the performance of the tracking control with exact knowledge of the initial condition with the one of unknown initial conditions.

Solutions SOL

Exercise 35: Constraints and degrees of freedom

The degrees of freedom in choosing the unconstrained control trajectory w_c are $mT_c + n$. With $T_p \geq \ell$, the specification of the past trajectory w_p has the effect of fixing the initial conditions, which leaves mT_c degrees of freedom in choosing $w_p \wedge w_c$. Then, the specification of the future trajectory w_f has the effect of fixing the terminal conditions, which requires n degrees of freedom. Thus, for $T_c \geq \ell$, the remaining degrees of freedom in choosing $w_p \wedge w_c$ is $mT_c - n$. Let's check this answer empirically.

```
ell = n; % assuming a SISO system
for Tc = ell:ell+5
  w = [wp; NaN(Tc, q); wf];
  S = [inf(n, q); zeros(Tc, q); inf(n, q)];
  [wh, N] = ddint(wd, w, S); wc = wh(n+1:n+Tc, :);
  [Tc, size(N, 2), m * Tc - n] % empirical and analytical # of DOF
end
```

In the trajectory planning problem the unconstrained trajectory has $mT_c + n$ degrees of freedom. Adding an interpolation point removes as many degrees of freedom as variables are specified. Thus, up to $mT_c + n$ variables can be interpolated for the whole horizon. In order to verify this empirically we generate random interpolation points with a different number of interpolation points.

```
Tc = 10; dof = m * Tc + n;
for nip = 1:dof
  w = NaN(Tc, q); S = zeros(Tc, q);;
  I_fixed = randperm(q * Tc, nip); w(I_fixed) = 0; S(I_fixed) = Inf;
  [wh, N] = ddint(wd, w, S);
  [nip, size(N, 2), dof - nip] % empirical and analytical # of DOF
end
```

Exercise 36: Uncertain initial conditions

Exercise 37: Unknown initial conditions

Summary and discussion

The controllability property defined in the behavioral setting is a necessary and sufficient condition for existence of solution of the state transition problem. The trajectory planning problem occurs in robotics applications [4]. Closed-loop data-driven control can be achieved by embedding the open-loop data-driven control methods in predictive control. This led to the data enabled predictive control (DeePC) method [1]. In the mini-projects we consider the case of inexact data `wd` and compare the data-driven methods with model-based ones.

A Structured matrices

Studying the behavior of discrete-time linear dynamical systems over a finite horizon is an application of linear algebra. When the systems, in addition to linear, are also time-invariant, the matrices involved have Hankel structure. The exercises in this section introduce three types of structured matrices—Hankel, mosaic Hankel, and polynomial multiplication. These structures are used for analysis, identification, and control of linear time-invariant systems.

The term *structure* in "structured matrix" refers to a function $\mathscr{S} : \mathbb{R}^{n_p} \to \mathbb{R}^{m \times n}$ mapping a vector $p \in \mathbb{R}^{n_p}$, called *structure parameter vector*, to an $m \times n$ matrix S. We say that a matrix D has the structure $\mathscr{S}$ if it is in the image of $\mathscr{S}$, *i.e.*, there is a p, such that $D = \mathscr{S}(p)$. In the exercises, you will

- prove that the Hankel, mosaic Hankel, and polynomial multiplication structures are linear (*i.e.*, $\mathscr{S}$ is linear);
- derive the orthogonal projector $\Pi_{\mathscr{S}}$ on image $\mathscr{S}$, *i.e.*, a function $\Pi_{\mathscr{S}} : \mathbb{R}^{m \times n} \to \mathbb{R}^{m \times n}$ solving the problem

$$\text{minimize} \quad \text{over } \widehat{p} \quad \|D - \mathscr{S}(\widehat{p})\|_{\mathrm{F}}, \tag{12}$$

  where $\|\cdot\|_{\mathrm{F}}$ is the Frobenius norm; and
- write functions that implement $\mathscr{S}$ and $\Pi_{\mathscr{S}}$.

Hankel structure

A Hankel matrix $\mathscr{H}_L(w)$ with L block rows, constructed from the finite vector time-series

$$w = \big(w(1), \ldots, w(T)\big), \quad w(t) \in \mathbb{R}^q, \tag{13}$$

is defined as

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

Exercise 38 (Hankel matrix constructor `blkhank` and projector `Pblkhank`).

1. Explain how $\mathscr{H}_L$ defines a matrix structure (*i.e.*, what are $\mathscr{S}$ and p).
2. Write a function `blkhank` that takes as inputs w and L and returns as an output $\mathscr{H}_L(w)$. (w is represented by a $q \times T$ matrix `w`, where `w(:, t)` is equal to $w(t)$.)
3. Show that the Hankel structure $\mathscr{H}_L$ is linear.
4. Derive the orthogonal projector $\Pi_{\mathscr{H}_L}$ on image $\mathscr{H}_L$.
5. Write a function `Pblkhank` that takes as inputs an $qL \times (T-L+1)$ matrix D and an integer L and returns as an output $\widehat{w}$, such that $\mathscr{H}_L(\widehat{w}) = \Pi_{\mathscr{H}_L} D$.

Solution **SOL**

1. Define the vectorization operation for a time series (13)

$$\mathrm{vec}(w) := \begin{bmatrix} w(1) \\ \vdots \\ w(T) \end{bmatrix} \in \mathbb{R}^{qT}.$$

Then, with $p := \mathrm{vec}(w)$, we have $\mathscr{S}(p) := \mathscr{H}_L(w)$.

2. An implementation of the Hankel matrix $\mathscr{H}_L(w)$ constructor is the function `blkhank`.

```
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, error('Not enough data.'), else,
  H = zeros(L * q, j);
  for i = 1:L, H(((i - 1) * q + 1):(i * q), :) = w(:, i:(i + j - 1)); end
end
```

Here is a test example

```
blkhank([1:7; 8:14], 3)
```

which gives

```
 1     2     3     4     5
 8     9    10    11    12
 2     3     4     5     6
 9    10    11    12    13
 3     4     5     6     7
10    11    12    13    14
```

3. Since $\mathscr{H}_L(\alpha w + \beta v) = \alpha \mathscr{H}_L(w) + \beta \mathscr{H}_L(v)$, for all $w, v \in (\mathbb{R}^q)^T$ and $\alpha, \beta \in \mathbb{R}$, $\mathscr{H}_L$ is linear.

4. A linear structure $\mathscr{S}$ has a representation

$$\mathscr{S}(p) = S_1 p_1 + \cdots + S_{n_p} p_{n_p},$$

for some basis matrices $S_i \in \mathbb{R}^{m \times n}$. We have,

$$\|D - \mathscr{S}(\widehat{p})\|_{\mathrm{F}} = \|\mathrm{vec}(D) - \mathrm{vec}\left(\mathscr{S}(\widehat{p})\right)\| = \|\mathrm{vec}(D) - \mathrm{vec}(S_1)p_1 - \cdots - \mathrm{vec}(S_{n_p})p_{n_p}\| = \|\mathrm{vec}(D) - \mathbf{S}p\|,$$

where $\mathbf{S} := \begin{bmatrix} \mathrm{vec}(S_1) & \cdots & \mathrm{vec}(S_{n_p}) \end{bmatrix}$. Then, (12) is a standard least-squares problem

$$\text{minimize} \quad \text{over } \widehat{p} \quad \|\mathrm{vec}(D) - \mathbf{S}p\|.$$

The projector is

$$\Pi_{\mathscr{S}} = \mathbf{S}^{\top}(\mathbf{S}\mathbf{S}^{\top})^{-1}. \tag{14}$$

This gives a solution to the projection problem (12) for a general linear structure $\mathscr{S}$.

In the case of a Hankel structure $\mathscr{H}_L$, the general solution simplifies to averaging the block-elements of D along the anti-diagonals. In order to show this, consider first the case of scalar Hankel structure. Let d_i be the i-th anti-diagonal of D and $\mathbf{1}$ a vector of all ones. We have,

$$\|D - \mathscr{H}_L(\widehat{p})\|_{\mathrm{F}} = \sum_{k=1}^{n_p} \|d_k - \mathbf{1}\widehat{p}_k\|_2.$$

Then, the minimization problem (12) is separable: it decomposes into n_p independent optimization problems

$$\text{minimize} \quad \text{over } \widehat{p}_k \quad \|d_k - \mathbf{1}\widehat{p}_k\|_2.$$

The solution is the mean

$$\widehat{p}_k = (\mathbf{1}^\top \mathbf{1})^{-1} \mathbf{1}^\top d_k.$$

Consider now a general (block) Hankel structure. The optimization problem (12) is again separable in $\widehat{p}_k$. Let D_{ij} be the (i,j)-th block of D. The subproblems are

$$\text{minimize} \quad \text{over } \widehat{p}_k \quad \sum_{i+j=k-1} \|d_{ij} - \mathbf{1}\widehat{p}_k\|_{\mathrm{F}}^2.$$

The solution $\widehat{p}_i$ is the average of all elements in the k-th anti-diagonal of D.

5. An implementation of the projector $\Pi_{\mathscr{H}_L}$ is the function `Pblkhank`

```
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)';
```

Here is a test example

```
Pblkhank(blkhank([1:7; 8:14], 3), 3)'
```

which gives

```
1     2     3     4     5     6     7
8     9    10    11    12    13    14
```

Mosaic Hankel structure

The mosaic Hankel matrix with L block rows, constructed from the set of finite vector time-series

$$w := \{w^1, \ldots, w^N\}, \quad w^i = \big(w^i(1), \ldots, w^i(T_i)\big), \quad w^i(t) \in \mathbb{R}^q$$

is defined as

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

Exercise 39 (Mosaic Hankel matrix constructor `moshank` and projector `Pmoshank`).

1. Explain how $\mathscr{H}_L$ defines a matrix structure (*i.e.*, what are $\mathscr{S}$ and p).
2. Write a function `moshank` that takes as inputs w and L and returns as an output $\mathscr{H}_L(w)$. (The set of time series w is represented by a $1 \times N$ cell array `w` of $q \times T_i$ matrices `w{i}`, where `w{i}(:, t)` is equal to $w^i(t)$.)
3. Show that the mosaic Hankel structure $\mathscr{H}_L$ is linear.
4. Derive the orthogonal projector $\Pi_{\mathscr{H}_L}$ on image $\mathscr{H}_L$.
5. Write a function `Pmoshank` that takes as inputs an $qL \times \sum_{k=1}^N n_i$, $n_i := T_i - L + 1$ matrix D and $n = \begin{bmatrix} n_1 & \cdots & n_N \end{bmatrix}$ and returns as an output $\widehat{w}$, such that $\mathscr{H}_L(\widehat{w}) = \Pi_{\mathscr{H}_L} D$.

Solution **SOL**

1. With

$$p := \begin{bmatrix} \mathrm{vec}(w^1) \\ \vdots \\ \mathrm{vec}(w^N) \end{bmatrix} \in \mathbb{R}^{qT},$$

where $T := \sum_{k=1}^N T_k$, define $\mathscr{S}(p) := \mathscr{H}_L(w)$.

2. An implementation of the mosaic Hankel structure $\mathscr{H}_L$ is the function `moshank`.

```
function H = moshank(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
```

Here is a test example

```
moshank({1:7, 8:14}, 3)
```

which gives

```
1     2     3     4     5     8     9    10    11    12
2     3     4     5     6     9    10    11    12    13
3     4     5     6     7    10    11    12    13    14
```

3. The proof is identical to the one for the Hankel structure.

4. The projection problem reduces to the one for Hankel matrices:

$$\|D - \mathscr{H}_L(w^1,\dots,w^N)\|_{\mathrm{F}}^2 = \sum_{k=1}^{N} \|D_k - \mathscr{H}_L(w^k)\|_{\mathrm{F}}^2,$$

where D_k is the submatrix of D corresponding to w^k.

5. An implementation of $\Pi\mathscr{H}_L$ for mosaic Hankel structure is the function `Pmoshank`

```
function wh = Pmoshank(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
```

Here is a test example

```
wh = Pmoshank(moshank({1:7; 8:14}, 3), 3, [5 5]); [wh{1}'; wh{2}']
```

which gives

```
1     2     3     4     5     6     7
8     9    10    11    12    13    14
```

Polynomial multiplication structure

The multiplication matrix $\mathscr{M}_T(R)$ with T block columns related to the polynomial

$$R(z) = R_0 z^0 + R_1 z^1 + \cdots + R_\ell z^\ell \in \mathbb{R}^{g\times q}[z]. \tag{15}$$

is defined as

$$\mathscr{M}_T(R) := \begin{bmatrix} R_0 & R_1 & \cdots & R_\ell & & & \\ & R_0 & R_1 & \cdots & R_\ell & & \\ & & \ddots & \ddots & & \ddots & \\ & & & R_0 & R_1 & \cdots & R_\ell \end{bmatrix} \in \mathbb{R}^{g(T-\ell)\times qT} \tag{16}$$

The polynomial R is represented by a $g \times q(\ell+1)$ matrix

$$R = \begin{bmatrix} R_0 & R_1 & \cdots & R_\ell z^\ell \end{bmatrix} \in \mathbb{R}^{g\times q(\ell+1)}. \tag{17}$$

Exercise 40 (Polynomial multiplication matrix constructor `mulmat` and projector `Pmulmat`).

1. Explain how $\mathscr{M}_T$ defines a matrix structure (*i.e.*, what are $\mathscr{S}$ and p).

2. Write a function `multmat` that takes as inputs R and T and returns as an output $\mathscr{M}_T(R)$.

3. Show that the polynomial multiplication structure $\mathscr{M}_T$ is linear.

4. Derive the orthogonal projector $\Pi_{\mathscr{M}_T}$ on image $\mathscr{M}_T$.

5. Write a function `Pmultmat` that takes as inputs $g(T-\ell)\times qT$ matrix D, p, and q and returns as an output $\widehat{R}$, such that $\mathscr{M}_T(\widehat{R}) = \Pi_{\mathscr{M}_T} D$.

Solution **SOL**

1. With $p := \mathrm{vec}(R) \in \mathbb{R}^{g(\ell+1)}$, define $\mathscr{S}(p) := \mathscr{M}_T(R)$.

2. An implementation of the polynomial multiplication structure $\mathscr{M}_T$ is the function `multmat`.

```
function M = multmat(R, T, q)
if ~exist('q') || isempty(q), q = 1; end
[g, nc] = size(R); n = T - nc / q + 1;
M = zeros(n * g, nc + (n - 1) * q);
for i = 1:n, M((1:g) + (i - 1) * g, (1:nc) + (i - 1) * q) = R; end
```

3. Since $\mathscr{M}_T(\alpha R+\beta S) = \alpha\mathscr{M}_T(R)+\beta\mathscr{M}_T(S)$, for all $R,S\in\mathbb{R}^{p\times q}$ and $\alpha,\beta\in\mathbb{R}$, $\mathscr{M}_T$ is linear.

4. In the case of a polynomial multiplication structure $\mathscr{M}_T$, the general solution (14) of (12) simplifies to averaging the block-elements of D along the first $\ell+1$ diagonals.

5. An implementation of $\Pi_{\mathscr{M}_T}$ for mosaic Hankel structure is the function `Pmultmat`

```
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), T);
for k = 1:np, ph(k) = mean(D(I == k)); end
Rh = reshape(ph, p, ell + 1);
```

Summary and discussion

We focused on the "mechanics" of the matrix structures $\mathscr{H}_L$ and $\mathscr{M}_T$ without explanation why they are important. The Hankel and mosaic Hankel matrices are used in the data-driven representation of the finite horizon behavior of a linear time-invariant system. The polynomial multiplication matrix $\mathscr{M}_T$, as its name suggests, is used for computing the product of polynomials. Indeed, $c := b\mathscr{M}_T(a)$ defines the product of the polynomials defined by a and b. Check it numerically:

```
a  = 1:4; b = 1:4;
c  = conv(a, b);                % -> 1     4    10    20    25    24    16
c_ = b * multmat([1 2 3 4], 7) % -> 1     4    10    20    25    24    16
```

A polynomial (15) is represented by the matrix (17) of its coefficients. (17) can represent also the matrix-valued sequence $(R_0, R_1, \cdots, R_\ell)$. With some abuse of notation, we use the same letter R to denote the polynomial, the matrix, and the corresponding sequence of coefficients. Depending on the context, either the polynomial or the sequence are the primary object of interest, while the matrix is the way of store and manipulate it analytically and computationally.

References

[1] J. Coulson, J. Lygeros, and F. Dörfler. "Distributionally robust chance constrained data-enabled predictive control". In: *IEEE Trans. Automat. Contr.* 67 (2022), pp. 3289–3304.

[2] F. Dörfler, J. Coulson, and I. Markovsky. "Bridging direct & indirect data-driven control formulations via regularizations and relaxations". In: *IEEE Trans. Automat. Contr.* (2023). DOI: 10.1109/TAC.2022.3148374.

[3] P. Dreesen and I. Markovsky. “Data-Driven Simulation Using The Nuclear Norm Heuristic”. In: *In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*. Brighton, UK, 2019. DOI: 10.1109/icassp.2019.8682993.

[4] E. Frazzoli, M. A. Dahleh, and E. Feron. “Real-time motion planning for agile autonomous vehicles”. In: *Proc. American Control Conf.* Vol. 1. 2001, pp. 43–49.

[5] R. Hamming. *Numerical Methods for Scientists and Engineers*. Dover Publications, 1962.

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

[7] P. Lemmerling and B. De Moor. “Misfit versus latency”. In: *Automatica* 37 (2001), pp. 2057–2067.

[8] I. Markovsky. *Low-Rank Approximation: Algorithms, Implementation, Applications*. 2nd edition. Springer, 2019. ISBN: 978-3-319-89619-9. DOI: 10.1007/978-3-319-89620-5.

[9] I. Markovsky. “On the most powerful unfalsified model for data with missing values”. In: *Systems & Control Lett.* 95 (2016), pp. 53–61. DOI: 10.1016/j.sysconle.2015.12.012.

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

[11] I. Markovsky and F. Dörfler. “Data-driven dynamic interpolation and approximation”. In: *Automatica* 135 (2022), p. 110008. DOI: 10.1016/j.automatica.2021.110008.

[12] I. Markovsky and F. Dörfler. “Identifiability in the Behavioral Setting”. In: *IEEE Trans. Automat. Contr.* (2023). DOI: 10.1109/TAC.2022.3209954.

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

[14] I. Markovsky and K. Usevich. “Structured low-rank approximation with missing data”. In: *SIAM J. Matrix Anal. Appl.* 34.2 (2013), pp. 814–830. DOI: 10.1137/120883050.

[15] Eric Schulte et al. “A Multi-Language Computing Environment for Literate Programming and Reproducible Research”. In: *Journal of Statistical Software* 46.3 (Jan. 2012), pp. 1–24.