mdof

Release 0.0.7

Chrystal Chern

Nov 13, 2023

CONTENTS:

CHAPTER

ONE

REFERENCE

1.1 modules

1.1.1 mdof.markov module

mdof.markov.**okid**(*inputs*, *outputs*, ***options*)

Identify Markov parameters, or discrete impulse response data, for a given set of input and output data. Observer Kalman Identification Algorithm (OKID) (Juang, Phan, Horta, Longman, 1993).

Parameters

- **inputs** (*array*) – input time history. dimensions: (q, nt), where q = number of inputs, and nt = number of timesteps
- **outputs** (*array*) – output response history. dimensions: (p, nt), where p = number of outputs, and nt = number of timesteps
- **m** (*int, optional*) – number of Markov parameters to compute. default: $min(300, nt)$

Returns

the Markov parameters, with dimensions $(p, q, m + 1)$

Return type

array

1.1.2 mdof.realize module

mdof.realize.**era**(*Y*, ***options*)

System realization from Markov parameters (discrete impulse response data). Eigensystem Realization Algorithm (ERA) (Ho and Kalman 1966, Juang and Pappa 1985).

Parameters

- **Y** (*array*) – Markov parameters. dimensions: (p, q, nt), where p = number of outputs, q = number of inputs, and nt = number of Markov parameters.
- **horizon** (*int, optional*) – number of block rows in Hankel matrix = order of observability matrix. default: $min(150, (nt - 1)/2)$
- **nc** (*int, optional*) – number of block columns in Hankel matrix = order of controllability matrix. default: $min(150, max(nt - 1 -$ horizon$, (nt - 1)/2))$
- **order** (*int, optional*) – model order. default: $min(20,$ horizon$/2)$

Returns
realization in the form of state space coefficients `(A,B,C,D)`

Return type
tuple of arrays

mdof.realize.**era_dc**(*Y*, ***options*)

System realization from Markov parameters (discrete impulse response data). Eigensystem Realization Algorithm with Data Correlations (ERA/DC) (Juang, Cooper, and Wright, 1988).

Parameters

- **Y** (*array*) – Markov parameters. dimensions: (p, q, nt), where p = number of outputs, q = number of inputs, and nt = number of Markov parameters.
- **horizon** (*int, optional*) – number of block rows in Hankel matrix = order of observability matrix. default: $min(150, (nt-1)/2)$
- **nc** (*int, optional*) – number of block columns in Hankel matrix = order of controllability matrix. default: $min(150, max(nt-1-$ `horizon`$, (nt-1)/2))$
- **order** (*int, optional*) – model order. default: $min(20,$ `horizon`$/2)$
- **a** (*int, optional*) – (α) number of block rows in Hankel of correlation matrix. default: 0
- **b** (*int, optional*) – (β) number of block columns in Hankel of correlation matrix. default: 0
- **l** (*int, optional*) – initial lag for data correlations. default: 0
- **g** (*int, optional*) – lags (gap) between correlation matrices. default: 1

Returns
realization in the form of state space coefficients `(A,B,C,D)`

Return type
tuple of arrays

mdof.realize.**srim**(*inputs*, *outputs*, ***options*)

System realization from input and output data, with output error minimization method. System Realization Using Information Matrix (SRIM) (Juang, 1997).

Parameters

- **inputs** (*array*) – input time history. dimensions: (q, nt), where q = number of inputs, and nt = number of timesteps
- **outputs** (*array*) – output response history. dimensions: (p, nt), where p = number of outputs, and nt = number of timesteps
- **horizon** (*int, optional*) – number of steps used for identification (prediction horizon). default: $min(300, nt)$
- **order** (*int, optional*) – model order. default: $min(20,$ `horizon`$/2)$
- **full** (*bool, optional*) – if True, full SVD. default: True
- **find** (*string, optional*) – “ABCD” or “AC”. default: “ABCD”
- **threads** (*int, optional*) – number of threads used during the output error minimization method. default: 6
- **chunk** (*int, optional*) – chunk size in output error minimization method. default: 200

Returns
realization in the form of state space coefficients `(A,B,C,D)`

Return type
tuple of arrays

1.1.3 mdof.modal module

This module implements functions that extract modal information from a state space realization.

`mdof.modal.`**`system_modes`**(*realization*, *dt*, ***options*)

Modal identification from a state space system realization.

Parameters

- **`realization`** (*`tuple`*) – realization in the form of state space coefficients `(A,B,C,D)`
- **`dt`** (*`float`*) – timestep.
- **`decimation`** (*`int, optional`*) – decimation factor. default: 1
- **`Observability`** (*`array, optional`*) – Observability matrix; can be reused from *`mdof.realize.srim()`*. default: None

Returns
system modes, including natural frequencies, damping ratios, mode shapes, condition numbers, and modal validation metrics EMAC and MPC.

Return type
dictionary

`mdof.modal.`**`spectrum_modes`**(*periods*, *amplitudes*, ***options*)

Modal identification from a transfer function.

Parameters

- **`periods`** (*`array`*) – transfer function periods
- **`amplitudes`** (*`array`*) – transfer function amplitudes

Returns
(fundamental_periods, fundamental_amplitudes)

Return type
tuple

1.1.4 mdof.transform module

`mdof.transform.`**`response_transfer`**(*inputs*, *outputs*, *step*, ***options*)

Response spectrum transfer function from input and output data.

Parameters

- **`inputs`** (*`array`*) – input time history. dimensions: (q, nt), where q = number of inputs, and nt = number of timesteps
- **`outputs`** (*`array`*) – output response history. dimensions: (p, nt), where p = number of outputs, and nt = number of timesteps
- **`step`** (*`float`*) – timestep.

- **pseudo** (*bool, optional*) – if True, uses pseudo accelerations. default: False
- **decimation** (*int, optional*) – decimation factor. default: 1

Returns
(periods, amplitudes)

Return type
tuple of arrays

mdof.transform.**fourier_transfer**(*inputs*, *outputs*, *step*, ***options*)

Fourier spectrum transfer function from input and output data.

Parameters

- **inputs** (*array*) – input time history. dimensions: (q, nt), where q = number of inputs, and nt = number of timesteps
- **outputs** (*array*) – output response history. dimensions: (p, nt), where p = number of outputs, and nt = number of timesteps
- **step** (*float*) – timestep.
- **decimation** (*int, optional*) – decimation factor. default: 1

Returns
(periods, amplitudes)

Return type
tuple of arrays

mdof.transform.**fourier_spectrum**(*series*, *step*, *period_band=None*, ***options*)

Fourier amplitude spectrum from a signal.

Parameters

- **series** (*1D array*) – time series.
- **step** (*float*) – timestep.
- **period_band** – minimum and maximum period of interest, in seconds.

Returns
(frequencies, amplitudes)

Return type
tuple of arrays.

1.2 top-level functions

mdof.**impulse**(*system*, *t*, ***kwds*)

mdof.**sysid**(*inputs*, *outputs*, ***options*)

State space system realization from input and output data.

Parameters

- **inputs** (*array*) – input time history. dimensions: (q, nt), where q = number of inputs, and nt = number of timesteps
- **outputs** (*array*) – output response history. dimensions: (p, nt), where p = number of outputs, and nt = number of timesteps

- **method** (*string, optional*) – system identification method. default is "srim", other options are "okid-era" and "okid-era-dc".
- **decimation** (*int, optional*) – decimation factor. default: 8

Returns
system realization in the form of state space coefficients `(A,B,C,D)`

Return type
tuple of arrays

`mdof.`**eigid**(*inputs*, *outputs*, ***options*)

System eigenspace identification from input and output data.

Parameters

- **inputs** (*array*) – input time history. dimensions: (q, nt), where q = number of inputs, and nt = number of timesteps
- **outputs** (*array*) – output response history. dimensions: (p, nt), where p = number of outputs, and nt = number of timesteps
- **dt** (*float*) – timestep.
- **decimation** (*int, optional*) – decimation factor. default: 1

Returns
(eigenvalues, eigenvectors)

Return type
tuple of 1D array, ND array

CHAPTER

TWO

THEORY

2.1 State Space Model of Structural Dynamics

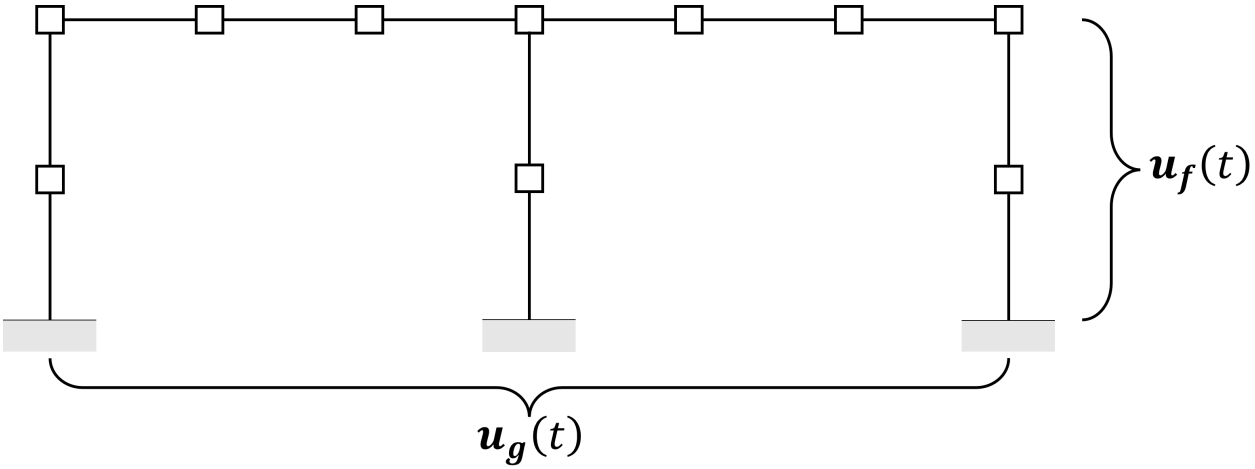


Fig. 1: MDOF Structure

When an multiple degree-of-freedom (MDOF) system is subject to multiple support excitation, such as in the figure above, the displacement vector is extended to include the support DOF. An *equation of motion* is derived as follows.

Begin by forming a partitioned equation of dynamic equilibrium for all the DOF:

2.1.1 Partitioned Equation of Dynamic Equilibrium

$$\begin{bmatrix} \mathbf{m} & \mathbf{m}_g \\ \mathbf{m}_g^T & \mathbf{m}_{gg} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{u}}_f^t \\ \ddot{\mathbf{u}}_g \end{bmatrix} + \begin{bmatrix} \mathbf{c} & \mathbf{c}_g \\ \mathbf{c}_g^T & \mathbf{c}_{gg} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{u}}_f^t \\ \dot{\mathbf{u}}_g \end{bmatrix} + \begin{bmatrix} \mathbf{k} & \mathbf{k}_g \\ \mathbf{k}_g^T & \mathbf{k}_{gg} \end{bmatrix} \begin{bmatrix} \mathbf{u}_f^t \\ \mathbf{u}_g \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{p}_g \end{bmatrix}$$

where the subscript g indicates support DOF, the subscript f indicates structural DOF, and the superscript t indicates the total of quasi-static ($\mathbf{u}_f^s$, due to static application of support displacements) and dynamic ($\mathbf{u}_f$, evaluated by dynamic analysis) structural displacements.

Taking the first half of the partitioned equilibrium, separating the structural displacements ($\mathbf{u}_f^t = \mathbf{u}_f^s + \mathbf{u}_f$), and moving all $\mathbf{u}_g$ and $\mathbf{u}_f^s$ terms to the right side,

$$\mathbf{m}\ddot{\mathbf{u}}_f + \mathbf{c}\dot{\mathbf{u}}_f + \mathbf{k}\mathbf{u}_f = -(\mathbf{m}\ddot{\mathbf{u}}_f^s + \mathbf{m}_g\ddot{\mathbf{u}}_g) - (\mathbf{c}\dot{\mathbf{u}}_f^s + \mathbf{c}_g\dot{\mathbf{u}}_g) - (\mathbf{k}\mathbf{u}_f^s + \mathbf{k}_g\mathbf{u}_g)$$

The term $(\mathbf{k}\mathbf{u}_f^s + \mathbf{k}_g\mathbf{u}_g) = \mathbf{0}$ due to static equilibrium, allowing the term to be dropped and giving $\mathbf{u}_f^s = -\mathbf{k}^{-1}\mathbf{k}_g\mathbf{u}_g = \iota\mathbf{u}_g$; the term $(\mathbf{c}\dot{\mathbf{u}}_f^s + \mathbf{c}_g\dot{\mathbf{u}}_g)$ is dropped because it is usually small relative to the inertia term; and the term $\mathbf{m}_g\ddot{\mathbf{u}}_g$ is dropped because mass is usually neglected at supports.

The equilibrium equation thus simplifies.

2.1.2 Equation of Motion

$$\mathbf{M}\ddot{\mathbf{u}}_f(t) + \mathbf{Z}\dot{\mathbf{u}}_f(t) + \mathbf{K}\mathbf{u}_f(t) = -\mathbf{M}\iota\ddot{\mathbf{u}}_g(t)$$
$$\mathbf{m}\ddot{\mathbf{u}}_f + \mathbf{c}\dot{\mathbf{u}}_f + \mathbf{k}\mathbf{u}_f = -\mathbf{m}\iota\ddot{\mathbf{u}}_g$$

Hence, the following equation presents the continuous linear time-invariant (LTI) state-space representation of a structural system.

2.1.3 Continuous LTI State-Space Representation

$$\dot{\mathbf{x}} = \mathbf{A}_c\mathbf{x} + \mathbf{B}_c\mathbf{u}$$
$$\begin{bmatrix}\dot{\mathbf{u}}_f(t)\\ \ddot{\mathbf{u}}_f(t)\end{bmatrix} = \begin{bmatrix}\mathbf{0} & \mathbf{I}\\ -\mathbf{M}^{-1}\mathbf{K} & -\mathbf{M}^{-1}\mathbf{Z}\end{bmatrix}\begin{bmatrix}\mathbf{u}_f(t)\\ \dot{\mathbf{u}}_f(t)\end{bmatrix} + \begin{bmatrix}\mathbf{0}\\ -\iota\end{bmatrix}\ddot{\mathbf{u}}_g(t)$$

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u}$$
$$\ddot{\mathbf{u}}_f(t) = \begin{bmatrix}-\mathbf{M}^{-1}\mathbf{K} & -\mathbf{M}^{-1}\mathbf{Z}\end{bmatrix}\begin{bmatrix}\mathbf{u}_f(t)\\ \dot{\mathbf{u}}_f(t)\end{bmatrix} + \begin{bmatrix}-\iota\end{bmatrix}\ddot{\mathbf{u}}_g(t)$$

In order to move from the continuous to the discrete case, the coefficients $\mathbf{A}_c$ and $\mathbf{B}_c$ are transformed by solving the first-order differential equation with the signal's value held constant between time steps ("zero-order hold method"). The coefficients $\mathbf{C}$ and $\mathbf{D}$ are unchanged. The results are shown in the following equation.

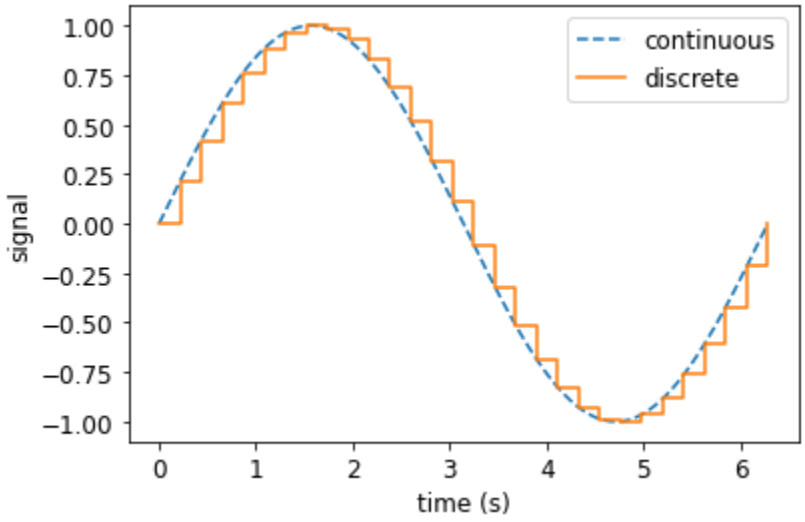


Fig. 2: Signal Discretization

2.1.4 Discrete LTI State-Space Representation

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k \end{aligned}$$

$$\mathbf{x}_k = \mathbf{x}(k\Delta t), \qquad \mathbf{u}_k = \mathbf{u}(k\Delta t), \qquad \mathbf{y}_k = \mathbf{y}(k\Delta t)$$

$$\mathbf{A} = e^{\mathbf{A}_c \Delta t}, \qquad \mathbf{B} = \int_0^{\Delta t} e^{\mathbf{A}_c \tau} \mathbf{B}_c d\tau$$

where:

$\mathbf{A}$: discrete state transition matrix

$\mathbf{B}$: discrete input influence matrix

$\mathbf{C}$: output influence matrix

$\mathbf{D}$: direct transmission or feed-through matrix

2.2 Eigensystem Realization Algorithm (ERA)

As shown in the previous section, a structural system's dynamic behavior can be represented by the four coefficients $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ of its discrete LTI state-space representation. The Ho-Kalman Algorithm, or Eigensystem Realization Algorithm, produces a *reduced order model* for these four coefficients, $(\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}}, \tilde{\mathbf{D}})$, based on an impulse input and its corresponding response output. Then, modal properties can be extracted from $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{C}}$.

With the discrete LTI model, a unit impulse input with zero initial conditions produces an output of constants $(\mathbf{D}, \mathbf{CB}, \mathbf{CAB}, ..., \mathbf{CA}^{k-1}\mathbf{B})$. These constants are called *Markov parameters* because they must be unique for a given system – there is only one possible output for a unit impulse input.

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k \end{aligned}$$

$$\begin{aligned} \mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_k &= \mathbf{I}, \mathbf{0}, \mathbf{0}, ..., \mathbf{0} \\ \mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_k &= \mathbf{0}, \mathbf{B}, \mathbf{AB}, ..., \mathbf{A}^{k-1}\mathbf{B} \\ \mathbf{y}_0, \mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_k &= \mathbf{D}, \mathbf{CB}, \mathbf{CAB}, ..., \mathbf{CA}^{k-1}\mathbf{B} \end{aligned}$$

Knowing that the impulse response output data directly give the Markov parameters, the data can then be stacked into the generalized blockwise Hankel matrix $\mathbf{H}$:

$$\mathbf{H} = \begin{bmatrix} \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_{m_c} \\ \mathbf{y}_2 & \mathbf{y}_3 & \cdots & \mathbf{y}_{m_c+1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{y}_{m_o} & \mathbf{y}_{m_o+1} & \cdots & \mathbf{y}_{m_o+m_c-1} \end{bmatrix} = \begin{bmatrix} \mathbf{CB} & \mathbf{CAB} & \cdots & \mathbf{CA}^{m_c-1}\mathbf{B} \\ \mathbf{CAB} & \mathbf{CA}^2\mathbf{B} & \cdots & \mathbf{CA}^{m_c}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{CA}^{m_o-1}\mathbf{B} & \mathbf{CA}^{m_o}\mathbf{B} & \cdots & \mathbf{CA}^{m_c+m_o-2}\mathbf{B} \end{bmatrix} = \mathcal{O}\mathcal{C}$$

where $\mathcal{O}$ and $\mathcal{C}$ are the observability and controllability matrices of the system:

$$\mathcal{O} = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \mathbf{CA}^2 \\ \vdots \\ \mathbf{CA}^{m_o-1} \end{bmatrix}, \qquad \mathcal{C} = \begin{bmatrix} \mathbf{B} & \mathbf{AB} & \mathbf{A}^2\mathbf{B} & \cdots & \mathbf{A}^{m_c-1}\mathbf{B} \end{bmatrix}$$

The shifted Hankel matrix, $\mathbf{H}'$ (one time step ahead of $\mathbf{H}$), is shown below:

$$\mathbf{H}' = \begin{bmatrix} \mathbf{y}_2 & \mathbf{y}_3 & \cdots & \mathbf{y}_{m_c+1} \\ \mathbf{y}_3 & \mathbf{y}_4 & \cdots & \mathbf{y}_{m_c+2} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{y}_{m_o+1} & \mathbf{y}_{m_o+2} & \cdots & \mathbf{y}_{m_o+m_c} \end{bmatrix} = \begin{bmatrix} \mathbf{CAB} & \mathbf{CA^2B} & \cdots & \mathbf{CA}^{m_c}\mathbf{B} \\ \mathbf{CA^2B} & \mathbf{CA^3B} & \cdots & \mathbf{CA}^{m_c+1}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{CA}^{m_o}\mathbf{B} & \mathbf{CA}^{m_o+1}\mathbf{B} & \cdots & \mathbf{CA}^{m_c+m_o-1}\mathbf{B} \end{bmatrix} = \mathcal{O}\mathbf{A}\mathcal{C}$$

By taking the dominant terms of the singular value decomposition of $\mathbf{H}$, and transforming the relationship between $\mathbf{H} = \mathcal{O}\mathcal{C}$ and $\mathbf{H}' = \mathcal{O}\mathbf{A}\mathcal{C}$, a reduced-order model is constructed as follows:

$$\mathbf{H} = \mathbf{U}\Sigma\mathbf{V}^H = \begin{bmatrix} \tilde{\mathbf{U}} & \mathbf{U}_t \end{bmatrix} \begin{bmatrix} \tilde{\Sigma} & \mathbf{0} \\ \mathbf{0} & \Sigma_t \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{V}}^H \\ \mathbf{V}_t^H \end{bmatrix} \approx \tilde{\mathbf{U}}\tilde{\Sigma}\tilde{\mathbf{V}}^H$$

where the superscript H denotes conjugate transpose and the subscript t indicates elements to be truncated such that only the first r dominant singular values in $\tilde{\Sigma}$ are retained,

$$\begin{aligned} \tilde{\mathbf{A}} &= \tilde{\Sigma}^{-1/2}\tilde{\mathbf{U}}^H\mathbf{H}'\tilde{\mathbf{V}}\tilde{\Sigma}^{-1/2} \\ \tilde{\mathbf{B}} &= \tilde{\Sigma}^{1/2}\tilde{\mathbf{V}}^H \begin{bmatrix} \mathbf{I}_q \\ \mathbf{0} \end{bmatrix} \\ \tilde{\mathbf{C}} &= \begin{bmatrix} \mathbf{I}_p & \mathbf{0} \end{bmatrix} \tilde{\mathbf{U}}\tilde{\Sigma}^{1/2} \\ \tilde{\mathbf{D}} &= \mathbf{y}_0 \end{aligned}$$

where p indicates the number of outputs and q the number of inputs, and

$$\begin{aligned} \tilde{\mathbf{x}}_{k+1} &= \tilde{\mathbf{A}}\tilde{\mathbf{x}}_k + \tilde{\mathbf{B}}\mathbf{u}_k \\ \mathbf{y}_k &= \tilde{\mathbf{C}}\tilde{\mathbf{x}}_k + \tilde{\mathbf{D}}\mathbf{u}_k. \end{aligned}$$

2.3 Observer Kalman Filter Identification (OKID)

Structural dynamics are noisy, hard to measure, and lightly damped, and ERA is intended only to characterize impulse responses rather than time histories. However, available data from ambient or small excitations during structure service can be de-noised and used to estimate impulse response data. Then, ERA can be used to obtain a reduced order model even if the available data are not a clean impulse response. This process is called Observer Kalman Identification, or OKID-ERA when combined with ERA.

When noise is incorporated into the discrete LTI state-space representation of a structural system, it becomes a *linear Gaussian model* of a *hidden Markov process*.

Because the data are assumed to follow a linear Gaussian model, Kalman filtering can estimate an impulse response that is most consistent with the input-output data. The estimated model after filtering is the same as that of ERA:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k \end{aligned}$$

Since the input is no longer an impulse, the state-space evolution includes more terms than ERA.

$$\begin{aligned} &\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_k := \text{given input} \\ &\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_k = \mathbf{0}, (\mathbf{Bu}_0), (\mathbf{ABu}_0 + \mathbf{Bu}_1), ..., (\mathbf{A}^{k-1}\mathbf{Bu}_0 + \mathbf{A}^{k-2}\mathbf{Bu}_1 + ... + \mathbf{Bu}_{k-1}) \\ &\mathbf{y}_0, \mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_k = \mathbf{Du}_0, (\mathbf{CBu}_0 + \mathbf{Du}_1), (\mathbf{CABu}_0 + \mathbf{CBu}_1 + \mathbf{Du}_2), ..., \\ &\qquad (\mathbf{CA}^{k-1}\mathbf{Bu}_0 + \mathbf{CA}^{k-2}\mathbf{Bu}_1 + ... + \mathbf{Du}_k). \end{aligned}$$

The output data can be expressed in terms of the Markov parameters and an upper triangular *data matrix* B built from the input data; however, inverting B is often computationally expensive or ill-conditioned.

$$\underbrace{\begin{bmatrix}\mathbf{y}_0 & \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_m\end{bmatrix}}_{\mathbf{S}} = \underbrace{\begin{bmatrix}\mathbf{y}_0 & \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_m\end{bmatrix}_\delta}_{\mathbf{S}_\delta} \underbrace{\begin{bmatrix}\mathbf{u}_0 & \mathbf{u}_1 & \cdots & \mathbf{u}_m \\ \mathbf{0} & \mathbf{u}_0 & \cdots & \mathbf{u}_{m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{u}_0\end{bmatrix}}_{B}$$

where the subscript δ indicates that the response comes from an impulse input.

The Kalman filter is applied by augmenting the system with the outputs $\mathbf{y}_i$ to form the *augmented data matrix* V:

$$V = \begin{bmatrix}\mathbf{u}_0 & \mathbf{u}_1 & \cdots & \mathbf{u}_l & \cdots & \mathbf{u}_m \\ \mathbf{0} & \mathbf{v}_0 & \cdots & \mathbf{v}_{l-1} & \cdots & \mathbf{v}_{m-1} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{v}_0 & \cdots & \mathbf{v}_{m-l}\end{bmatrix}, \qquad \mathbf{v}_i = \begin{bmatrix}\mathbf{u}_i \\ \mathbf{y}_i\end{bmatrix}.$$

Then, the Markov parameters (i.e., the impulse response) can be estimated as a function of the input and output data as follows:

$$\hat{\mathbf{S}}_\delta = \mathbf{S}V^\dagger$$

where the superscript † indicates pseudo-inverse,

Extract the estimated, or *observer*, Markov parameters from the block columns of $\hat{\mathbf{S}}_\delta$:

$$\begin{aligned}\hat{\mathbf{S}}_{\delta 0} &\in \mathbb{R}^{p\times q} \\ \hat{\mathbf{S}}_{\delta k} &= \begin{bmatrix}\hat{\mathbf{S}}_{\delta k}^{(1)} & \hat{\mathbf{S}}_{\delta k}^{(2)}\end{bmatrix}, \qquad k \in [1, 2, ...] \\ \hat{\mathbf{S}}_{\delta k}^{(1)} &\in \mathbb{R}^{p\times q}, \qquad \hat{\mathbf{S}}_{\delta k}^{(2)} \in \mathbb{R}^{p\times p}\end{aligned}$$

Reconstruct the system Markov parameters:

$$\mathbf{y}_{\delta 0} = \hat{\mathbf{S}}_{\delta 0} = \mathbf{D}, \qquad \mathbf{y}_{\delta k} = \hat{\mathbf{S}}_{\delta k}^{(1)} + \sum_{i=1}^{k} \hat{\mathbf{S}}_{\delta k}^{(2)} \mathbf{y}_{\delta(k-i)}.$$

2.4 System Realization by Information Matrix (SRIM)

For discrete-time systems, the correlation between inputs, outputs, and state yield information about the system's state evolution and response. In fact, the state equations can be estimated by manipulating correlation matrices through the method, System Realization by Information Matrix (SRIM).

2.4.1 Discrete-Time System Matrix Equation

We begin with discrete-time state equations that correspond to the structure's dynamics (see Discrete LTI State-Space Representation).

$$\begin{aligned}\mathbf{x}(k+1) &= \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k) \\ \mathbf{y}(k) &= \mathbf{C}\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k)\end{aligned}$$

By noting the state evolution

$$\begin{aligned}\mathbf{x}(k+1) &= \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{U}_p(k)\\ \mathbf{x}(k+2) &= \mathbf{A}^2\mathbf{X}(k) + \mathbf{A}\mathbf{B}\mathbf{U}_p(k) + \mathbf{B}\mathbf{u}(k+1)\\ \mathbf{x}(k+3) &= \mathbf{A}^3\mathbf{X}(k) + \mathbf{A}^2\mathbf{B}\mathbf{u}(k) + \mathbf{A}\mathbf{B}\mathbf{u}(k+1) + \mathbf{B}\mathbf{u}(k+2),\end{aligned}$$

we can generalize the response for the timepoint $k+p-1$:

$$\begin{aligned}\mathbf{x}(k+p) &= \mathbf{A}^p\mathbf{x}(k) + \sum_{i=1}^{p}\mathbf{A}^{p-i}\mathbf{B}\mathbf{u}(k+i-1)\\ \mathbf{x}(k+p-1) &= \mathbf{A}^{p-1}\mathbf{x}(k) + \sum_{i=1}^{p-1}\mathbf{A}^{p-i-1}\mathbf{B}\mathbf{u}(k+i-1)\\ \mathbf{y}(k+p-1) &= \mathbf{C}\mathbf{A}^{p-1}\mathbf{x}(k) + \sum_{i=1}^{p-1}\mathbf{C}\mathbf{A}^{p-i-1}\mathbf{B}\mathbf{u}(k+i-1) + \mathbf{D}\mathbf{u}(k+p-1)\,.\end{aligned}$$

Then, we can vertically stack p successive time-points into a column vector and express this vector as $\mathbf{y}_p(k)$:

$$\begin{aligned}\mathbf{y}_p(k) &= \mathcal{O}_p\mathbf{x}(k) + \mathcal{T}_p\mathbf{u}_p(k)\\ \begin{bmatrix}\mathbf{y}(k)\\ \mathbf{y}(k+1)\\ \vdots\\ \mathbf{y}(k+p-1)\end{bmatrix} &= \begin{bmatrix}\mathbf{C}\\ \mathbf{C}\mathbf{A}\\ \mathbf{C}\mathbf{A}^2\\ \vdots\\ \mathbf{C}\mathbf{A}^{p-1}\end{bmatrix}\mathbf{x}(k) +\\ &\quad \begin{bmatrix}\mathbf{D} & & & & \\ \mathbf{C}\mathbf{B} & \mathbf{D} & & & \\ \mathbf{C}\mathbf{A}\mathbf{B} & \mathbf{C}\mathbf{B} & \mathbf{D} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \mathbf{C}\mathbf{A}^{p-2}\mathbf{B} & \mathbf{C}\mathbf{A}^{p-3}\mathbf{B} & \mathbf{C}\mathbf{A}^{p-4}\mathbf{B} & \cdots & \mathbf{D}\end{bmatrix}\begin{bmatrix}\mathbf{u}(k)\\ \mathbf{u}(k+1)\\ \vdots\\ \mathbf{u}(k+p-1)\end{bmatrix}.\end{aligned}$$

Next, we horizontally stack N successive timepoints of these column vectors in a matrix, to get the matrix equation

$$\boxed{\mathbf{Y}_p(k) = \mathcal{O}_p\mathbf{X}(k) + \mathcal{T}_p\mathbf{U}_p(k)}\,,$$

where

$$\begin{aligned}\mathbf{Y}_p(k) &= \begin{bmatrix}\mathbf{y}_p(k) & \mathbf{y}_p(k+1) & \cdots & \mathbf{y}_p(k+N-1)\end{bmatrix}\\ &= \begin{bmatrix}\mathbf{y}(k) & \mathbf{y}(k+1) & \cdots & \mathbf{y}(k+N-1)\\ \mathbf{y}(k+1) & \mathbf{y}(k+2) & \cdots & \mathbf{y}(k+N)\\ \vdots & \vdots & \ddots & \vdots\\ \mathbf{y}(k+p-1) & \mathbf{y}(k+p) & \cdots & \mathbf{y}(k+N+p-2)\end{bmatrix}\\ \mathbf{X}(k) &= \begin{bmatrix}\mathbf{x}(k) & \mathbf{x}(k+1) & \cdots & \mathbf{x}(k+N-1)\end{bmatrix}\\ \mathbf{U}_p(k) &= \begin{bmatrix}\mathbf{u}_p(k) & \mathbf{u}_p(k+1) & \cdots & \mathbf{u}_p(k+N-1)\end{bmatrix}\\ &= \begin{bmatrix}\mathbf{u}(k) & \mathbf{u}(k+1) & \cdots & \mathbf{u}(k+N-1)\\ \mathbf{u}(k+1) & \mathbf{u}(k+2) & \cdots & \mathbf{u}(k+N)\\ \vdots & \vdots & \ddots & \vdots\\ \mathbf{u}(k+p-1) & \mathbf{u}(k+p) & \cdots & \mathbf{u}(k+N+p-2)\end{bmatrix}.\end{aligned}$$

2.4.2 Observability Matrix from Information Matrix

By post-multiplying the matrix equation by $\frac{1}{N}\mathbf{U}_p^T(k)$, $\frac{1}{N}\mathbf{Y}_p^T(k)$ or $\frac{1}{N}\mathbf{X}_p^T(k)$, we obtain relationships between correlation matrices $\mathbf{R}_{yy}$, $\mathbf{R}_{yu}$, $\mathbf{R}_{uu}$, and $\mathbf{R}_{xx}$ (See *Appendix*).

$$\mathbf{R}_{yy} - \mathbf{R}_{yu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{yu}^T = \mathcal{O}_p\mathbf{R}_{xx}\mathcal{O}_p^T \ ,$$

where

$$\mathbf{R}_{yy} = \frac{1}{N}\mathbf{Y}_p(k)\mathbf{Y}_p^T(k), \quad \mathbf{R}_{yu} = \frac{1}{N}\mathbf{Y}_p(k)\mathbf{U}_p^T(k)$$
$$\mathbf{R}_{uu} = \frac{1}{N}\mathbf{U}_p(k)\mathbf{U}_p^T(k), \quad \mathbf{R}_{xx} = \frac{1}{N}\mathbf{X}(k)\mathbf{X}^T(k) \ .$$

The left side of the equation is found from input and output measurements, and is called the *information matrix* of the data. Its singular value decomposition is computed to yield the *observability matrix* $\mathcal{O}_p$.

$$\mathbf{R}_{yy} - \mathbf{R}_{yu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{yu}^T = \mathbf{U}\Sigma\mathbf{U}^T = \mathcal{O}_p\mathbf{R}_{xx}\mathcal{O}_p^T \ .$$

2.4.3 State Equation Matrices from Observability Matrix

Now, the state equation matrices $\mathbf{A}$ and $\mathbf{C}$ can be obtained from the observability matrix $\mathcal{O}_p$.

$$\mathcal{O}_p = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \mathbf{CA}^2 \\ \vdots \\ \mathbf{CA}^{p-1} \end{bmatrix}, \quad \mathcal{O}_p(:-1) = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \mathbf{CA}^2 \\ \vdots \\ \mathbf{CA}^{p-2} \end{bmatrix}, \quad \mathcal{O}_p(1:) = \begin{bmatrix} \mathbf{CA} \\ \mathbf{CA}^2 \\ \mathbf{CA}^3 \\ \vdots \\ \mathbf{CA}^{p-1} \end{bmatrix}$$

$$\mathbf{A} = \mathcal{O}_p(:-1)^+\mathcal{O}_p(1:)$$

$$\mathbf{C} = \mathcal{O}_p(0)$$

2.4.4 Appendix: Manipulation of discrete-time system matrix equation into correlation matrix relationships

In (Juang 1997), the discrete-time system matrix equation is manipulated into a form describing the relationship between correlation matrices $\mathbf{R}_{yy}$, $\mathbf{R}_{yu}$, $\mathbf{R}_{uu}$, and $\mathbf{R}_{xx}$.

Post-multiplying the *discrete-time system matrix equation* by $\frac{1}{N}\mathbf{U}_p^T(k)$:

$$\frac{1}{N}\mathbf{Y}_p(k)\mathbf{U}_p^T(k) = \mathcal{O}_p\frac{1}{N}\mathbf{X}(k)\mathbf{U}_p^T(k) + \mathcal{T}_p\frac{1}{N}\mathbf{U}_p(k)\mathbf{U}_p^T(k)$$
$$\mathbf{R}_{yu} = \mathcal{O}_p\mathbf{R}_{xu} + \mathcal{T}_p\mathbf{R}_{uu}$$
$$\mathcal{T}_p = \left(\mathbf{R}_{yu} - \mathcal{O}_p\mathbf{R}_{xu}\right)\mathbf{R}_{uu}^{-1}$$

Post-multiplying by $\frac{1}{N}\mathbf{Y}_p^T(k)$:

$$\frac{1}{N}\mathbf{Y}_p(k)\mathbf{Y}_p^T(k) = \mathcal{O}_p\frac{1}{N}\mathbf{X}(k)\mathbf{Y}_p^T(k) + \mathcal{T}_p\frac{1}{N}\mathbf{U}_p(k)\mathbf{Y}_p^T(k)$$
$$\mathbf{R}_{yy} = \mathcal{O}_p\mathbf{R}_{yx}^T + \mathcal{T}_p\mathbf{R}_{yu}^T$$
$$\mathbf{R}_{yy} = \mathcal{O}_p\mathbf{R}_{yx}^T + \left(\mathbf{R}_{yu} - \mathcal{O}_p\mathbf{R}_{xu}\right)\mathbf{R}_{uu}^{-1}\mathbf{R}_{yu}^T$$

Post-multiplying by $\frac{1}{N}\mathbf{X}_p^T(k)$:

$$\begin{aligned}\frac{1}{N}\mathbf{Y}_p(k)\mathbf{X}_p^T(k) &= \mathcal{O}_p\frac{1}{N}\mathbf{X}(k)\mathbf{X}_p^T(k) + \mathcal{T}_p\frac{1}{N}\mathbf{U}_p(k)\mathbf{X}_p^T(k)\\ \mathbf{R}_{yx} &= \mathcal{O}_p\mathbf{R}_{xx} + \mathcal{T}_p\mathbf{R}_{xu}^T\\ \mathbf{R}_{yx} &= \mathcal{O}_p\mathbf{R}_{xx} + \left(\mathbf{R}_{yu} - \mathcal{O}_p\mathbf{R}_{xu}\right)\mathbf{R}_{uu}^{-1}\mathbf{R}_{xu}^T\end{aligned}$$

Substituting the equation for $\mathbf{R}_{yx}$ into the equation for $\mathbf{R}_{yy}$:

$$\begin{aligned}\mathbf{R}_{yy} &= \mathcal{O}_p\left(\mathcal{O}_p\mathbf{R}_{xx} + \left(\mathbf{R}_{yu} - \mathcal{O}_p\mathbf{R}_{xu}\right)\mathbf{R}_{uu}^{-1}\mathbf{R}_{xu}^T\right)^T\\ &\quad + \left(\mathbf{R}_{yu} - \mathcal{O}_p\mathbf{R}_{xu}\right)\mathbf{R}_{uu}^{-1}\mathbf{R}_{yu}^T\\ &= \mathcal{O}_p\mathbf{R}_{xx}\mathcal{O}_p^T + \mathcal{O}_p\mathbf{R}_{xu}\mathbf{R}_{uu}^{-1}\left(\mathbf{R}_{yu}^T - \mathbf{R}_{xu}^T\mathcal{O}_p^T\right)\\ &\quad + \left(\mathbf{R}_{yu} - \mathcal{O}_p\mathbf{R}_{xu}\right)\mathbf{R}_{uu}^{-1}\mathbf{R}_{yu}^T\\ &= \mathcal{O}_p\mathbf{R}_{xx}\mathcal{O}_p^T + \mathcal{O}_p\mathbf{R}_{xu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{yu}^T - \mathcal{O}_p\mathbf{R}_{xu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{xu}^T\mathcal{O}_p^T\\ &\quad + \mathbf{R}_{yu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{yu}^T - \mathcal{O}_p\mathbf{R}_{xu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{yu}^T\\ &= \mathcal{O}_p\mathbf{R}_{xx}\mathcal{O}_p^T - \mathcal{O}_p\mathbf{R}_{xu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{xu}^T\mathcal{O}_p^T + \mathbf{R}_{yu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{yu}^T\end{aligned}$$

Moving all of the terms that can be composed with measured data to the left side:

$$\begin{aligned}\mathbf{R}_{yy} - \mathbf{R}_{yu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{yu}^T &= \mathcal{O}_p\mathbf{R}_{xx}\mathcal{O}_p^T - \mathcal{O}_p\mathbf{R}_{xu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{xu}^T\mathcal{O}_p^T\\ &= \mathcal{O}_p\left(\mathbf{R}_{xx} - \mathbf{R}_{xu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{xu}^T\right)\mathcal{O}_p^T\end{aligned}$$

We make the assumption that all current and future input data is uncorrelated with the current state, which means that the average of the products $\mathbf{x}(k)\mathbf{u}(k+i),\ \ i \in [0, 1, 2, \dots]$ is zero. This gives:

$$\begin{aligned}\mathbf{R}_{xu} &= \frac{1}{N}\begin{bmatrix}\sum_{j=0}^{N-1}\mathbf{x}(k+j)\mathbf{u}(k+j)\\ \sum_{j=0}^{N-1}\mathbf{x}(k+j)\mathbf{u}(k+j+1)\\ \sum_{j=0}^{N-1}\mathbf{x}(k+j)\mathbf{u}(k+j+2)\\ \vdots\\ \sum_{j=0}^{N-1}\mathbf{x}(k+j)\mathbf{u}(k+j+p-1)\end{bmatrix}^T\\ &= \mathbf{0}\end{aligned}$$

in order to yield:

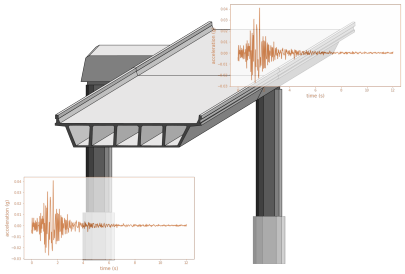
$$\mathbf{R}_{yy} - \mathbf{R}_{yu}\mathbf{R}_{uu}^{-1}\mathbf{R}_{yu}^T = \mathcal{O}_p\mathbf{R}_{xx}\mathcal{O}_p^T\ .$$

CHAPTER

THREE

EXAMPLES

3.1 Overview of the `mdof` package



The `mdof` package is designed to provide a convenient interface for computing dynamic analyses of structural vibrations.

3.1.1 Installation

If you'd like to install `mdof` in your own environment to work with it locally, install mdof from pypi. In your environment manager (e.g., Anaconda),

```
pip install mdof
```

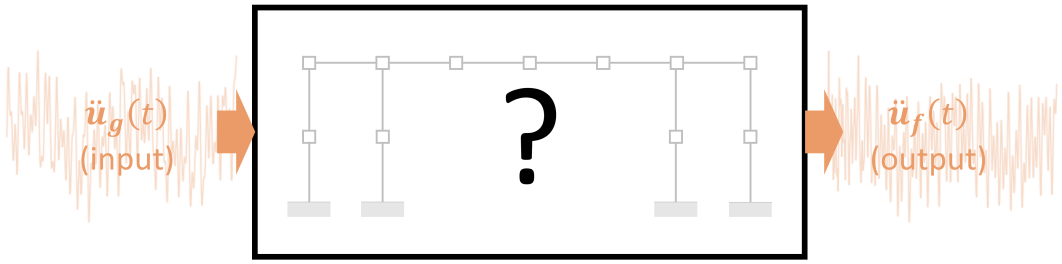
3.1.2 Import

In Python, import the package.

[1]:
```
import mdof
```

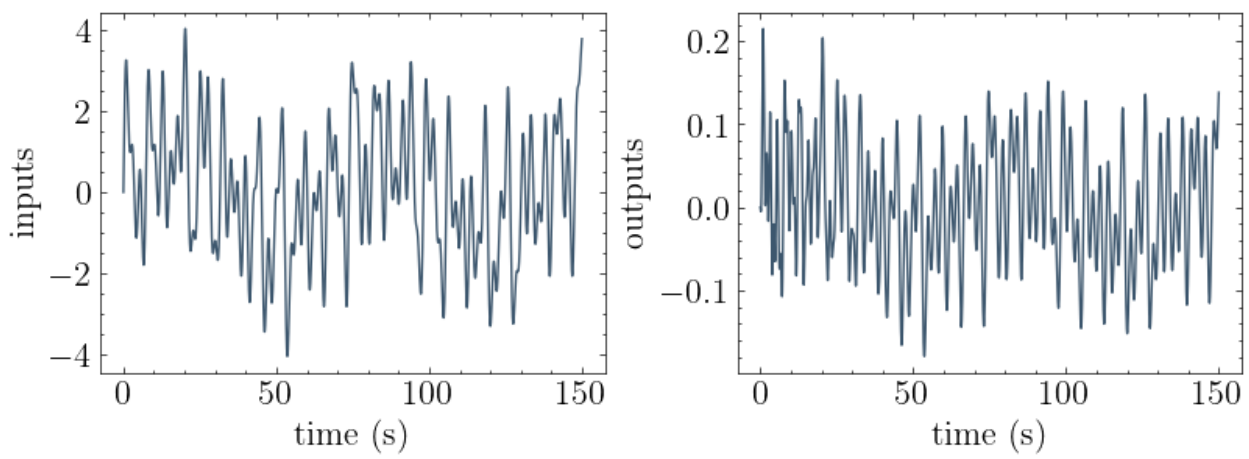
3.1.3 Investigate Structural Vibrations

Load an the input motion, output motion, and time vector for a mystery structural system.

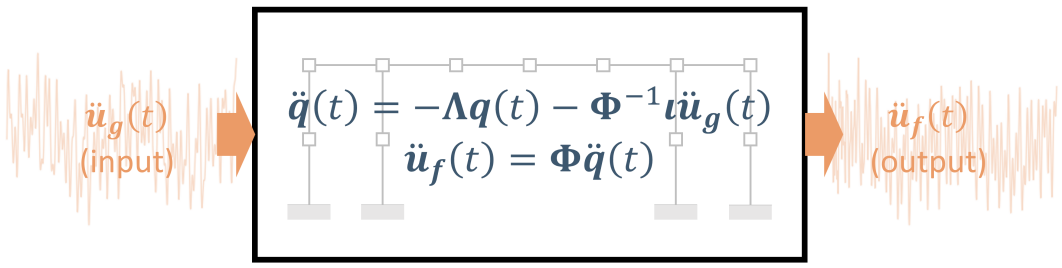


[2]:
```
import numpy as np
input_motion = np.loadtxt("uploads/input_motion.txt")
output_motion = np.loadtxt("uploads/output_motion.txt")
times = np.loadtxt("uploads/times.txt")
```

[3]:
```
from mdof.utilities.printing import plot_io
plot_io(input_motion, output_motion, times)
```



Inverse Eigenanalysis with `mdof.eigid()`

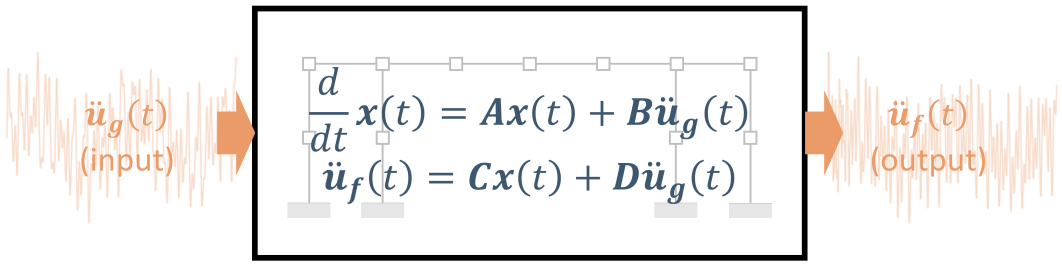


[4]:
```
eigvecs, eigvals = mdof.eigid(input_motion, output_motion, order=2)
```
```
100%|| 4700/4701 [00:00<00:00, 83230.16it/s]
```

[5]:
```
print(f"{eigvecs=}\n {eigvals=}")
```

```
eigvecs=array([0.98491199+0.16330168j, 0.98491199-0.16330168j])
 eigvals=array([[ 0.71332805+0.j        ,  0.71332805-0.j        ],
       [-0.00497201+0.70081265j, -0.00497201-0.70081265j]])
```

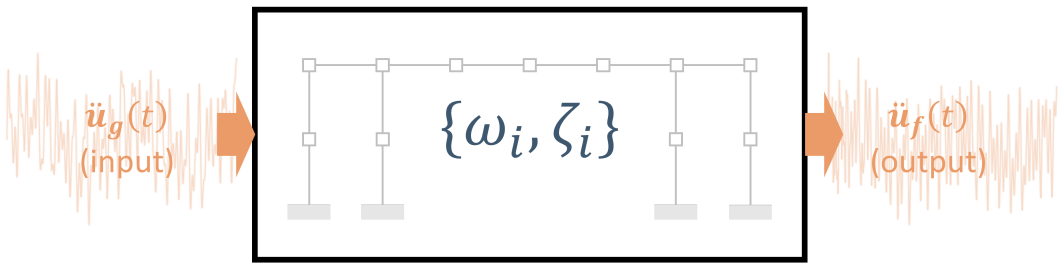
State Space Realization with `mdof.sysid()`



[6]:
```
A,B,C,D = mdof.sysid(input_motion, output_motion, order=2)
```
```
100%|| 4700/4701 [00:00<00:00, 84670.45it/s]
```

[7]:
```
print(f"{A=} \n{B=} \n{C=} \n{D=}")
```
```
A=array([[ 0.98607055,  0.16621799],
       [-0.16044462,  0.98375342]])
B=array([[-0.1743355 ],
       [-0.10978558]])
C=array([[-0.05126065,  0.08870003]])
D=array([[-0.09489982]])
```

Modal Estimation with `mdof.modes()`



[8]:
```
modes = mdof.modes(input_motion, output_motion, dt=times[1]-times[0], order=2)
```
```
100%|| 4700/4701 [00:00<00:00, 89741.83it/s]
```

[9]:
```
from mdof.utilities.printing import print_modes
print_modes(modes)
```
```
Spectral quantities:
       T(s)                EMACO       MPC        EMACO*MPC
```

(continues on next page)

(continued from previous page)

```
      1.147      0.01       1.0        1.0        1.0
Mean Period(s): 1.1471474419090963
Standard Dev(s): 0.0
```

Spectra with `power_transfer()` and `spectrum_modes()`

[10]:
```
from mdof.transform import power_transfer
periods, amplitudes = power_transfer(input_motion, output_motion, step=times[1]-times[0],
↪ period_band=(0,5))
```

[11]:
```
from mdof.modal import spectrum_modes
peak_period, peak_amplitude = spectrum_modes(periods, amplitudes)
print(f"{peak_period=}")
```

```
peak_period=array([1.14503817])
```

[12]:
```
from mdof.utilities.printing import plot_transfer
plot_transfer(np.array([periods,amplitudes]), plotly=True)
```

Stablization Diagram with `stabilization()`

[13]:
```
from mdof.macro import stabilization
stabilization(input_motion, output_motion, dt=times[1]-times[0], orders=(2,20,2),␣
↪plotly=True)
```

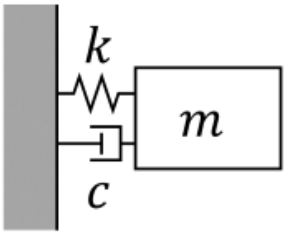
```
100%|| 4700/4701 [00:00<00:00, 103130.17it/s]
100%|| 4700/4701 [00:00<00:00, 43797.93it/s]
100%|| 4700/4701 [00:00<00:00, 23331.09it/s]
100%|| 4700/4701 [00:00<00:00, 18435.29it/s]
100%|| 4700/4701 [00:00<00:00, 13157.71it/s]
100%|| 4700/4701 [00:00<00:00, 9773.15it/s]
100%|| 4700/4701 [00:00<00:00, 6034.75it/s]
100%|| 4700/4701 [00:00<00:00, 4881.76it/s]
100%|| 4700/4701 [00:01<00:00, 3857.66it/s]
```

3.2 Introduction to SISO

[1]:
```
import numpy as np
from mdof.utilities.printing import *
from mdof.utilities.testing import test_method
from mdof.utilities.config import Config
import mdof
from mdof import modal, transform
from control import ss, forced_response
```

3.2.1 Define a SDOF system

parameter	value
m	mass
k	stiffness
c	damping coefficient
nt	number of timesteps
dt	timestep



[2]:
```
# parameters of SDOF system
mass = 1        # mass
k = 30          # stiffness
zeta = 0.01     # damping ratio
omega_n = np.sqrt(k/mass)  # natural frequency (rad/s)
Tn = 2*np.pi/omega_n  # natural periods (s)
c = 2*zeta*mass*omega_n    # damping coefficient
print(f"natural period: {Tn:<3.5}s")
print(f"damping ratio: {zeta}")

# forcing frequencies (rad/s)
omega_f = [0.017*omega_n, 0.14*omega_n, 0.467*omega_n, 0.186*omega_n, 0.2937*omega_n]
```

```
natural period: 1.1471s
damping ratio: 0.01
```

[3]:
```
# forcing function (input)
nt = 5000       # number of timesteps
dt = 0.03       # timestep
tf = nt*dt      # final time
t = np.arange(start = 0, stop = tf, step = dt)
f = np.sum(np.sin([omega*t for omega in omega_f]), axis=0)
```

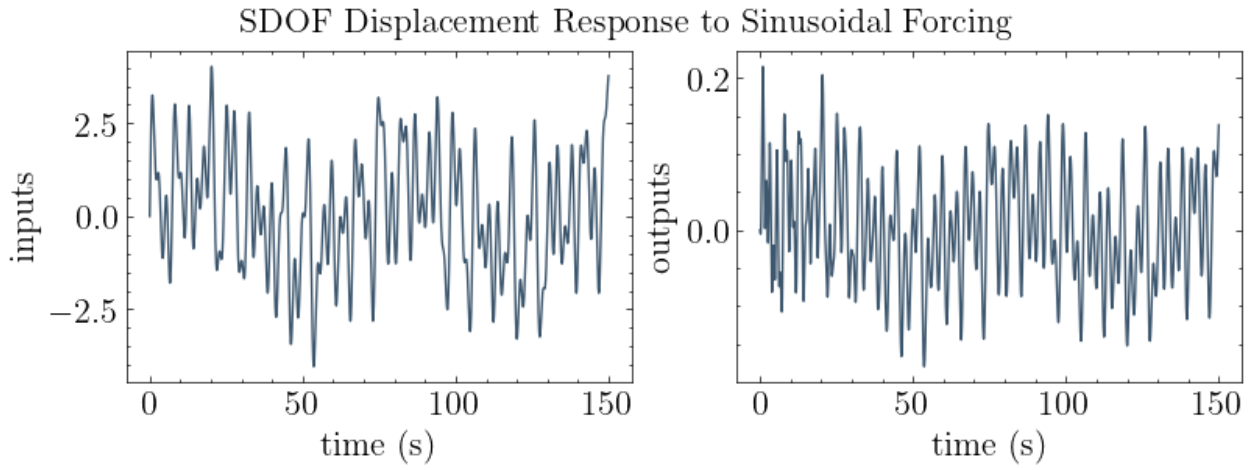
[4]:
```
# displacement response (analytical solution) (output)
omega_D = omega_n*np.sqrt(1-zeta**2)
y = np.zeros((len(omega_f),nt))
for i,omega in enumerate(omega_f):
    C3 = (1/k)*(1-(omega/omega_n)**2)/((1-(omega/omega_n)**2)**2+(2*zeta*omega/omega_
↪n))**2
    C4 = -(2*zeta*omega/omega_n)*(1-(omega/omega_n)**2)/((1-(omega/omega_
↪n)**2)**2+(2*zeta*omega/omega_n))**2
```

(continues on next page)

(continued from previous page)

```
    C1 = -C4
    C2 = (zeta*omega_n*C1-omega*C3)/omega_D
    y[i,:] = np.exp(-zeta*omega_n*t)*(C1*np.cos(omega_D*t)+C2*np.sin(omega_D*t)) + C3*np.
↪sin(omega*t) + C4*np.cos(omega*t)
y = np.sum(y,axis=0)
```

[5]:
```
# plot input vs. output
plot_io(inputs=f, outputs=y, t=t, title="SDOF Displacement Response to Sinusoidal Forcing
↪")
```



3.2.2 Perform System Identification

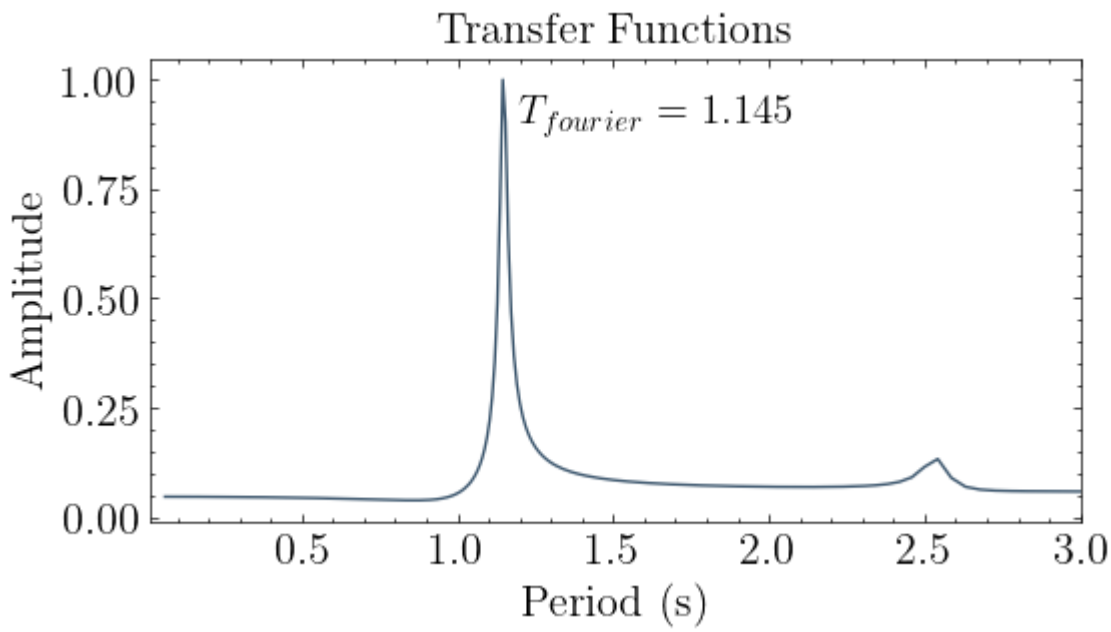
Transfer Function Methods

[6]:
```
# Set parameters
conf = Config()
conf.damping = zeta
conf.period_band = (0.01,3) # Period band (s)

# A place to store models and their predictions
transfer_models = {}

# Generate a transfer function representation of the system
transfer_models["Fourier Transform"] = transform.fourier_transfer(inputs=f, outputs=y,␣
↪step=dt, **conf)
# transfer_models["Response Spectrum"] = transform.response_transfer(inputs=f, outputs=y,
↪ step=dt, pseudo=False, threads=8, periods=(*conf.period_band, 500), **conf)

# Determing the fundamental frequency
fourier_periods, fourier_amplitudes = modal.spectrum_modes(*transfer_models["Fourier␣
↪Transform"])
# response_periods, response_amplitudes = modal.spectrum_modes(*transfer_models[
↪"Response Spectrum"],height=0.1)
plot_transfer(transfer_models, title="Transfer Functions")
color_cycle = plt.rcParams['axes.prop_cycle'].by_key()['color']
```

(continues on next page)

(continued from previous page)

```
# plt.vlines([fourier_periods[0], response_periods[0]],ymin=0,ymax=1,linestyles='--',
↪colors=color_cycle[:2])
plt.text(fourier_periods[0]+0.05,0.9,r"$T_{fourier}$ = "+str(np.round(fourier_periods[0],
↪3)),fontsize=15)
# plt.text(response_periods[0]+0.05,0.8,r"$T_{response}$ = "+str(np.round(response_
↪periods[0],3)),fontsize=15)
plt.xlim(conf.period_band);
# plt.gcf().set_figwidth(9)
# plt.legend();
```



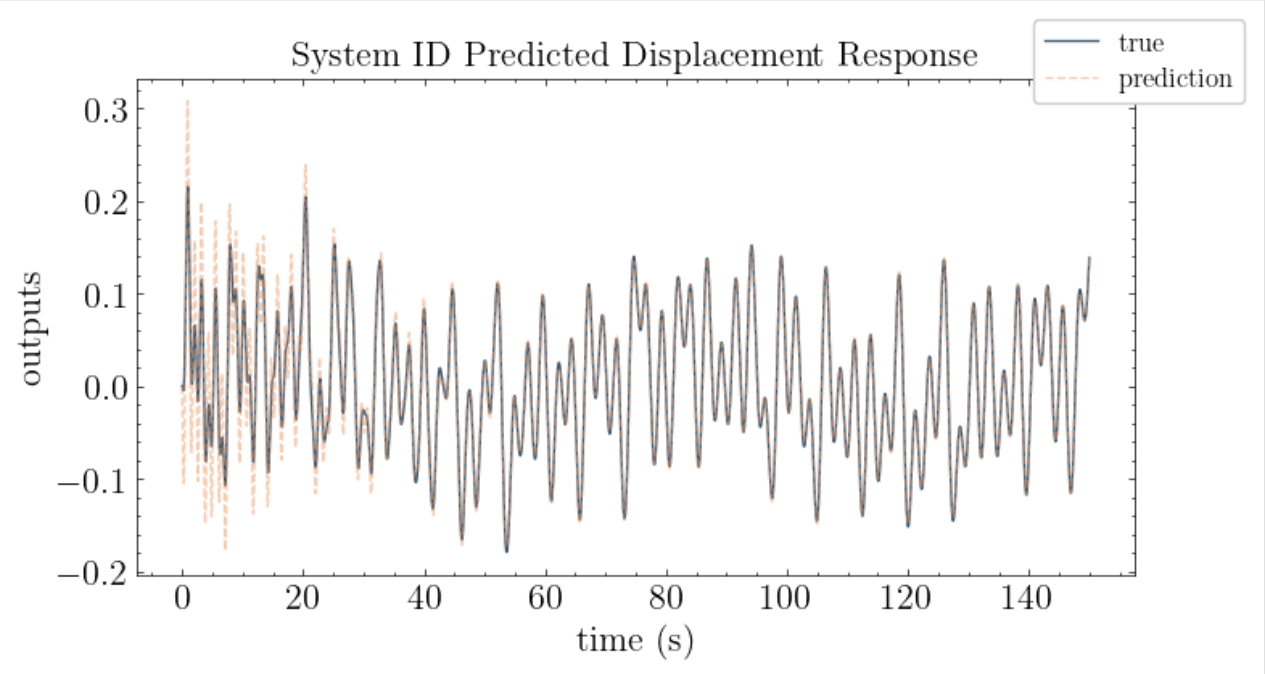
State Space Methods

[7]:
```
# Generate a state space realization of the system
A,B,C,D = mdof.system(inputs=f, outputs=y, r=2)
# Obtain natural period and damping ratio from the state space model
ss_modes = modal.system_modes((A,B,C,D),dt,outlook=190)
print_modes(ss_modes, Tn=Tn, zeta=zeta)
```

```
100%|| 4700/4701 [00:00<00:00, 86808.38it/s]
```

```
Spectral quantities:
       T(s)                EMACO      MPC       EMACO*MPC     T % error     % error
      1.147      0.01       1.0        1.0         1.0          9.678e-14    1.622e-11
Mean Period(s): 1.1471474419090963
Standard Dev(s): 0.0
```

[8]:
```
# Reproduce the response with the state space model
y_mdof = forced_response(ss(A,B,C,D,dt), T=t, U=f, squeeze=False, return_x=False).outputs
plot_pred(ytrue=y, models=y_mdof, t=t, title="System ID Predicted Displacement Response")
```



3.2.3 Breakdown of State Space Methods

General Parameters

parameter	value
`p`	number of output channels
`q`	number of input channels
`nt`	number of timesteps
`dt`	timestep
`decimation`	decimation (downsampling) factor
`order`	model order (2 times number of DOF)

Parameters for Mode Validation

parameter	value
`outlook`	number of steps used for temporal consistency in EMAC

Method Inputs

[9]:
```
# Set parameters
conf = Config()
conf.m  = 1500
conf.horizon = 150
conf.nc = 150
conf.order  =   2
conf.a  =   0
conf.b  =   0
conf.l  =  10
conf.g  =   3

# A place to store models and their predictions
models = {}
```

Specific to Observer Kalman Identification (OKID)

parameter	value
`m`	number of Markov parameters to compute (at most = nt)

Specific to Eigensystem Realization Algorithm (ERA)

parameter	value
`horizon`	number of observability parameters, or prediction horizon
`nc`	number of controllability parameters

OKID-ERA

[10]:
```
# OKID-ERA
method = "okid-era"
models[method] = test_method(method=method, inputs=f, outputs=y, dt=dt, t=t, **conf)
print_modes(models[method]["modes"], Tn=Tn, zeta=zeta)
```

```
Spectral quantities:
       T(s)                EMACO       MPC        EMACO*MPC      T % error      % error
      1.801       0.3442      1.0         1.0          1.0           57.01         3.342e+03
Mean Period(s): 1.8011804589415248
Standard Dev(s): 0.0
```

Specific to Data Correlations (DC)

parameter	value
a	(alpha) number of additional block rows in Hankel matrix of correlation matrices
b	(beta) number of additional block columns in Hankel matrix of correlation matrices
l	initial lag
g	lag (gap) between correlations

OKID-ERA-DC

[11]:
```
# OKID-ERA-DC
method = "okid-era-dc"
models[method] = test_method(method=method, inputs=f, outputs=y, dt=dt, t=t, **conf)
print_modes(models[method]["modes"], Tn=Tn, zeta=zeta)
```

```
Spectral quantities:
      T(s)                EMACO      MPC       EMACO*MPC     T % error     % error
     1.776      0.2423     1.0        1.0         1.0          54.85        2.323e+03
Mean Period(s): 1.7763378693304586
Standard Dev(s): 0.0
```

Specific to System Realization with Information Matrix (SRIM)

parameter	value
horizon	number of steps used for identification, or prediction horizon

SRIM

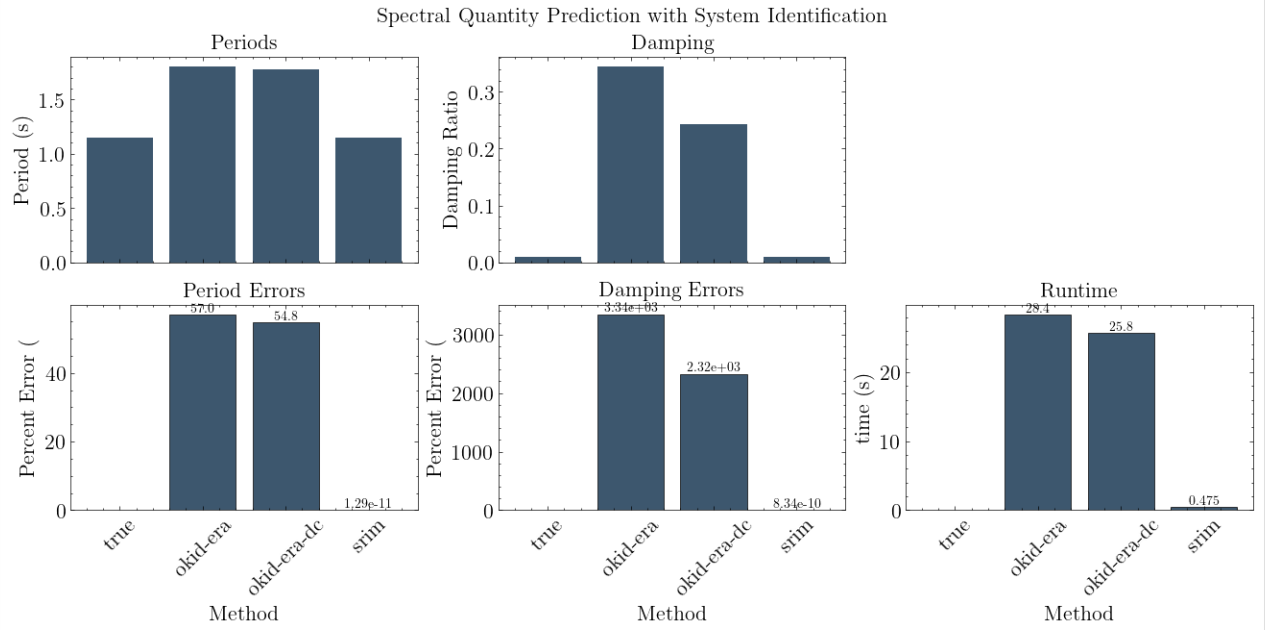
[12]:
```
# SRIM
method = "srim"
models[method] = test_method(method=method, inputs=f, outputs=y, dt=dt, t=t, **conf)
print_modes(models[method]["modes"], Tn=Tn, zeta=zeta)
```

```
100%|| 4850/4851 [00:00<00:00, 80591.63it/s]
```

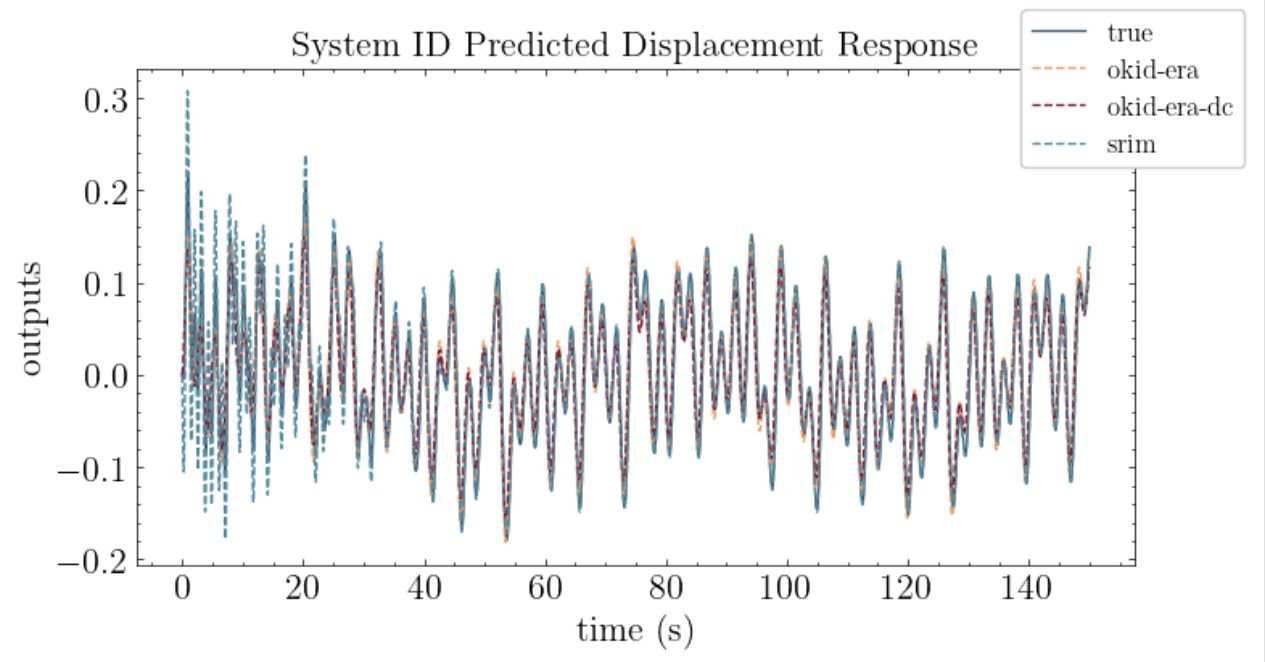
```
Spectral quantities:
      T(s)                EMACO      MPC       EMACO*MPC     T % error     % error
     1.147      0.01       1.0        1.0         1.0          1.295e-11    8.338e-10
Mean Period(s): 1.1471474419092438
Standard Dev(s): 0.0
```

Compare Methods

[13]:
```
plot_models(models, Tn, zeta)
```



[14]:
```
plot_pred(ytrue=y, models=models, t=t, title="System ID Predicted Displacement Response")
```

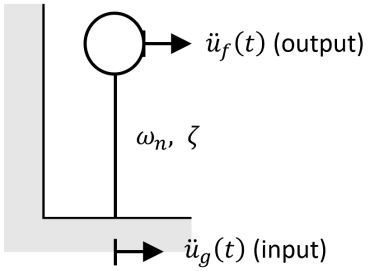


3.3 SISO for a Ground Motion Event

```
[ ]: import mdof
     from mdof import modal, transform
     import sdof
     import numpy as np
     from numpy import linspace, sqrt, pi
     from mdof.utilities.config import Config
     from mdof.utilities.printing import *
     from mdof.numerics import decimate
```

3.3.1 Unknown system with one input and one output

parameter	value
ω_n	natural period
ζ	damping ratio



3.3.2 Configure

Inputs

```
[ ]: inputs  = np.loadtxt("uploads/elcentro.txt")
     dt = 0.02

     # ta = np.arange(0, inputs.size*dt, dt)
     t = linspace(0, (inputs.size-1)*dt, len(inputs))
```

Outputs

[ ]:
```
outputs = None
```

[ ]:
```
# Example SDOF system
mass = 1        # mass
k = 30          # stiffness
zeta = 0.01     # damping ratio
```

Generate output if one was not given:

[ ]:
```
if outputs is None:
    omega_n = sqrt(k/mass)  # natural frequency (rad/s)
    Tn = 2*pi/omega_n  # natural periods (s)
    c = 2*zeta*mass*omega_n    # damping coefficient
    print(f"natural period: {Tn:<3.5}s")
    print(f"damping ratio:  {zeta}")
    displ, veloc, outputs = sdof.integrate(mass,c,k,inputs,dt)
```

```
natural period: 1.1471s
damping ratio:  0.01
```

Method

[ ]:
```
conf = Config()
conf.decimation = 1 # transfer function decimation
```

3.3.3 Analysis with System Identification

Transfer Function Methods

[ ]:
```
# Set parameters
conf.damping = zeta
conf.period_band = (0.1,3) # Period band (s)
conf.pseudo = True # use pseudo accelerations (Sa)

# A place to store models and their predictions
transfer_models = {}

# Generate a transfer function representation of the system
transfer_models["Fourier Transform"] = transform.fourier_transfer(inputs, outputs, dt,␣
↪**conf)
transfer_models["Response Spectrum"] = transform.response_transfer(inputs, outputs, dt,␣
↪**conf)

# Determing the fundamental frequency
fourier_periods, fourier_amplitudes = modal.spectrum_modes(*transfer_models["Fourier␣
↪Transform"])
response_periods, response_amplitudes = modal.spectrum_modes(*transfer_models["Response␣
↪Spectrum"], height=10)
```

State Space Methods

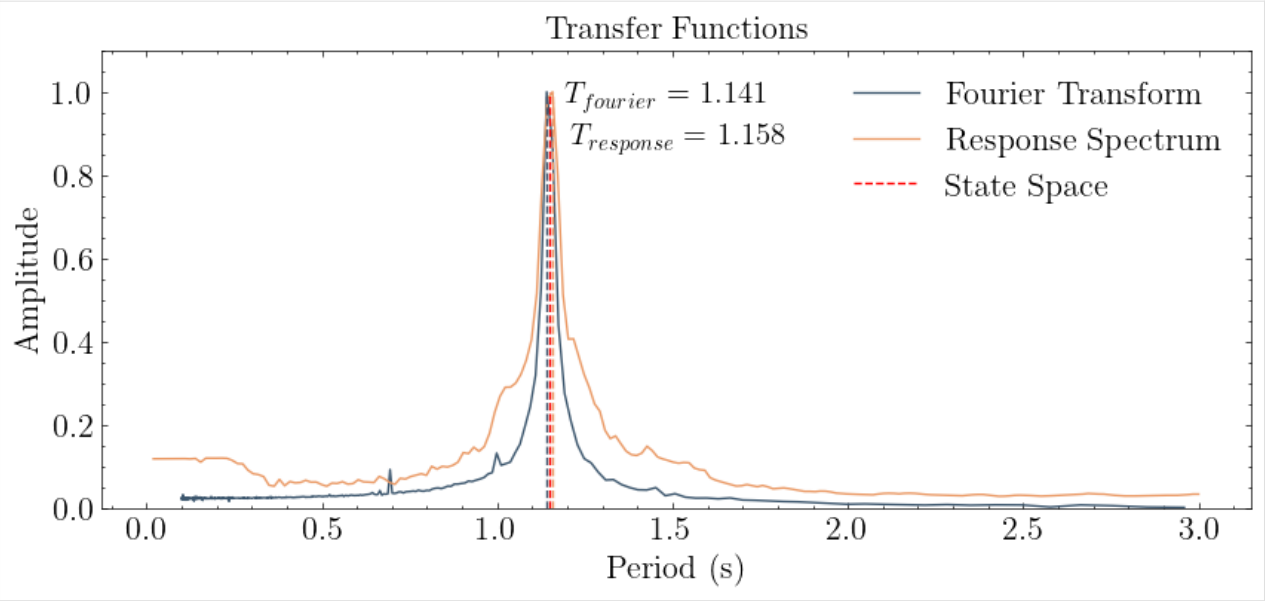
[ ]:
```
# Generate a state space realization of the system
conf.decimation = 1
conf.order = 2
conf.horizon = 300
realization = mdof.system(method="srim", inputs=inputs, outputs=outputs, **conf)
# Obtain natural period and damping ratio from the state space model
ss_modes = modal.system_modes(realization, dt, **conf)
print_modes(ss_modes, Tn=Tn, zeta=zeta)
```

```
100%|| 3695/3696 [00:00<00:00, 28868.69it/s]
```

```
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC          T % error     % error
      1.148      0.00998    1.0        1.0        1.0          0.0999       -0.1995
Mean Period(s): 1.1482934446028523
Standard Dev(s): 0.0
```

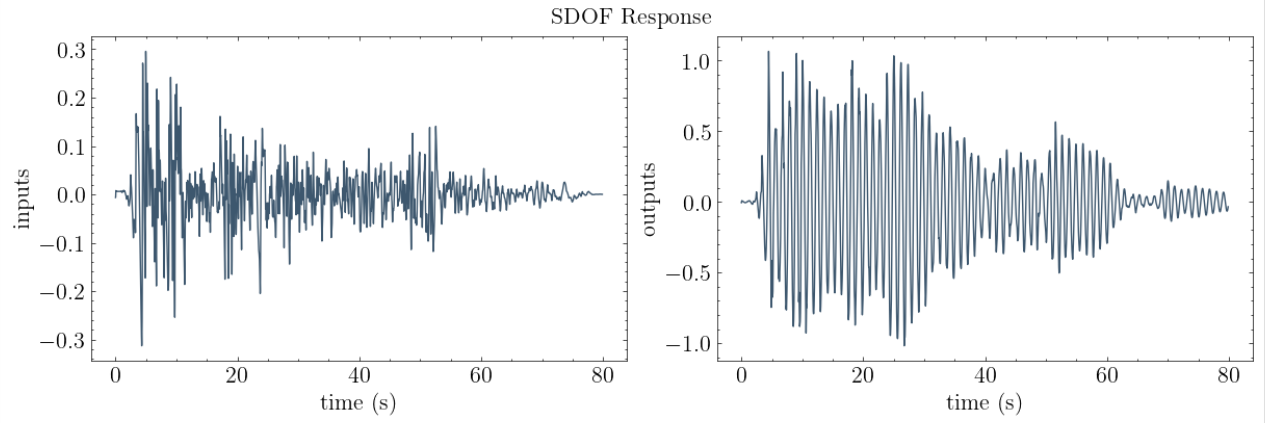
Visualize Transfer Functions vs. State Space Methodds

[ ]:
```
plot_transfer(transfer_models, title="Transfer Functions")
color_cycle = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.gcf().axes[0].vlines([fourier_periods[0], response_periods[0]],ymin=0,ymax=1,
↪linestyles='--',colors=color_cycle[:2])
plt.gcf().axes[0].vlines(1/ss_modes[next(iter(ss_modes.keys()))]["freq"],ymin=0,ymax=1,
↪linestyles='--',colors='r',label="State Space")
plt.gcf().axes[0].legend()
plt.gcf().axes[0].text(fourier_periods[0]+0.05,0.975,r"$T_{fourier}$ = "+str(np.
↪round(fourier_periods[0],3)),fontsize=15)
plt.gcf().axes[0].text(response_periods[0]+0.05,0.875,r"$T_{response}$ = "+str(np.
↪round(response_periods[0],3)),fontsize=15)
plt.gcf().axes[0].set_ylim((0,1.1));

# fig, ax = plt.subplots(figsize=(5,3))
# ax.plot(*transfer_models["Fourier Transform"] )
# plt.tick_params(left = False, right = False , labelleft = False ,
#                 labelbottom = False, bottom = False)
# ax.set_xlabel("Period (s)")
# ax.set_ylabel("Amplitude")
# plt.gcf().savefig("output/transfer.svg");
```



[ ]:
```
assert np.isclose(1/Tn, ss_modes[next(iter(ss_modes.keys()))]["freq"], atol=1e-2), (1/Tn,
↪ 1/ss_modes[next(iter(ss_modes.keys()))]["freq"])
assert np.isclose(Tn, fourier_periods[0], atol=1e-2)
```

[ ]:
```
plot_io(inputs=inputs, outputs=outputs, t=t, title="SDOF Response")
```

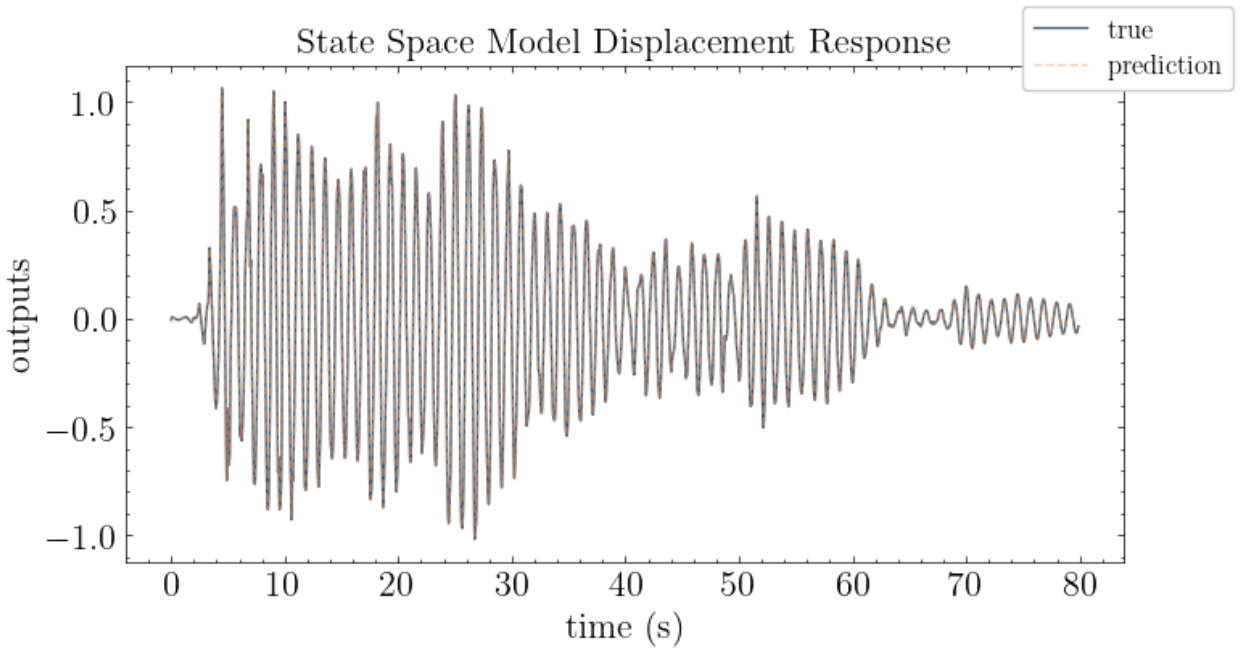


[ ]:
```
# Reproduce the response with the state space model
from control import ss, forced_response
out_mdof = forced_response(ss(*realization,dt*conf.decimation), U=decimate(inputs,conf.
↪decimation), squeeze=False, return_x=False).outputs
plot_pred(ytrue=decimate(outputs,conf.decimation), models=out_mdof, t=decimate(t,conf.
↪decimation), title="State Space Model Displacement Response")

# color_cycle = plt.rcParams['axes.prop_cycle'].by_key()['color']
# fig, ax = plt.subplots(figsize=(5,4))
# ax.step(decimate(t,conf.decimation*16),decimate(outputs,conf.decimation*16),label="true
↪")
# ax.plot(decimate(t,conf.decimation),decimate(out_mdof[0,:],1),"--",label=f"prediction")
# ax.set_xlabel("Time (s)")# , fontsize=13)
```

(continues on next page)

(continued from previous page)

```
# ax.set_ylabel("Output")# , fontsize=13)
# ax.set_xlim((25,45))
# plt.tick_params(left = False, right = False , labelleft = False ,
#                 labelbottom = False, bottom = False);
```



3.4 SISO for Event History

[1]:
```
import numpy as np
from mdof.utilities.printing import *
import mdof
from mdof import modal, transform
from mdof.utilities.config import Config
```

3.4.1 Upload a Dataset and Identify Spectral Quantities

[2]:
```
inputs = np.loadtxt("./uploads/opensees_sdof/Accgrd_set1.csv", delimiter=",")
outputs = np.loadtxt("./uploads/opensees_sdof/Accrsp_set1.csv", delimiter=",")
# inputs = np.loadtxt("./uploads/opensees_sdof/Accgrd_set2.csv", delimiter=",")
# outputs = np.loadtxt("./uploads/opensees_sdof/Accrsp_set2.csv", delimiter=",")
t = inputs[0,:]
dt = t[1] - t[0]
```

Method Inputs

See `01_SISO_Intro <./01_SISO_Intro.ipynb>`__ for parameter definitions

[3]:
```
# Set parameters
conf = Config()
conf.m  = 300
conf.horizon = 140
conf.nc = 140
conf.order  =   2
conf.a  =   0
conf.b  =   0
conf.l  =  10
conf.g  =   3
conf.damping = 0.0001
conf.period_band = (0.1, 3.0)
```

Perform System Identification

[4]:
```
mode_predictions = np.empty((inputs.shape[0]-1, 2))
plt.rc
fig, ax = plt.subplots(figsize=(8,4))
for i,motion in enumerate(inputs[1:,:]):
    conf.decimation = 8         # transfer function decimation
    A,B,C,D = mdof.system(method="srim", inputs=motion, outputs=outputs[i+1,:], **conf)
    ss_modes = modal.system_modes((A,B,C,D),dt,decimation=conf.decimation)
    mode_predictions[i,:] = [[1/v["freq"], v["damp"]] for v in ss_modes.values()][0]  #␣
↪save predicted period and damping
    conf.decimation = 1         # transfer function decimation
    periods, amplitudes = transform.fourier_transfer(inputs=motion, outputs=outputs[i+1,:
↪], step=dt, **conf)
    ax.plot(periods, amplitudes, color="blue", label=["Fourier" if i==0 else None][0])
    periods, amplitudes = transform.response_transfer(inputs=motion, outputs=outputs[i+1,
↪:], step=dt, periods=periods, threads=10, **conf)
    ax.plot(periods, amplitudes, color="green", label=["Response Spectrum" if i==0 else␣
↪None][0])
    ax.vlines([1/v["freq"] for v in ss_modes.values()], 0, max(amplitudes), colors="r",␣
↪linestyles="--", label=["State Space" if i==0 else None][0])
plt.legend()
plt.xlabel("Period (s)")
plt.ylabel("Amplitude")
print("   period(s)  damping\n", mode_predictions)
```

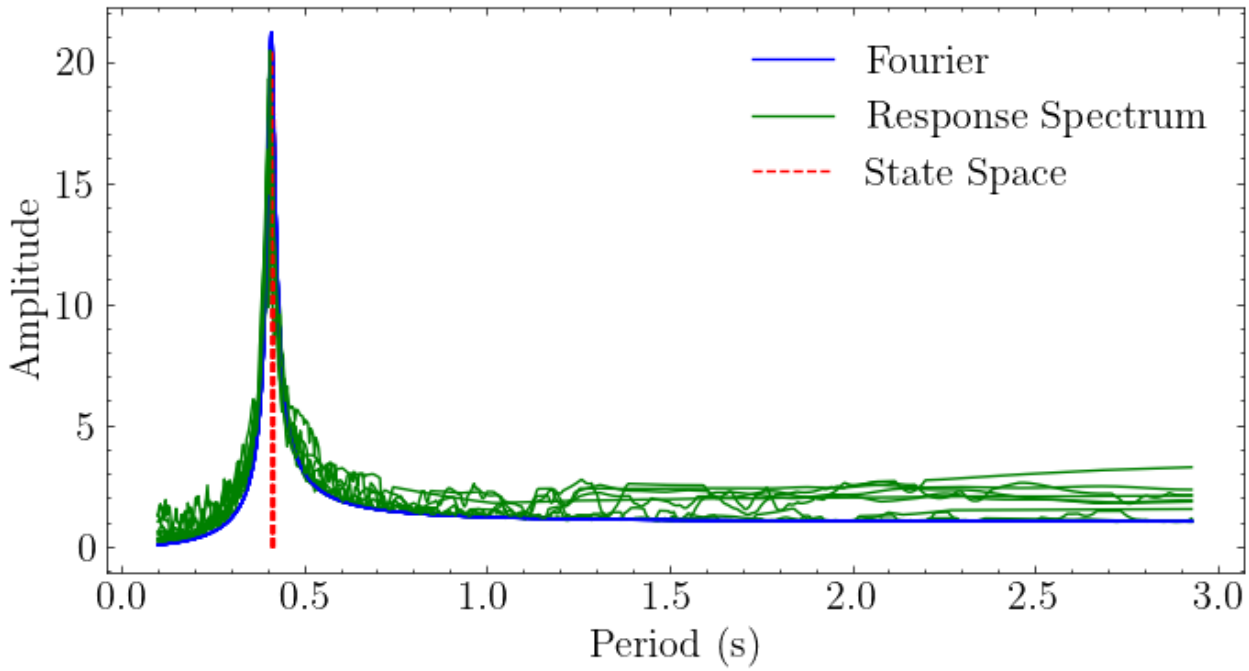
```
100%|| 885/886 [00:00<00:00, 43449.79it/s]
100%|| 885/886 [00:00<00:00, 140370.56it/s]
100%|| 885/886 [00:00<00:00, 524584.38it/s]
100%|| 885/886 [00:00<00:00, 105184.44it/s]
100%|| 885/886 [00:00<00:00, 29318.75it/s]
100%|| 885/886 [00:00<00:00, 120463.39it/s]
100%|| 885/886 [00:00<00:00, 26359.04it/s]
100%|| 885/886 [00:00<00:00, 16509.04it/s]
100%|| 885/886 [00:00<00:00, 21930.00it/s]
```

(continues on next page)

(continued from previous page)

```
100%|| 885/886 [00:00<00:00, 22432.14it/s]
100%|| 885/886 [00:00<00:00, 18971.38it/s]
```

```
  period(s)  damping
[[0.41068031 0.02342356]
 [0.41083068 0.02249289]
 [0.4107063  0.00068529]
 [0.41201371 0.02015356]
 [0.41051848 0.02368659]
 [0.4112227  0.02544649]
 [0.40933146 0.01991049]
 [0.41281326 0.0191038 ]
 [0.40928958 0.02311352]
 [0.41883737 0.0023086 ]
 [0.40943662 0.02034821]]
```

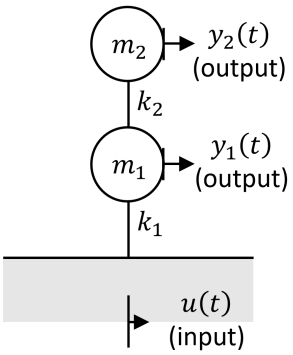


3.5 Introduction to MIMO

[1]:
```
import mdof
from mdof import modal, transform
import sdof
import numpy as np
from numpy import linspace, sqrt, pi
from mdof.utilities.config import Config
from mdof.utilities.printing import *
from mdof.numerics import decimate
```

3.5.1 Unknown multi-input, multi-output system

parameter	value
m_1	mass at dof 1
m_2	mass at dof 2
k_1	stiffness at dof 1
k_2	stiffness at dof 2
ω_1	natural periods for mode 1
ω_2	natural periods for mode 2
ζ_1	modal damping for mode 1
ζ_2	modal damping for mode 2



$$a = \frac{m_1}{m_2}$$

$$b = \frac{k_1}{k_2}$$

$$\mathbf{M} = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix} = m_2 \begin{bmatrix} a & 0 \\ 0 & 1 \end{bmatrix}$$

$$\mathbf{K} = \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix} = k_2 \begin{bmatrix} 1+b & -1 \\ -1 & 1 \end{bmatrix}$$

$$\mathbf{M^{-1}K} = \frac{k_2}{am_2} \begin{bmatrix} 1+b & -1 \\ -a & a \end{bmatrix}$$

$$(1 + b - \lambda)(a - \lambda) + a = 0$$

$$\lambda^2 + (-b - am - 1)\lambda + 2a + ab = 0$$

$$\mathbf{M\ddot{u} + Ku = f(t)}$$

$$\mathbf{\ddot{u} + M^{-1}Ku = M^{-1}f(t)}$$

$$\mathbf{u} = \Phi\mathbf{q}, \qquad \mathbf{M^{-1}K} = \Phi\Lambda\Phi^{-1}$$

$$\mathbf{\ddot{u} + M^{-1}Ku = M^{-1}f(t)}$$

$$\mathbf{\ddot{u}} + \Phi\Lambda\Phi^{-1}\mathbf{u} = \mathbf{M^{-1}f(t)}$$

$$\Phi^{-1}\mathbf{\ddot{u}} + \Phi^{-1}\Phi\Lambda\Phi^{-1}\mathbf{u} = \Phi^{-1}\mathbf{M^{-1}f(t)}$$

$$\mathbf{\ddot{q}} + \Lambda\mathbf{q} = \Phi^{-1}\mathbf{M^{-1}f(t)}$$

[2]:
```
# get modal coordinates
def diag2dof(m1, m2, k1, k2):
  m = m2
  k = k2
  a = m1/m2
  b = k1/k2
  kab1 = k*(a + b + 1)/(2*a*m)
  ksqr = k*sqrt(a**2 - 2*a*b + 2*a + b**2 + 2*b + 1)/(2*a*m)
  eigvals = np.array([kab1 - ksqr,  kab1 + ksqr])
  eigvecs = np.array([[1 - m*(kab1 - ksqr)/k,   1 - m*(kab1 + ksqr)/k],
                      [1, 1]])
  return eigvals, eigvecs

# displacement response (analytical solution) (output)
def harmonic_sdof(forcing_frequencies, nt, t, k, omega_n, zeta):
  omega_D = omega_n*np.sqrt(1-zeta**2)
  output = np.zeros((len(forcing_frequencies),nt))
  for i,omega in enumerate(forcing_frequencies):
    C3 = (1/k)*(1-(omega/omega_n)**2)/((1-(omega/omega_n)**2)**2+(2*zeta*omega/omega_
↪n))**2
    C4 = -(2*zeta*omega/omega_n)*(1-(omega/omega_n)**2)/((1-(omega/omega_
↪n)**2)**2+(2*zeta*omega/omega_n))**2
    C1 = -C4
    C2 = (zeta*omega_n*C1-omega*C3)/omega_D
    output[i,:] = np.exp(-zeta*omega_n*t)*(C1*np.cos(omega_D*t)+C2*np.sin(omega_D*t)) +␣
↪C3*np.sin(omega*t) + C4*np.cos(omega*t)
  output = np.sum(output,axis=0)
  return output

def harmonic_2dof(forcing_frequencies, nt, t, m1, m2, k1, k2, zeta1, zeta2):
  ks = [k1, k2]
  zs = [zeta1, zeta2]
  omega_ns, phis = diag2dof(m1, m2, k1, k2)
  outputs = np.empty((2,nt))
  for i in range(2):
    outputs[i] = harmonic_sdof(forcing_frequencies, nt, t, ks[i], omega_ns[i], zs[i])
  outputs = phis@outputs
  return omega_ns, phis, outputs
```

[3]:
```
# parameters of 2DOF system
m1 = 2          # mass
m2 = 1          # mass
ms = [m1, m2]
k1 = 30         # stiffness
k2 = 10         # stiffness
ks = [k1, k2]
zeta1 = 0.01    # damping ratio
zeta2 = 0.02    # damping ratio
zetas = [zeta1, zeta2]

omega_ns, phis = diag2dof(*ms, *ks)
```

(continues on next page)

(continued from previous page)

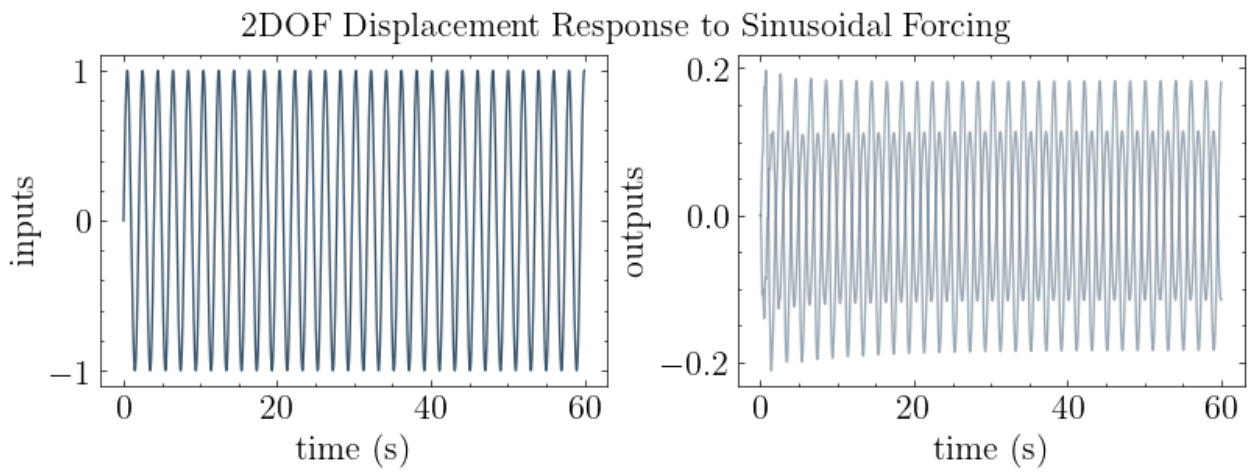
```
C = phis @ np.diag([2*zeta*omega_n for zeta,omega_n in zip(zetas, omega_ns)])   #␣
↪damping coefficients

Tns = [2*np.pi/omega_n for omega_n in omega_ns]
print(f"natural periods: {Tns[0]:<3.3}s, {Tns[1]:<3.3}s")
print(f"damping ratios: {zeta1}, {zeta2}")
# print(f"damping matrix: {C}")
```

```
natural periods: 0.991s, 0.266s
damping ratios: 0.01, 0.02
```

[4]:
```
# forcing function (input)
nt = 2000       # number of timesteps
dt = 0.03       # timestep
tf = nt*dt      # final time
t = np.arange(start = 0, stop = tf, step = dt)  # times
forcing_frequencies = [0.5*omega_ns[0]] # [0.017*omega_ns[0], 0.14*omega_ns[0], 0.
↪467*omega_ns[0], 0.186*omega_ns[0], 0.2937*omega_ns[0]]  # forcing frequencies (rad/s)
inputs = np.sum(np.sin([omega*t for omega in forcing_frequencies]), axis=0)
```

[5]:
```
# displacement response (output)
_, _, outputs = harmonic_2dof(forcing_frequencies, nt, t, *ms, *ks, *zetas)
```

[6]:
```
# plot input vs. output
plot_io(inputs=inputs, outputs=outputs, t=t, title="2DOF Displacement Response to␣
↪Sinusoidal Forcing")
```



3.5.2 Configure

Method

[7]:
```
conf = Config()
conf.order = 4
conf.horizon = 1000
```

3.5.3 Analysis with System Identification

Modal Estimation

[12]:
```
# Set parameters
conf.damping = zetas[0]
conf.period_band = (0.1,3) # Period band (s)
conf.pseudo = False # use pseudo accelerations (Sa)

# A place to store models and their predictions
transfer_models = {}

# Generate a transfer function representation of the system
conf.decimation = 1
transfer_models["Fourier Transform"] = transform.fourier_transfer(inputs, outputs[0], dt,
↪ **conf)
# transfer_models["Response Spectrum"] = transform.response_transfer(inputs, outputs[0],␣
↪dt, **conf)

# Determing the fundamental frequency
fourier_periods, fourier_amplitudes = modal.spectrum_modes(*transfer_models["Fourier␣
↪Transform"])
# response_periods, response_amplitudes = modal.spectrum_modes(*transfer_models[
↪"Response Spectrum"],height=0.15)

# Generate a state space realization of the system
conf.decimation = 1
realization = mdof.system(method="srim", inputs=inputs, outputs=outputs, **conf)
# Obtain natural period and damping ratio from the state space model
ss_modes = modal.system_modes(realization, dt, **conf)
ss_periods = [1/f for f in [ss_modes[mode]["freq"] for mode in ss_modes.keys()]]
print(ss_periods)

plot_transfer(transfer_models, title="Transfer Functions")
color_cycle = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.gcf().axes[0].vlines(ss_periods[:2],0,1,linestyles='--',colors='r',label='State Space
↪')
# plt.gcf().axes[0].vlines([*fourier_periods[:2], *response_periods[:2]],ymin=0,ymax=1,
↪linestyles='--',colors=[color_cycle[0],color_cycle[0],color_cycle[1],color_cycle[1]])
plt.gcf().axes[0].vlines([*fourier_periods[:2]],ymin=0,ymax=1,linestyles='--',
↪colors=color_cycle[0])
for i in range(2):
    plt.gcf().axes[0].text(fourier_periods[i]+0.05,0.975,r"$T_{fourier}$ = "+str(np.
```

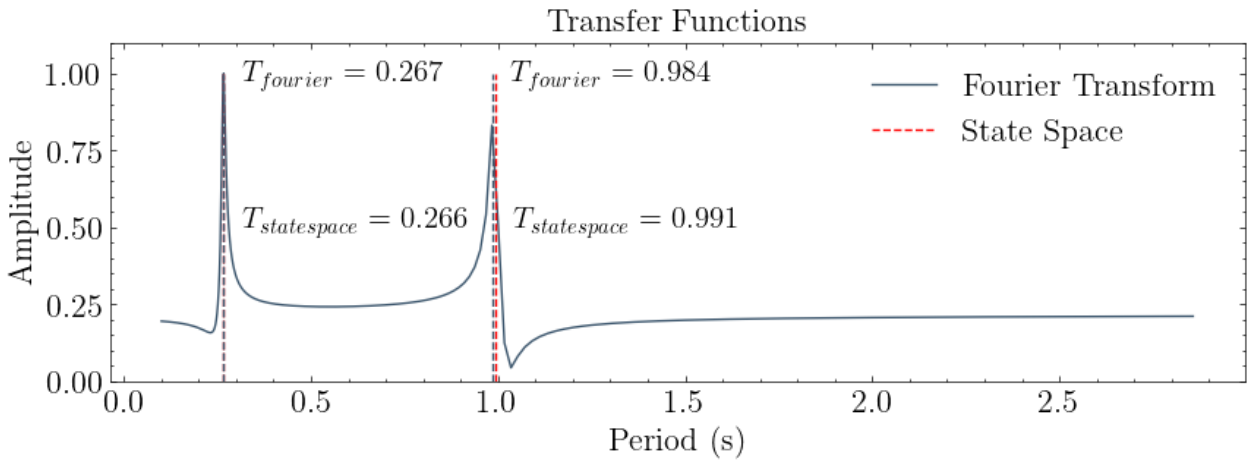
(continues on next page)

(continued from previous page)

```
↪round(fourier_periods[i],3)),fontsize=15)
    plt.gcf().axes[0].text(ss_periods[i]+0.05,0.5,r"$T_{state space}$ = "+str(np.
↪round(ss_periods[i],3)),fontsize=15)
    # plt.gcf().axes[0].text(response_periods[i]+0.05,0.875,r"$T_{response}$ = "+str(np.
↪round(response_periods[i],3)),fontsize=15)
plt.gcf().set_figwidth(10)
plt.gcf().axes[0].legend()
plt.gcf().axes[0].set_ylim((0,1.1));
```

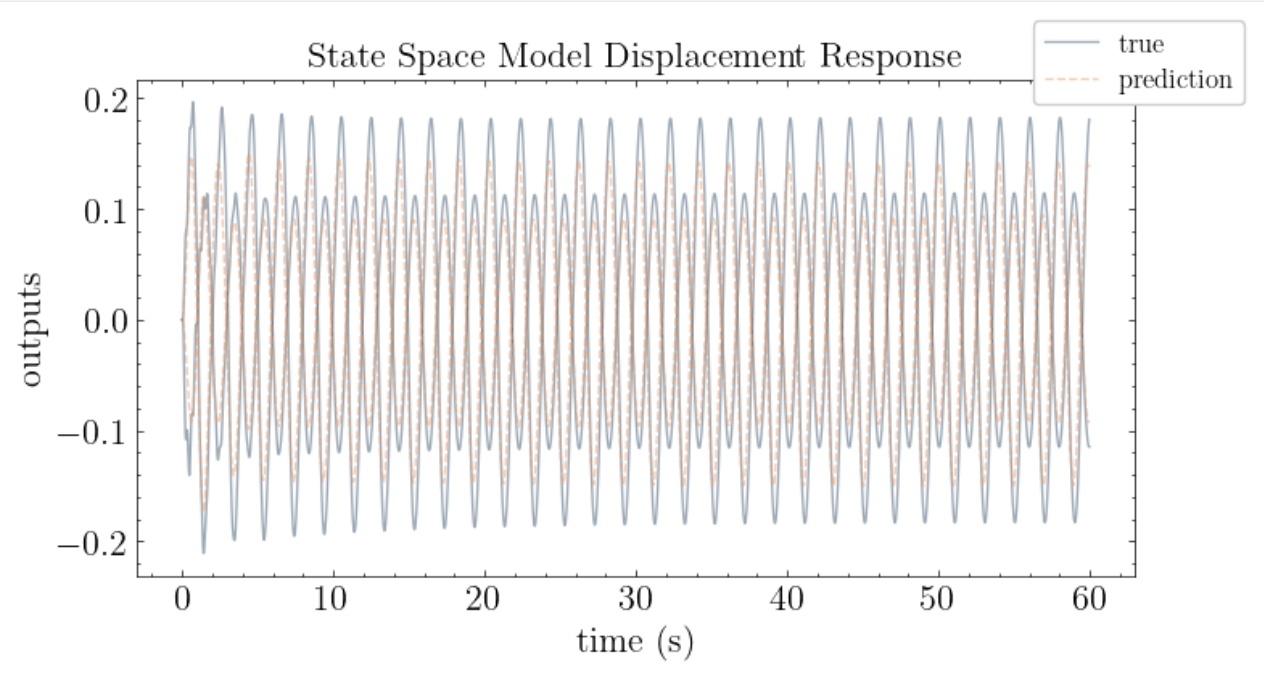
```
100%|| 1200/1201 [00:00<00:00, 15390.62it/s]
```

```
[0.9910784035648029, 0.2655586578710922]
```



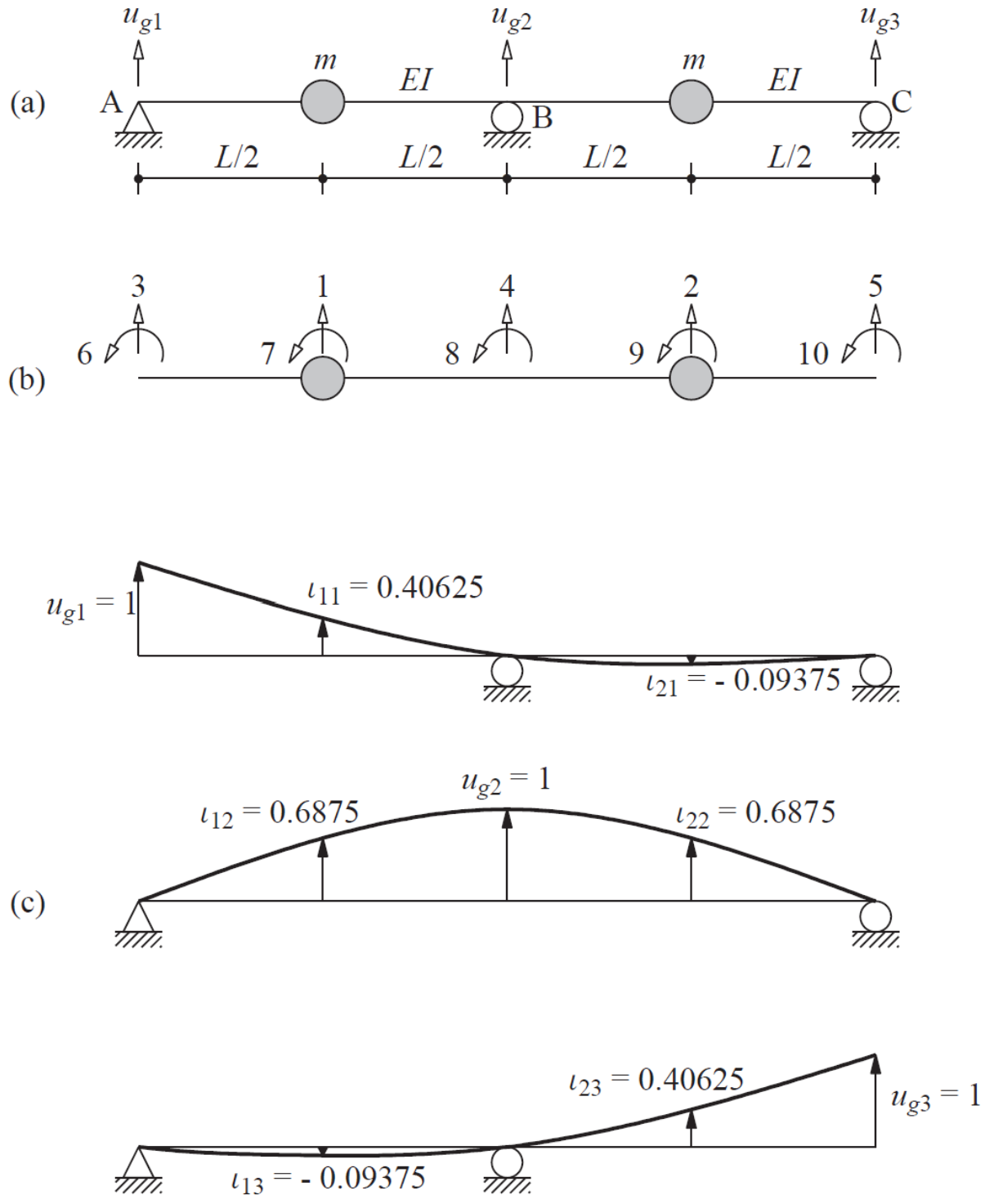
Response Prediction

[16]:
```
# Reproduce the response with the state space model
from control import ss, forced_response
conf.horizon = 800
realization = mdof.system(method="okid-era-dc", inputs=inputs, outputs=outputs, **conf)
out_pred = forced_response(ss(*realization,dt*conf.decimation), U=decimate(inputs,conf.
↪decimation), squeeze=False, return_x=False).outputs
plot_pred(ytrue=decimate(outputs,conf.decimation), models=out_pred, t=decimate(t,conf.
↪decimation), title="State Space Model Displacement Response")
```


State Space Model Displacement Response
true
prediction
outputs
0.2
0.1
0.0
−0.1
−0.2
0
10
20
30
40
50
60
time (s)

3.6 MIMO for a Ground Motion Event

3.6.1 Chopra Section 9.7



Partitioned equation of dynamic equilibrium (Chopra Eq. 9.7.1)

$$\begin{bmatrix} \mathbf{m} & \mathbf{m}_g \\ \mathbf{m}_g^T & \mathbf{m}_{gg} \end{bmatrix} \begin{Bmatrix} \ddot{\mathbf{u}}^t \\ \ddot{\mathbf{u}}_g \end{Bmatrix} + \begin{bmatrix} \mathbf{c} & \mathbf{c}_g \\ \mathbf{c}_g^T & \mathbf{c}_{gg} \end{bmatrix} \begin{Bmatrix} \dot{\mathbf{u}}^t \\ \dot{\mathbf{u}}_g \end{Bmatrix} + \begin{bmatrix} \mathbf{k} & \mathbf{k}_g \\ \mathbf{k}_g^T & \mathbf{k}_{gg} \end{bmatrix} \begin{Bmatrix} \mathbf{u}^t \\ \mathbf{u}_g \end{Bmatrix} = \begin{Bmatrix} \mathbf{0} \\ \mathbf{p}_g(t) \end{Bmatrix}$$

Problem Statement

given: $\mathbf{u}_g$, $\dot{\mathbf{u}}_g$, and $\ddot{\mathbf{u}}_g$

find: $\mathbf{u}^t$ and $\mathbf{p}_g(t)$

–> (system ID: given output $\mathbf{u}^t$ and input $\ddot{\mathbf{u}}_g$, determine fundamental frequencies and mode shapes of the structure.)

General equation of motion for multiple support excitation

1. split displacements into quasi-static ($\mathbf{u}^s$) and dynamic ($\mathbf{u}$) displacements (chopra eq 9.7.2):

$$\begin{Bmatrix} \mathbf{u}^t \\ \mathbf{u}_g \end{Bmatrix} = \begin{Bmatrix} \mathbf{u}^s \\ \mathbf{u}_g \end{Bmatrix} + \begin{Bmatrix} \mathbf{u} \\ \mathbf{0} \end{Bmatrix}$$

2. take the first half of the partitioned equilibrium equation from eq 9.7.1 (chopra eq 9.7.4)

$$\mathbf{m}\ddot{\mathbf{u}}^t + \mathbf{m}_g\ddot{\mathbf{u}}_g + \mathbf{c}\dot{\mathbf{u}}^t + \mathbf{c}_g\dot{\mathbf{u}}_g + \mathbf{k}\mathbf{u}^t + \mathbf{k}_g\mathbf{u}_g = \mathbf{0}$$

3. substitute eq 9.7.2 ($\mathbf{u}^t = \mathbf{u}^s + \mathbf{u}$) and move all $\mathbf{u}_g$ and $\mathbf{u}^s$ terms to the right side (chopra eq 9.7.5):

$$\mathbf{m}\ddot{\mathbf{u}} + \mathbf{c}\dot{\mathbf{u}} + \mathbf{k}\mathbf{u} = \mathbf{p}_{eff}(t)$$

   where

$$\mathbf{p}_{eff}(t) = -(\mathbf{m}\ddot{\mathbf{u}}^s + \mathbf{m}_g\ddot{\mathbf{u}}_g) - (\mathbf{c}\dot{\mathbf{u}}^s + \mathbf{c}_g\dot{\mathbf{u}}_g) - (\mathbf{k}\mathbf{u}^s + \mathbf{k}_g\mathbf{u}_g)$$

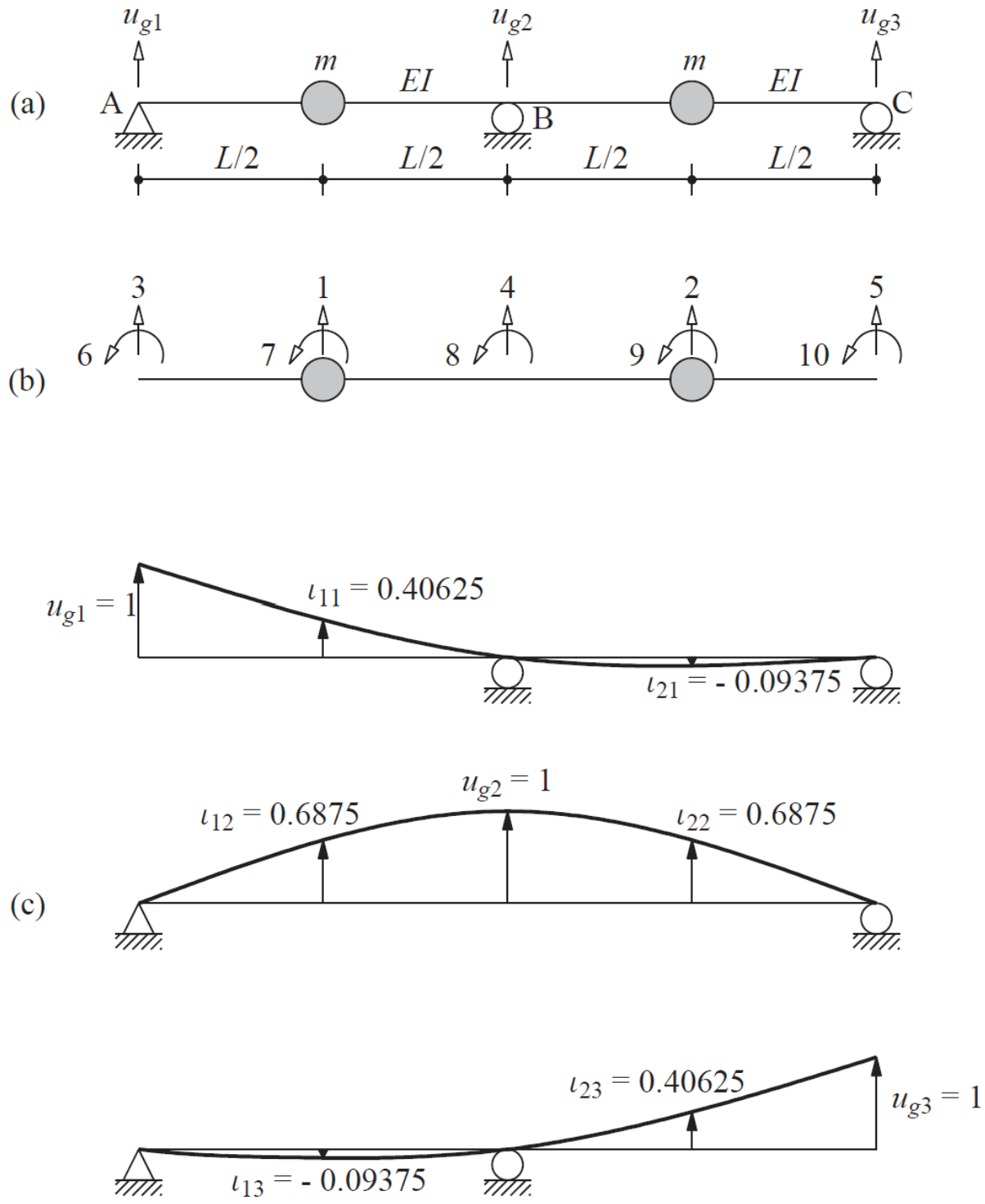
   which simplifies to

$$\mathbf{p}_{eff}(t) = -\mathbf{m}\iota\ddot{\mathbf{u}}_g(t)$$

4. final equation of motion

$$\mathbf{m}\ddot{\mathbf{u}} + \mathbf{c}\dot{\mathbf{u}} + \mathbf{k}\mathbf{u} = -\mathbf{m}\iota\ddot{\mathbf{u}}_g(t)$$

Example 9.10



[1]:
```
import numpy as np
from matplotlib import pyplot as plt

# EI/L^3 = 1
```

(continues on next page)

(continued from previous page)

```
EIL3 = 1
# m = 1
m_node = 1

k_hat = EIL3*np.array(
    [
        [78.86, 30.86, -29.14, -75.43, -5.14],
        [30.86, 78.86, -5.13, -75.43, -29.14],
        [-29.14, -5.14, 12.86, 20.57, 0.86],
        [-75.43, -75.43, 20.57, 109.71, 20.57],
        [-5.14, -29.14, 0.86, 20.57, 12.86]
    ]
)

k = k_hat[:2,:2]
k_g = k_hat[:2,2:]
k_gg = k_hat[2:,2:]

print(f"{k=}, \n{k_g=}, \n{k_gg=}")

m = m_node*np.identity(2)

print(f"{m=}")

iota = -np.linalg.inv(k)@k_g

print(f"{iota=}")
```

```
k=array([[78.86, 30.86],
       [30.86, 78.86]]),
k_g=array([[-29.14, -75.43,  -5.14],
       [ -5.13, -75.43, -29.14]]),
k_gg=array([[ 12.86,  20.57,   0.86],
       [ 20.57, 109.71,  20.57],
       [  0.86,  20.57,  12.86]])
m=array([[1., 0.],
       [0., 1.]])
iota=array([[ 0.40627442,  0.68747721, -0.09378418],
       [-0.09393392,  0.68747721,  0.40621582]])
```

Equation of motion for chopra example 9.10

$$\mathbf{m\ddot{u}} + \mathbf{ku} = -\sum_{l=1}^{3} \mathbf{m}\iota_l \ddot{\mathbf{u}}_{gl}(t)$$

where

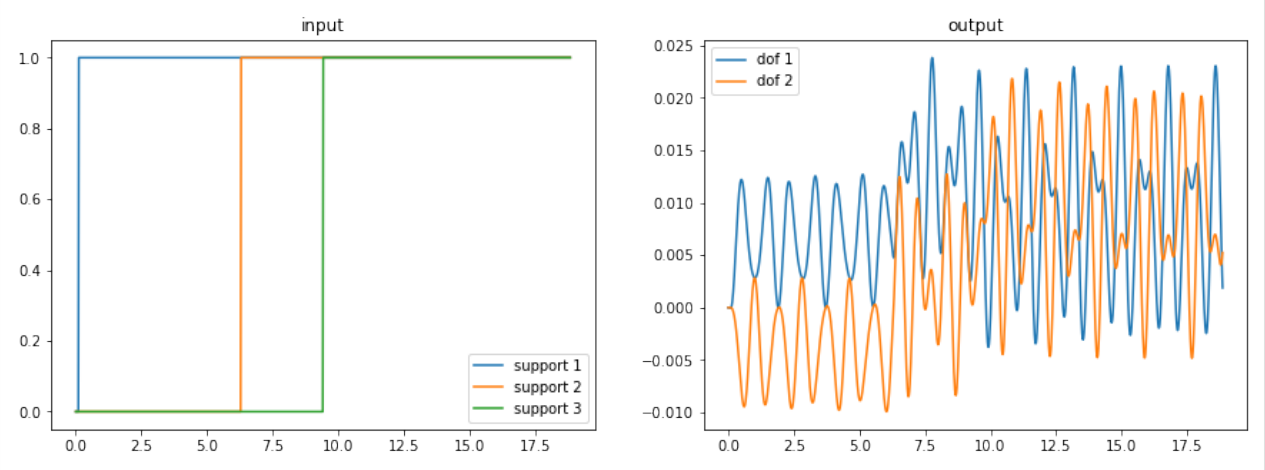
$$m = m \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$k = \frac{EI}{L^3} \begin{bmatrix} 78.86 & 30.86 \\ 30.86 & 78.86 \end{bmatrix}$$

$$\iota_1 = \begin{Bmatrix} 0.40625 \\ -0.09375 \end{Bmatrix} \iota_2 = \begin{Bmatrix} 0.68750 \\ 0.68750 \end{Bmatrix} \iota_1 = \begin{Bmatrix} -0.09375 \\ 0.40625 \end{Bmatrix}$$

Plot example input and output motions for chopra example 9.10

[2]:
```
from matplotlib import pyplot as plt
from scipy import integrate
from numpy import pi
tf = 6*pi
dt = 0.01
ns = 3
nf = 2
t = np.linspace(0., tf, 1000)
input = [
    lambda t: float(t > 0.1),
    lambda t: float(t > 2*pi),
    lambda t: float(t > 3*pi)
    # lambda t: np.sin(t),
    # lambda t: np.sin(t),
    # lambda t: np.sin(t-0.1)
]
# input = np.sin(t) + 20*np.random.rand(nt)
fig, ax = plt.subplots(1,2, figsize=(15,5))
ax[0].plot(t, [input[0](ti) for ti in t], t, [input[1](ti) for ti in t], t,␣
↪[input[2](ti) for ti in t])
ax[0].set_title('input')
ax[0].legend(["support 1", "support 2", "support 3"])
minvk = np.linalg.inv(m)@k
def ex910(y,t,minvk,iota):
    dydt = y
    mipeff = sum(input[i](t)*iota[:,i] for i in range(ns))
    return np.array([
        *y[nf:],
        *(mipeff - minvk@y[:nf])
    ])
output = integrate.odeint(ex910, np.zeros(4), t, args=(minvk,iota))
ax[1].plot(t, output[:,0], t, output[:,1])
ax[1].set_title('output')
ax[1].legend(["dof 1", "dof 2"]);
```



[3]:
```
import scipy
D, V = scipy.linalg.eig(k, m)
print(f"{V=}, \n{D=}")
D, V = np.linalg.eig(minvk)
print(f"{V=}, \n{D=}")
freqs = D**0.5
print(f"{freqs=}")
```

```
V=array([[-0.70710678, -0.70710678],
       [ 0.70710678, -0.70710678]]),
D=array([ 48.  +0.j, 109.72+0.j])
V=array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]]),
D=array([109.72,  48.  ])
freqs=array([10.4747315 ,  6.92820323])
```

The fundamental frequencies of the system are **10.47 rad/s** and **6.93 rad/s.**

- [ ] get fundamental frequencies and mode shapes from equation of motion
- [ ] use system identification to determine fundamental frequencies and mode shapes

[4]:
```
t[1]
```

[4]:
```
0.018868424345884642
```

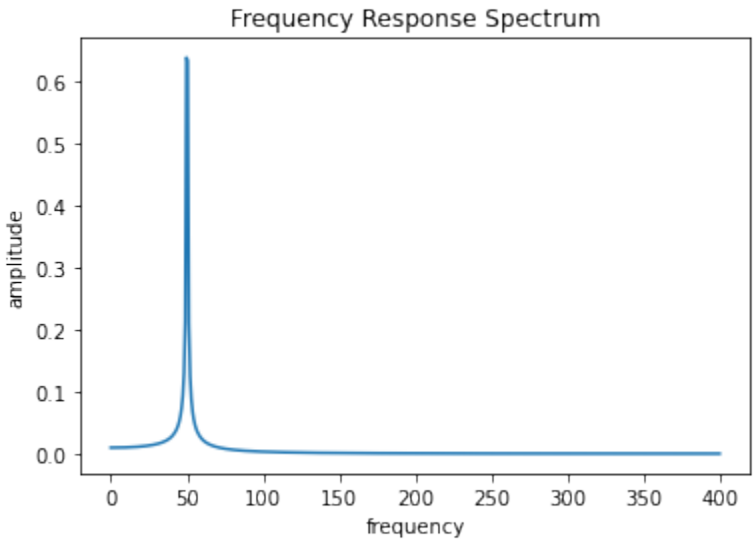
[5]:
```
from scipy.fft import fft, fftfreq
N = 1000
# T = t[1]
# T = 0.0189
T = 1.0/800
x = np.linspace(0.0, N*T, N, endpoint=False)
y = np.sin(50.0*2.0*pi*x)
xf = fftfreq(N, T)[:N//2]
yf = fft(y)
yff = 2.0/N * np.abs(yf[0:N//2])
plt.plot(xf, yff)
plt.xlabel("frequency")
plt.ylabel("amplitude")
```

(continues on next page)

(continued from previous page)

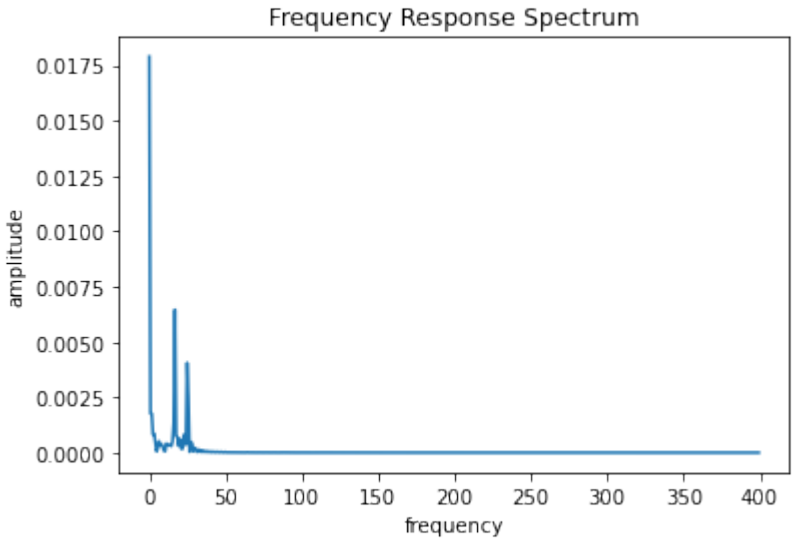
```
plt.title("Frequency Response Spectrum")
freqs_fft = xf[np.argpartition(yff, -3)[-3:]]
print(f"{freqs_fft=}")
```

```
freqs_fft=array([48.8, 50.4, 49.6])
```



[6]:
```
x = t
y = output[:,0]
N = len(x)
# T = x[1]
T = 1/800
xf = fftfreq(N,T)[:N//2]
yf = fft(y)
yff = 2.0/N * np.abs(yf[0:N//2])
plt.plot(xf, yff)
plt.xlabel("frequency")
plt.ylabel("amplitude")
plt.title("Frequency Response Spectrum")
# plt.xlim([0,5])
print([np.argpartition(yff, -5)[-5:]])
freqs_fft = xf[np.argpartition(yff, -5)[-5:]]
print(f"{freqs_fft=}")
```

```
[array([ 1, 21, 31, 32,  0], dtype=int64)]
freqs_fft=array([ 0.8, 16.8, 24.8, 25.6,  0. ])
```



[7]:
```
print(f"{freqs[1]/freqs[0]=}")
print(f"{freqs_fft[1]/freqs_fft[0]=}")
```

```
freqs[1]/freqs[0]=0.6614206034955363
freqs_fft[1]/freqs_fft[0]=21.0
```

[8]:
```
freqs_fft[:2]/freqs/pi

# np.sqrt(2)/2
np.sqrt(3)/2
```

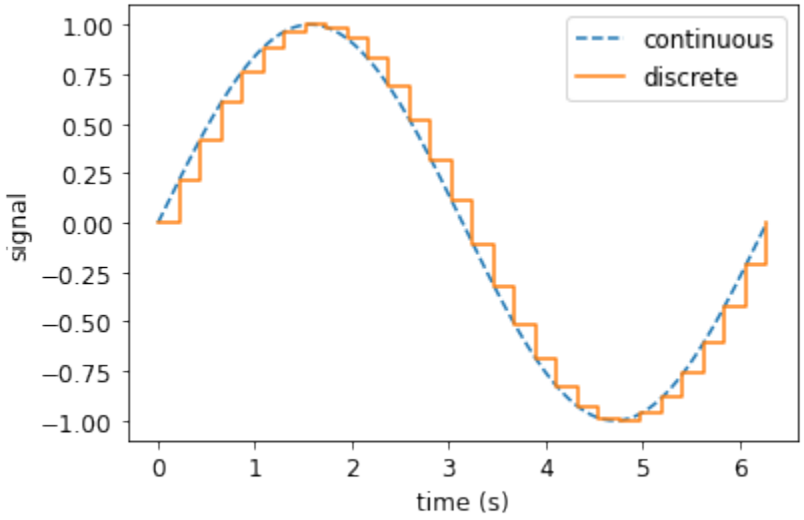
[8]:
```
0.8660254037844386
```

Parameterization of inputs and outputs for OKID-ERA and SRIM

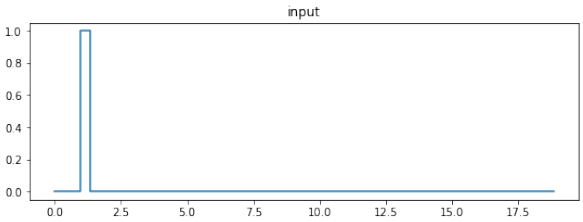
I X O -> SS

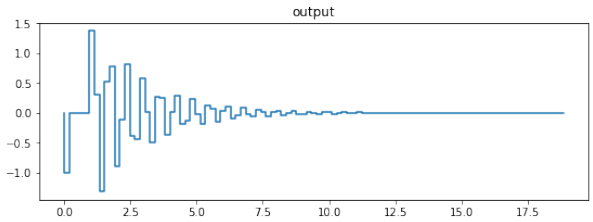
Class SS - coeff: A, B, C, D - obsv: $\mathcal{O}_p$ or V or V_r - ctrl: $\mathcal{C}_p$ or W or W_s - shapes: Φ, Ψ - freq: Ω, Λ

[9]:
```
tf = 2*pi
t = np.linspace(0., tf, 1000)
td = np.linspace(0., tf, 30)

fig, ax = plt.subplots(figsize=(6,4))
ax.plot(t, np.sin(t), linestyle="--", label="continuous")
ax.step(td, np.sin(td), where='post', label="discrete")
ax.set_xlabel("time (s)", size=12)
ax.set_ylabel("signal", size=12)
ax.tick_params(axis='both', which='major', labelsize=12)
ax.legend(fontsize=12);
```

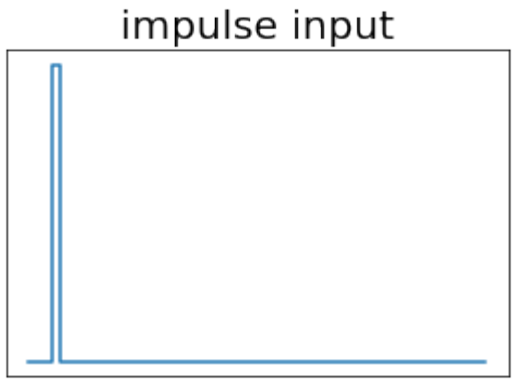


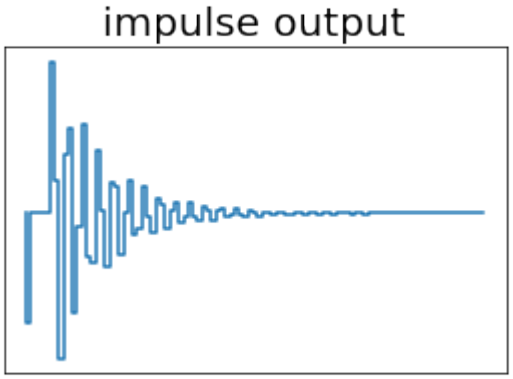
[10]:
```
# SDOF example for impulse response and system id

# Timesteps
tf = 6*pi
nt = 100
t = np.linspace(0., tf, nt)
# Impulse input
input = lambda t: float(pi/3-0.1 < t < pi/3+0.1)

# Mass, stiffness, and damping
m = 1
k = 100
w = np.sqrt(k/m)
c = 0.05*2*m*w

fig, ax = plt.subplots(1,2, figsize=(20,3))
# fig, ax = plt.subplots(1,2, figsize=(10,3.5))
ax[0].step(t, [input(ti) for ti in t], where='post')
ax[0].set_title('input')
def eom(y,t,m,c,k):
    return [y[1], -k*y[0]/m-c*y[1]/m-input(t)]
output = integrate.odeint(eom, [1e-5,0], t, args=(m,c,k))
# ax[1].step(t, output[:,0], where='post')
# ax[1].plot(t, output[:,0], where='post')
ax[1].step(t, [-k*output[i,0]/m-c*output[i,1]/m-input(i) for i in range(nt)])
# ax[1].plot(t, [-k*output[i,0]/m-c*output[i,1]/m-input(i) for i in range(nt)])
ax[1].set_title('output');
# ax[1].plot(t+pi/3-0.1, -1*np.sin(w*(t+pi/3-0.1)));
```





[11]:
```
fig, ax = plt.subplots(figsize=(4.5,3))
ax.step(t, [input(ti) for ti in t], where='post')
ax.set_title('impulse input', fontsize=20)
ax.axes.get_xaxis().set_visible(False)
ax.axes.get_yaxis().set_visible(False)
ax.tick_params(axis='x', labelsize=15)
ax.tick_params(axis='y', labelsize=15)
ax.set_xlabel(r'$t$ (s)', fontsize=20)
ax.set_ylabel(r'$\ddot{u}$ (cm/s$^2$)', fontsize=20);

fig, ax = plt.subplots(figsize=(4.5,3))
ax.step(t, [-k*output[i,0]/m-c*output[i,1]/m-input(i) for i in range(nt)])
ax.set_title('impulse output', fontsize=20)
ax.axes.get_xaxis().set_visible(False)
ax.axes.get_yaxis().set_visible(False)
ax.tick_params(axis='x', labelsize=15)
ax.tick_params(axis='y', labelsize=15)
ax.set_xlabel(r'$t$ (s)', fontsize=20)
ax.set_ylabel(r'$\ddot{u}$ (cm/s$^2$)', fontsize=20);
```





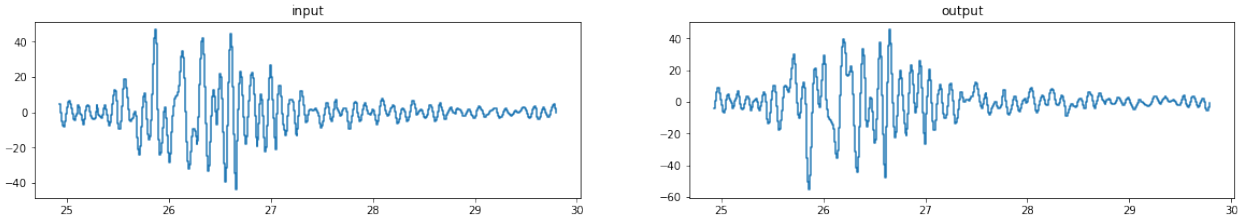
[12]:
```
def husid(accRH, plothusid, dt, lb=0.05, ub=0.95):
    ai = np.tril(np.ones(len(accRH)))@accRH**2
    husid = ai/ai[-1]
    ilb = next(x for x, val in enumerate(husid) if val > lb)
    iub = next(x for x, val in enumerate(husid) if val > ub)
    if plothusid:
        fig, ax = plt.subplots()
        if dt is not None:
            print("duration between ", f"{100*lb}%", " and ", f"{100*ub}%", " (s): ",␣
↪dt*(iub-ilb))
            ax.plot(dt*np.arange(len(accRH)), husid)
            ax.set_xlabel("time (s)")
        else:
            ax.plot(np.arange(len(accRH)), husid)
            ax.set_xlabel("timestep")
        ax.axhline(husid[ilb], linestyle=":", label=f"{100*lb}%")
        ax.axhline(husid[iub], linestyle="--", label=f"{100*ub}%")
        ax.set_title("Husid Plot")
        ax.legend()
    return (ilb, iub)
```

[13]:
```
# San Lorenzo Bent 4 south column base input
from pathlib import Path
import quakeio
event_path = Path("./hayward/58658_003_20210628_18.29.26.P_SanLo.zip")
event = quakeio.read(event_path)
channel = event.match("l", station_channel='25')
response = channel.accel.data
wt = husid(response, False, dt, lb=0.01, ub=0.99)
nt = len(response)
dt = 0.01
nf = dt*nt
t = np.arange(0,nf,dt)
input = lambda t: response[int(t/dt)]

# Mass, stiffness, and damping
m = 1
k = 100
```

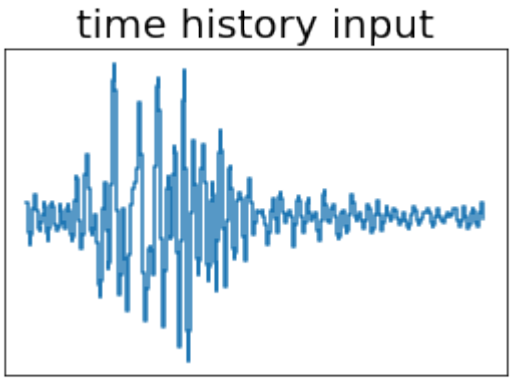
(continues on next page)

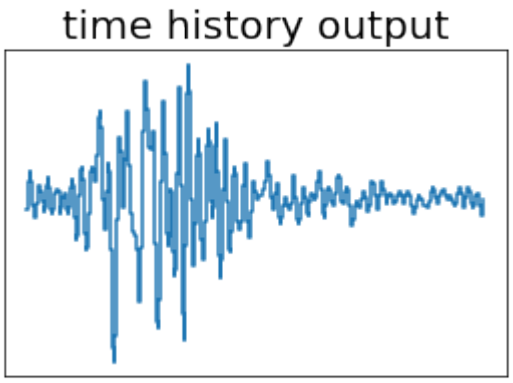
(continued from previous page)

```
w = np.sqrt(k/m)
c = 0.05*2*m*w

# fig, ax = plt.subplots(1,2, figsize=(15,5))
fig, ax = plt.subplots(1,2, figsize=(20,3))
ax[0].step(t[wt[0]:wt[1]], [input(ti) for ti in t][wt[0]:wt[1]], where='post')
ax[0].set_title('input')
def eom(y,t,m,c,k):
    return [y[1], -k*y[0]/m-c*y[1]/m-input(t)]
output = integrate.odeint(eom, [1e-5,0], t, args=(m,c,k))
accel_output = [-k*output[int(ti/dt),0]/m-c*output[int(ti/dt),1]/m-input(ti) for ti in t]
ax[1].step(t[wt[0]:wt[1]], accel_output[wt[0]:wt[1]], where='post')
# markerline, stemlines, baseline = ax[1].stem(t[wt[0]:wt[1]], accel_output[wt[0]:wt[1]],
↪ basefmt=' ', linefmt='-', markerfmt=' ')
# plt.setp(stemlines, 'linewidth', 0.5)
ax[1].set_title('output');
# ax[1].plot(t+pi/3, -0.01*np.sin(w*(t+pi/3)));

# plt.subplots(figsize=(12,3))
# markerline, stemlines, baseline = plt.stem(t[wt[0]:wt[1]], accel_output[wt[0]:wt[1]],␣
↪basefmt=' ')
# plt.setp(stemlines, 'linewidth', 0.5)
# plt.setp(markerline, 'markersize', 2);
```



[14]:
```
fig, ax = plt.subplots(figsize=(4.5,3))
ax.step(t[wt[0]:wt[1]], [input(ti) for ti in t][wt[0]:wt[1]], where='post')
ax.set_title('time history input', fontsize=20)
ax.axes.get_xaxis().set_visible(False)
ax.axes.get_yaxis().set_visible(False)
ax.tick_params(axis='x', labelsize=15)
ax.tick_params(axis='y', labelsize=15)
ax.set_xlabel(r'$t$ (s)', fontsize=20)
ax.set_ylabel(r'$\ddot{u}$ (cm/s$^2$)', fontsize=20);

fig, ax = plt.subplots(figsize=(4.5,3))
ax.step(t[wt[0]:wt[1]], accel_output[wt[0]:wt[1]], where='post')
ax.set_title('time history output', fontsize=20)
ax.axes.get_xaxis().set_visible(False)
ax.axes.get_yaxis().set_visible(False)
ax.tick_params(axis='x', labelsize=15)
ax.tick_params(axis='y', labelsize=15)
ax.set_xlabel(r'$t$ (s)', fontsize=20)
ax.set_ylabel(r'$\ddot{u}$ (cm/s$^2$)', fontsize=20);
```





3.6.2 Continuous vs Discrete

$$\dot{x} = A_c x$$

$$A_c = \Psi \Lambda \Psi^{-1}$$

$$x\left((k+1)\Delta t\right) = A_d x(k\Delta t)$$

$$A_d = e^{A_c \Delta t}$$

$$A_d = \Psi \Gamma \Psi^{-1} = e^{\Psi \Lambda \Psi^{-1} \Delta t} = \Psi e^{\Lambda \Delta t} \Psi^{-1}$$

$$\Gamma = e^{\Lambda \Delta t}$$

$$\lambda_j = \frac{\ln \gamma_j}{\Delta t}$$

[15]:
```
import scipy.linalg as sl

dt = 0.01
tf = 10
nt = int(tf/dt)
t = np.arange(0,tf,dt)
assert len(t) == nt
```

(continues on next page)

(continued from previous page)

```
# Mass, stiffness, and damping
m = 1
k = 100
w = np.sqrt(k/m)
c = 0.02*2*m*w
z = c/(2*m*w)

Ac = np.array([[0, 1],[-k/m, -c/m]])
print(f"{Ac=}")
Ad = sl.expm(Ac*dt)
print(f"{Ad=}")

Gam,Psi = sl.eig(Ad)
Lam,Psic = sl.eig(Ac)

print(f"{Psi=}")
print(f"{Psic=}")
print(f"{Gam=}")
print(f"{np.exp(Lam*dt)=}")
print(f"{Lam=}")
print(f"{np.log(Gam)/dt=}")

lams = np.log(Gam)/dt
omegas = np.sqrt(lams*lams.conj())
zetas = -np.real(lams/omegas)
print(f"{omegas=}")
print(f"{zetas=}")
print(f"{w=}")
```

```
Ac=array([[   0. ,    1. ],
       [-100. ,   -0.4]])
Ad=array([[ 0.99501082,  0.0099634 ],
       [-0.99634016,  0.99102546]])
Psi=array([[-0.00199007-0.09948382j, -0.00199007+0.09948382j],
       [ 0.99503719+0.j        ,  0.99503719-0.j        ]])
Psic=array([[-0.00199007-0.09948382j, -0.00199007+0.09948382j],
       [ 0.99503719+0.j        ,  0.99503719-0.j        ]])
Gam=array([0.99301814+0.09961409j, 0.99301814-0.09961409j])
np.exp(Lam*dt)=array([0.99301814+0.09961409j, 0.99301814-0.09961409j])
Lam=array([-0.2+9.9979998j, -0.2-9.9979998j])
np.log(Gam)/dt=array([-0.2+9.9979998j, -0.2-9.9979998j])
omegas=array([10.+0.j, 10.+0.j])
zetas=array([0.02, 0.02])
w=10.0
```

3.6.3 Trajectories

$$\dot{x} = A_c x$$

$$A_d = e^{A_c \Delta t}$$

$$x\left((k+1)\Delta t\right) = A_d x(k\Delta t) = e^{A_c \Delta t} x(k\Delta t)$$

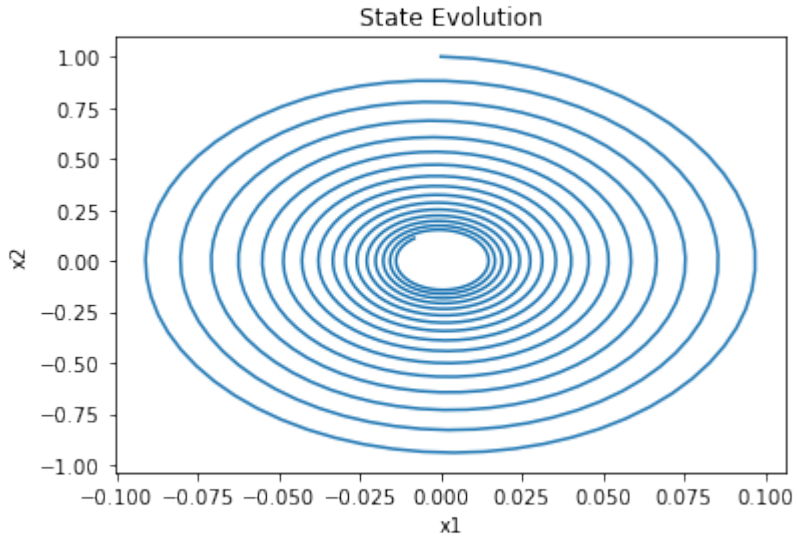
[16]:
```
A1 = np.array([[1,0.1],[-0.1,1]])
A2 = np.array([[0,0.5],[-0.5,0]])
A3 = np.array([[0,1],[-1,-0.5]])

from sympy.matrices import Matrix
display(Matrix(Ac))
```

$$\begin{bmatrix} 0 & 1.0 \\ -100.0 & -0.4 \end{bmatrix}$$

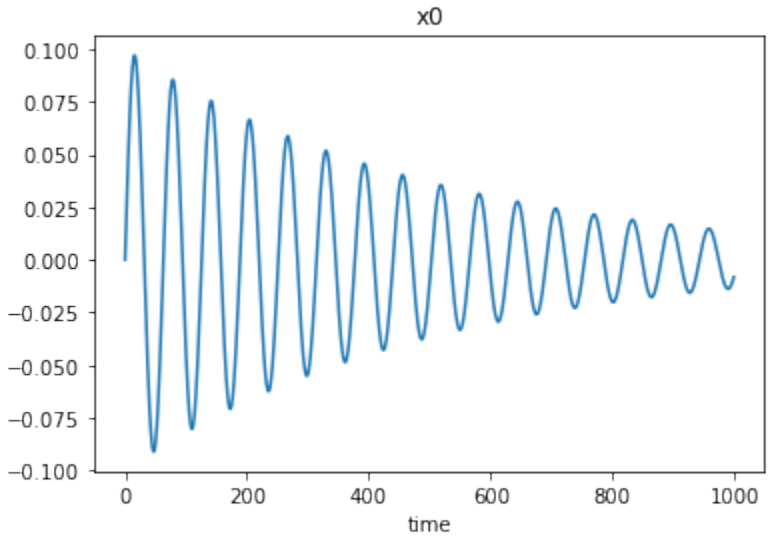
[17]:
```
x0 = np.array([0,1])
xs = np.zeros((2,nt))
xs[:,0] = x0
for i in range(1,nt):
    # xs[:,i] = A3@xs[:,i-1]
    xs[:,i] = Ad@xs[:,i-1]
    # xs[:,i] = xs[:,i-1]+Ac@xs[:,i-1]*dt
```

$$x_{k+1} = e^{A_c \Delta t} x_k$$

[18]:
```
plt.plot(xs[0], xs[1])
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('State Evolution');
```

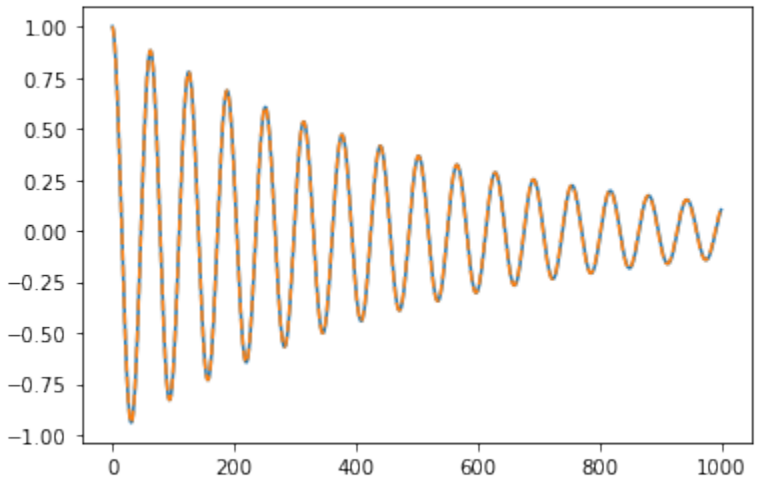


[19]:
```
plt.plot(xs[0])
plt.xlabel('time')
plt.title('x0');
```

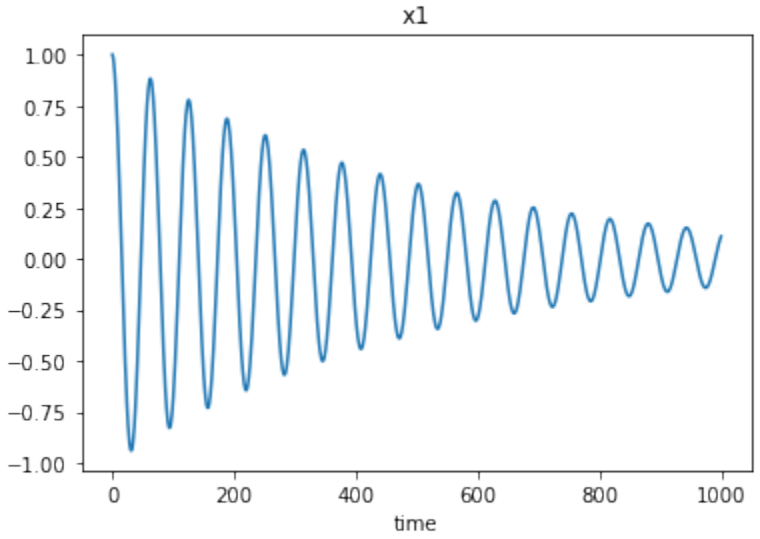


[20]:
```
import sdof
u,v,a = sdof.integrate(m,c,k, np.zeros(nt), dt, v0=1)
plt.plot(v)
plt.plot(xs[1],"--")
```

[20]:
```
[<matplotlib.lines.Line2D at 0x20b087f5a00>]
```



[21]:
```
plt.plot(xs[1])
plt.xlabel('time')
plt.title('x1');
```



3.6.4 ERA

[22]:
```
from sympy.matrices import Matrix
from sympy import *
k, m, w, dt, tau = symbols(r'k,m,\omega_{n},{\Delta}t,\tau')
Ac = Matrix(np.array([[0,1],[-w**2,0]]))
Bc = Matrix(np.array([[0],[-1]]))
C = Matrix(np.array([[-w**2,0]]))
D = Matrix(np.array([[-1]]))

display("Ac=", Ac)
```

```
'Ac='
```

$$\begin{bmatrix} 0 & 1 \\ -\omega_n^2 & 0 \end{bmatrix}$$

[23]:
```
display(Ac.diagonalize()[0])
```

$$\begin{bmatrix} \frac{i}{\omega_n} & -\frac{i}{\omega_n} \\ 1 & 1 \end{bmatrix}$$

[24]:
```
display(Ac.diagonalize()[1])
```

$$\begin{bmatrix} -i\omega_n & 0 \\ 0 & i\omega_n \end{bmatrix}$$

[25]:
```
i = np.round((-1)**0.5)
Lam = Matrix(np.array([[i*w, 0],[0,  -i*w]]))
Psi = Matrix(np.array([[1, 1],[i*w, -i*w]]))
```

[26]:
```
Psi@Lam@Psi.inv()
```

[26]:
$$\begin{bmatrix} 0 & 1.0 \\ -1.0\omega_n^2 & 0 \end{bmatrix}$$

[27]:
```
A = Psi@exp(Lam*dt)@Psi.inv()
```

[28]:
```
A = simplify(exp(Ac*dt))
```

[29]:
```
simplify(A)
```

[29]:
$$\begin{bmatrix} \cos\left(\omega_n \Delta t\right) & \frac{\sin\left(\omega_n \Delta t\right)}{\omega_n} \\ -\omega_n \sin\left(\omega_n \Delta t\right) & \cos\left(\omega_n \Delta t\right) \end{bmatrix}$$

[30]:
```
Bintegrand = simplify(exp(Ac*tau)@Bc)
Bintegrand
```

[30]:
$$\begin{bmatrix} -\frac{\sin\left(\omega_n \tau\right)}{\omega_n} \\ -\cos\left(\omega_n \tau\right) \end{bmatrix}$$

[31]:
```
B = integrate(Bintegrand,(tau,0,dt))
B
```

[31]:
$$\begin{bmatrix} \begin{cases} \frac{\cos\left(\omega_n \Delta t\right)}{\omega_n^2} - \frac{1}{\omega_n^2} & \text{for } \omega_n > -\infty \wedge \omega_n < \infty \wedge \omega_n \neq 0 \\ 0 & \text{otherwise} \end{cases} \\ \begin{cases} -\frac{\sin\left(\omega_n \Delta t\right)}{\omega_n} & \text{for } \omega_n > -\infty \wedge \omega_n < \infty \wedge \omega_n \neq 0 \\ -\Delta t & \text{otherwise} \end{cases} \end{bmatrix}$$

[32]:
```
simplify(A.inv()@B)
```

[32]:
$$\begin{bmatrix} \begin{cases} \frac{1-\cos\left(\omega_n \Delta t\right)}{\omega_n^2} & \text{for } \omega_n > -\infty \wedge \omega_n < \infty \wedge \omega_n \neq 0 \\ \frac{\Delta t \sin\left(\omega_n \Delta t\right)}{\omega_n} & \text{otherwise} \end{cases} \\ \begin{cases} -\frac{\sin\left(\omega_n \Delta t\right)}{\omega_n} & \text{for } \omega_n > -\infty \wedge \omega_n < \infty \wedge \omega_n \neq 0 \\ -\Delta t \cos\left(\omega_n \Delta t\right) & \text{otherwise} \end{cases} \end{bmatrix}$$

3.7 MIMO for Event History

[1]:
```
import numpy as np
import mdof
import quakeio
from mdof import modal, transform
from mdof.utilities import Config, extract_channels, list_files, print_modes, mode_
↪statistics
```

3.7.1 Data inputs

[2]:
```
directory = "../CESMD/CE89324/"
pattern  = "????????*.[zZ][iI][pP]"
# pattern = "sanlorenzo_28june2021.zip"
# pattern = "*[sb][ae][nr][lk]*"

with open('../../Caltrans.Hayward/CGS_data/function_test_zips.txt', 'r') as readfile:
    function_tests = readfile.read().split("\n")
import glob
events = [quakeio.read(event) for event in glob.glob('../CGS_data/58658*P.zip') if␣
↪event[-33:] not in function_tests]

from pathlib import Path
# for file in Path(directory).glob(pattern):
#     if str(file)[-33:] in function_tests:
#         print(file)
files = [file for file in Path(directory).glob(pattern) if str(file)[-33:] not in␣
↪function_tests]
```

[3]:
```
data_conf = Config()

# PAINTER RIO DELL TRANSVERSE (CASE 1)
title = "Painter St Bridge Transverse Mode (In: Ch17, Out: Ch9)"
data_conf.inputs  = [3,17,20]
data_conf.outputs = [7,9,4]
# # PAINTER RIO DELL LONGITUDINAL (CASE 2)
# data_conf.inputs  = [15,1,18]
# data_conf.outputs = [11]

# # HWY8/MELOLAND TRANSVERSE (CASE 1)
# data_conf.inputs  = [2],
# data_conf.outputs = [5,7,9]
# # HWY8/MELOLAND TRANSVERSE (CASE 2)
# data_conf.inputs  = [11,2,26],
# data_conf.outputs = [5,7,9]
# # HWY8/MELOLAND LONGITUDINAL (CASE 3)
# data_conf.inputs  = [12,4,25],
# data_conf.outputs = [27,8]

# # CROWLEY TRANSVERSE (CASE 1)
# data_conf.inputs = [4]
# data_conf.outputs = [6,7,9]
# # CROWLEY TRANSVERSE (CASE 2)
# data_conf.inputs = [6,4,9]
# data_conf.outputs = [7]
# # CROWLEY TRANSVERSE (CASE 3)
# data_conf.inputs = [4]
# data_conf.outputs = [7]
# # CROWLEY LONGITUDINAL (CASE 4)
# data_conf.inputs = [5]
# data_conf.outputs = [8]
```

(continues on next page)

(continued from previous page)

```
# # RIDGECREST TRANSVERSE (CASE 1)
# data_conf.inputs  = [4]
# data_conf.outputs = [6,7,9]

# # CAPISTRANO TRANSVERSE (CASE 1)
# data_conf.inputs  = [4]
# data_conf.outputs = [10,7]

# # HAYWARD TRANSVERSE (CASE 1)
# title = "Hayward Bridge Transverse Mode (In: Ch25, Out: Ch23)"
# data_conf.inputs  = [25,2,7,18]
# data_conf.outputs = [23,13,15,20]
# # HAYWARD LONGITUDINAL (CASE 2)
# title = "Hayward Bridge Longitudinal Mode (In: Ch3, Out: Ch12)"
# data_conf.inputs  = [3,6,17]
# data_conf.outputs = [12,14,19]
# # HAYWARD LONGITUDINAL Alternative
# title = "Hayward Bridge Longitudinal Mode (In: Ch6, Out: Ch14)"
# data_conf.inputs  = [6,3,17]
# data_conf.outputs = [14,12,19]

# # BERNARDINO TRANSVERSE BENT 3 (CASE 1)
# data_conf.inputs  = [6]
# data_conf.outputs = [7,8]
# # BERNARDINO TRANSVERSE BENT 8 (CASE 2)
# data_conf.inputs  = [24]
# data_conf.outputs = [19,20]
# # BERNARDINO LONGITUDINAL BENT 3 (CASE 3)
# data_conf.inputs  = [4]
# data_conf.outputs = [10]
# # BERNARDINO LONGITUDINAL BENT 8 (CASE 4)
# data_conf.inputs  = [22]
# data_conf.outputs = [17,18]

# # VINCENT THOMAS (CE14406) TRANSVERSE (CASE 1)
# data_conf.inputs  = [1,9,24]
# data_conf.outputs = [2,5,7]
# # VINCENT THOMAS TRANSVERSE DENSE (CASE 2)
# data_conf.inputs  = [1,9,24]
# data_conf.outputs = [2,4,5,6,7]
# # VINCENT THOMAS VERTICAL SOUTH DECK EDGE (CASE 3)
# data_conf.inputs  = [14,19,26]
# data_conf.outputs = [16,18,22]
```

3.7.2 Method Inputs

3.7.3 General Parameters

parameter	value
`p`	number of output channels
`q`	number of input channels
`nt`	number of timesteps
`dt`	timestep
`decimation`	decimation (downsampling) factor
`order`	model order (2 times number of DOF)

3.7.4 Specific to Observer Kalman Identification (OKID)

parameter	value
`m`	number of Markov parameters to compute (at most = nt)

3.7.5 Specific to Eigensystem Realization Algorithm (ERA)

parameter	value
`horizon`	number of observability parameters, or prediction horizon
`nc`	number of controllability parameters

3.7.6 Specific to Data Correlations (DC)

parameter	value
`a`	(alpha) number of additional block rows in Hankel matrix of correlation matrices
`b`	(beta) number of additional block columns in Hankel matrix of correlation matrices
`l`	initial lag
`g`	lag (gap) between correlations

3.7.7 Specific to System Realization with Information Matrix (SRIM)

parameter	value
`horizon`	number of steps used for identification, or prediction horizon

3.7.8 Parameters for Mode Validation

parameter	value
`outlook`	number of steps used for temporal consistency in EMAC

[4]:
```
# Set Parameters
conf = Config()
conf.m  = 500
conf.horizon = 190
conf.nc = 190
conf.order  = 12
conf.a  = 0
conf.b  = 0
conf.l  = 10
conf.g  = 3
conf.period_band = (0.1,0.6)
conf.damping = 0.06
conf.pseudo = True
conf.outlook = 190
```

[5]:
```
# event_names = ["Berkeley", "San Lorenzo"]
event_modes = []
from matplotlib import pyplot as plt
file_axes = {}

fig, axs = plt.subplots(len(files), figsize=(6,2*len(files)), sharex=True, constrained_
↪layout=True)
axi = iter(axs)

method = "srim"

for i, file in enumerate(files):
    ax = next(axi)
    # fig, ax = plt.subplots(figsize=(10,5))
    print(file)
    try:
        event = quakeio.read(file, exclusions=["*filter*"])#, "*date*"])
        print("peak acceleration (cm/s/s):", event["peak_accel"])
        inputs, dt = extract_channels(event, data_conf.inputs)
        outpts, dt = extract_channels(event, data_conf.outputs)
        conf.decimation = 8 # decimation factor for state space method
        realization = mdof.system(method=method, inputs=inputs, outputs=outpts,␣
↪threads=18, chunk=200, **conf)
    except Exception as e:
        # raise e
        print(e)
        print(file.name)
        continue
    ss_modes = modal.system_modes(realization,dt,**conf)
    event_modes.append(list(ss_modes.values()))
    print_modes(ss_modes)
```

(continues on next page)

(continued from previous page)

```
    conf.decimation = 1 # decimation factor for transfer function methods
    periods, amplitudes = transform.fourier_transfer(inputs=inputs[0], outputs=outpts[0],
↪ step=dt, **conf)
    amplitudes = amplitudes/max(amplitudes)
    ax.plot(periods, amplitudes, label=["fourier" if i==0 else None][0], color="blue") #,
↪ alpha = (1/(len(files)+2))*(i+1))
    periods, amplitudes = transform.response_transfer(inputs=inputs[0],␣
↪outputs=outpts[0], step=dt, periods=periods, threads=8, **conf)
    amplitudes = amplitudes/max(amplitudes)
    ax.plot(periods, amplitudes, label=["response spectrum" if i==0 else None][0], color=
↪"green") #, alpha = (1/(len(files)+2))*(i+1))
    ax.vlines([1/value["freq"] for value in ss_modes.values() if value["energy_condensed_
↪emaco"]>0.5 and value["mpc"]>0.5], 0, 1, color='r', linestyles='dashed', label=[f
↪"state space ({method})" if i==0 else None][0])
    ax.set_xlim(conf.period_band)
    ax.set_title(event["event_date"])
fig.legend(bbox_to_anchor=(1.45,0.85))
event_frequencies = mode_statistics(event_modes, "freq")
fig.suptitle(title, fontsize=16)
```

```
../CESMD/CE89324/bayview_11oct2013_72086051_ce89324p.zip
peak acceleration (cm/s/s): 87.954
```

```
100%|| 1310/1311 [00:00<00:00, 2237.95it/s]
```

```
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.2709     0.001132   1.0        0.8338     0.8338
      0.2527     -0.003276  1.0        0.9749     0.9749
      0.2409     -0.0003217  1.0        0.8874     0.8874
      0.2337     -0.0001491  1.0        0.8368     0.8368
      0.2282     0.03774    1.0        0.9219     0.9219
      0.2133     0.02379    1.0        0.3374     0.3374
Mean Period(s): 0.23994997335785784
Standard Dev(s): 0.01829979557828316
../CESMD/CE89324/ferndaleoffshore_08dec2016_us20007z6r_ce89324p.zip
peak acceleration (cm/s/s): -30.371
```

```
100%|| 686/687 [00:00<00:00, 2885.01it/s]
/mnt/c/Users/cmp/Documents/GitHub/SystemIdentification/src/mdof/validation.py:137:␣
↪RuntimeWarning: divide by zero encountered in scalar divide
  nu[i]    = (s22[i]-s11[i])/(2*s12[i])
/mnt/c/Users/cmp/Documents/GitHub/SystemIdentification/src/mdof/validation.py:138:␣
↪RuntimeWarning: invalid value encountered in scalar multiply
  lam[0,i] = (s11[i]+s22[i])/2 + s12[i]*np.sqrt(nu[i]**2+1)
/mnt/c/Users/cmp/Documents/GitHub/SystemIdentification/src/mdof/validation.py:139:␣
↪RuntimeWarning: invalid value encountered in scalar multiply
  lam[1,i] = (s11[i]+s22[i])/2 - s12[i]*np.sqrt(nu[i]**2+1)
```

```
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.2359     0.0038     1.0        0.9453     0.9453
      0.227      0.03918    1.0        0.9725     0.9725
      0.2238     0.005151   1.0        0.9662     0.9662
```

(continues on next page)

(continued from previous page)

```
      0.2093     0.009579   1.0        0.604      0.604
      0.2032     0.02393    1.0        0.9483     0.9483
Mean Period(s): 0.21983789850293506
Standard Dev(s): 0.011930582991559505
../CESMD/CE89324/ferndale_28jan2015_72387946_ce89324p.zip
peak acceleration (cm/s/s): -285.334
100%|| 874/875 [00:00<00:00, 2546.99it/s]
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.3119     0.01321    1.0        0.8887     0.8887
      0.2602     -0.004556  1.0        0.7266     0.7266
      0.2555     0.04334    1.0        0.972      0.972
      0.2482     0.01867    1.0        0.878      0.878
      0.2299     0.008439   1.0        0.7712     0.7712
Mean Period(s): 0.26113898415437486
Standard Dev(s): 0.027387551087975463
../CESMD/CE89324/nc73201181_ce89324p.zip
peak acceleration (cm/s/s): 243.242
100%|| 831/832 [00:00<00:00, 2802.56it/s]
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.7111     0.06989    1.0        0.2373     0.2373
      0.2911     0.4793     0.0        0.2207     0.0
      0.2839     0.01621    1.0        0.9861     0.9861
      0.2669     0.008783   1.0        0.8865     0.8865
      0.2449     0.02909    1.0        0.9715     0.9715
      0.2391     0.09027    0.9849     0.9734     0.9587
Mean Period(s): 0.3394953575119655
Standard Dev(s): 0.16723260983987168
../CESMD/CE89324/nc73351710_ce89324p.zip
peak acceleration (cm/s/s): -53.539
100%|| 868/869 [00:00<00:00, 2740.65it/s]
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.2298     0.02758    1.0        0.9018     0.9018
      0.2293     0.01291    1.0        0.6597     0.6597
      0.2141     0.005794   1.0        0.8125     0.8125
      0.198      0.01652    1.0        0.954      0.954
      0.1856     0.007892   1.0        0.9607     0.9607
      0.1639     0.004047   1.0        0.953      0.953
Mean Period(s): 0.20345404869738717
Standard Dev(s): 0.023751109375293063
../CESMD/CE89324/nc73666231_ce89324p.zip
peak acceleration (cm/s/s): 30.272
100%|| 631/632 [00:00<00:00, 2616.91it/s]
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.2969     0.04186    1.0        0.3244     0.3244
```

(continues on next page)

(continued from previous page)

```
      0.2725     0.03001    1.0        0.8513     0.8513
      0.2432     0.02007    1.0        0.9816     0.9816
      0.2255     0.03043    1.0        0.8075     0.8075
      0.2231     0.0144     1.0        0.8528     0.8528
      0.1644     0.007564   1.0        0.9913     0.9913
Mean Period(s): 0.23760239670892977
Standard Dev(s): 0.041786791965444915
../CESMD/CE89324/nc73667866_ce89324p.zip
peak acceleration (cm/s/s): 32.846
```

```
100%|| 647/648 [00:00<00:00, 3019.03it/s]
```

```
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.2674     0.07844    1.0        0.2192     0.2192
      0.2388     0.01897    1.0        0.08927    0.08927
      0.2295     0.06567    1.0        0.5181     0.5181
      0.2229     0.03194    1.0        0.1985     0.1985
      0.2029     0.0984     0.004007   0.7416     0.002972
Mean Period(s): 0.23230004145748845
Standard Dev(s): 0.021121709869982592
../CESMD/CE89324/nc73714181_ce89324p.zip
peak acceleration (cm/s/s): 54.457
```

```
100%|| 694/695 [00:00<00:00, 3319.68it/s]
```

```
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.3978     0.1306     0.999      0.7373     0.7366
      0.3016     0.02847    1.0        0.7979     0.7979
      0.2591     0.03579    1.0        0.7103     0.7103
      0.236      0.009554   1.0        0.8541     0.8541
      0.2        0.03356    1.0        0.4057     0.4057
      0.1853     0.004686   1.0        0.1831     0.1831
Mean Period(s): 0.2633002998041649
Standard Dev(s): 0.0712209453393016
../CESMD/CE89324/nc73821036_ce89324p.zip
peak acceleration (cm/s/s): -1356.923
```

```
100%|| 1229/1230 [00:00<00:00, 2421.65it/s]
```

```
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.4041     0.02136    1.0        0.1498     0.1498
      0.3177     0.008901   1.0        0.8627     0.8627
      0.3006     0.00409    1.0        0.9268     0.9268
      0.2818     0.019      1.0        0.9697     0.9697
      0.2801     0.007793   1.0        0.8783     0.8783
      0.2441     0.1093     0.0004749  0.9427     0.0004477
Mean Period(s): 0.3047263057826448
Standard Dev(s): 0.049794929438951606
../CESMD/CE89324/nc73821046_ce89324p.zip
peak acceleration (cm/s/s): -139.897
```

```
100%|| 662/663 [00:00<00:00, 2844.54it/s]
```

```
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.3001     0.04037    1.0        0.7879     0.7879
      0.2674     0.03467    1.0        0.1593     0.1593
      0.2591     0.06272    1.0        0.6758     0.6758
      0.2469     0.03627    1.0        0.8245     0.8245
      0.2256     0.03078    1.0        0.7015     0.7015
      0.1745     0.03043    1.0        0.8787     0.8787
Mean Period(s): 0.24562036099179363
Standard Dev(s): 0.03891069442719765
../CESMD/CE89324/nc73821636_ce89324p.zip
peak acceleration (cm/s/s): 50.951
```

```
100%|| 673/674 [00:00<00:00, 3177.12it/s]
```

```
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.2833     0.009959   1.0        0.6816     0.6816
      0.2434     0.01689    1.0        0.8856     0.8856
      0.2357     0.0112     1.0        0.2451     0.2451
      0.2286     0.07124    0.9998     0.5308     0.5307
      0.2061     0.04924    1.0        0.9692     0.9692
      0.1684     0.03321    1.0        0.5699     0.5699
Mean Period(s): 0.22757668980327647
Standard Dev(s): 0.03510575192028343
../CESMD/CE89324/nc73827571_ce89324p.zip
peak acceleration (cm/s/s): -1012.657
```

```
100%|| 914/915 [00:00<00:00, 2689.89it/s]
```

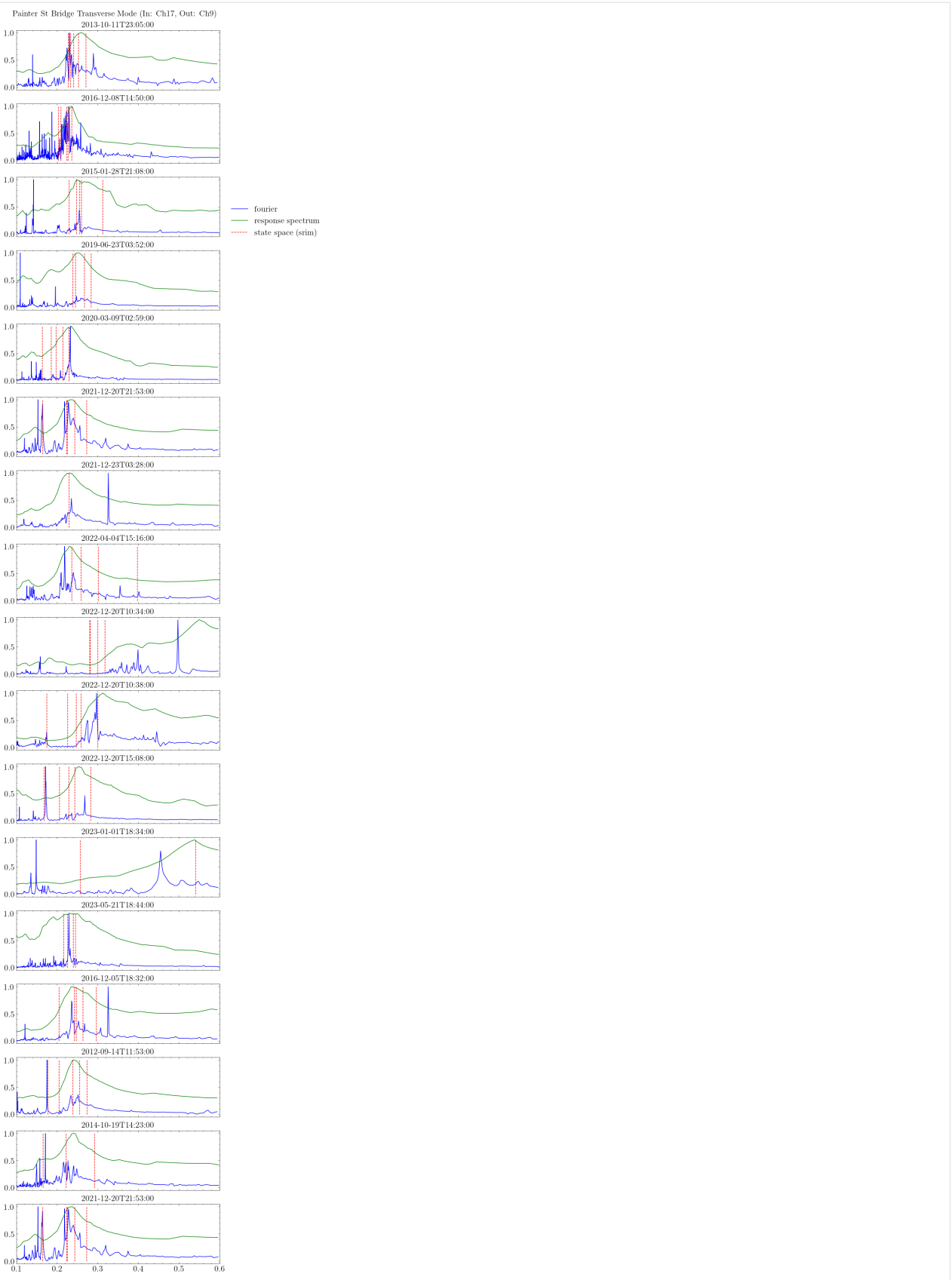
```
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.6212     0.09436    1.0        0.6528     0.6528
      0.5406     0.009841   1.0        0.8737     0.8737
      0.4367     0.04506    1.0        0.2452     0.2452
      0.317      0.01029    1.0        0.4412     0.4412
      0.2569     0.08572    0.9968     0.9454     0.9424
Mean Period(s): 0.4344633802066828
Standard Dev(s): 0.13524874602820816
../CESMD/CE89324/nc73890906_ce89324p.zip
peak acceleration (cm/s/s): 33.218
```

```
100%|| 694/695 [00:00<00:00, 2891.35it/s]
```

```
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.2607     0.02224    1.0        0.003157   0.003157
      0.2449     0.006788   1.0        0.9932     0.9932
      0.24       -0.0008864  1.0        0.8373     0.8373
      0.2263     0.02832    1.0        0.7906     0.7906
      0.2163     0.04559    1.0        0.9555     0.9555
      0.1748     0.001106   1.0        0.2454     0.2454
Mean Period(s): 0.22716129703326704
Standard Dev(s): 0.027283894562143994
../CESMD/CE89324/petrolia_05dec2016_72733405_ce89324p.zip
peak acceleration (cm/s/s): 165.262
```

```
100%|| 1310/1311 [00:00<00:00, 2282.88it/s]
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.2964     0.01012    1.0        0.9602     0.9602
      0.2634     0.04313    1.0        0.8054     0.8054
      0.2474     0.0159     1.0        0.8673     0.8673
      0.2425     0.012      1.0        0.8418     0.8418
      0.2228     0.01331    1.0        0.4284     0.4284
      0.2051     0.005986   1.0        0.8064     0.8064
Mean Period(s): 0.246280600421937
Standard Dev(s): 0.02905011899350602
../CESMD/CE89324/riodell_14sep2012_71842075_ce89324p.zip
peak acceleration (cm/s/s): -219.633
100%|| 1310/1311 [00:00<00:00, 2320.65it/s]
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.2735     0.03582    1.0        0.94       0.94
      0.2556     0.06232    1.0        0.998      0.998
      0.2391     0.005624   1.0        0.9759     0.9759
      0.2049     0.01358    1.0        0.8958     0.8958
      0.1772     0.0223     1.0        0.8821     0.8821
      0.1482     0.01812    1.0        0.3485     0.3485
Mean Period(s): 0.21639806443332765
Standard Dev(s): 0.044021503974172094
../CESMD/CE89324/riodell_19oct2014_72330211_ce89324p.zip
peak acceleration (cm/s/s): -60.641
100%|| 671/672 [00:00<00:00, 2351.78it/s]
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.2918     0.01039    1.0        0.9987     0.9987
      0.2418     -0.006623  1.0        0.3285     0.3285
      0.2226     0.02352    1.0        0.9517     0.9517
      0.1945     0.1566     3.285e-19  0.9412     3.092e-19
      0.1651     0.006289   1.0        0.7292     0.7292
Mean Period(s): 0.2231605195046272
Standard Dev(s): 0.04301403503187152
../CESMD/CE89324/RioDell_Petrolia_Processed_Data.zip
peak acceleration (cm/s/s): 30.272
100%|| 631/632 [00:00<00:00, 2284.98it/s]
Spectral quantities:
       T(s)                EMACO        MPC     EMACO*MPC
      0.2969     0.04186    1.0        0.3244     0.3244
      0.2725     0.03001    1.0        0.8513     0.8513
      0.2432     0.02007    1.0        0.9816     0.9816
      0.2255     0.03043    1.0        0.8075     0.8075
      0.2231     0.0144     1.0        0.8528     0.8528
      0.1644     0.007564   1.0        0.9913     0.9913
Mean Period(s): 0.23760239670892977
Standard Dev(s): 0.041786791965444915
```

[5]:
```
Text(0.5, 0.98, 'Painter St Bridge Transverse Mode (In: Ch17, Out: Ch9)')
```


Painter St Bridge Transverse Mode (In: Ch17, Out: Ch9)
2013-10-11T23:05:00
2016-12-08T14:50:00
2015-01-28T21:08:00
2019-06-23T03:52:00
2020-03-09T02:59:00
2021-12-20T21:53:00
2021-12-23T03:28:00
2022-04-04T15:16:00
2022-12-20T10:34:00
2022-12-20T10:38:00
2022-12-20T15:08:00
2023-01-01T18:34:00
2023-05-21T18:44:00
2016-12-05T18:32:00
2012-09-14T11:53:00
2014-10-19T14:23:00
2021-12-20T21:53:00
fourier
response spectrum
state space (srim)
0.0
0.5
1.0
0.1
0.2
0.3
0.4
0.5
0.6

PYTHON MODULE INDEX

m

mdof, ??
mdof.markov, ??
mdof.modal, ??
mdof.realize, ??
mdof.transform, ??