pysib — System Identification Toolbox

User Manual

Version 0.2.0

Diego Eckhard
Universidade Federal do Rio Grande do Sul
diego@eckhard.com.br

Contents

1 Overview

pysib is a Python toolbox for parameter identification of discrete-time, single-input single-output (SISO) dynamic systems. It implements the classical prediction-error estimators for polynomial input–output model structures: ARX, ARMAX, OE, and Box–Jenkins, together with auxiliary methods (Stieglitz–McBride, instrumental variables, correlation error minimization) and filtered variants designed to improve convergence in the presence of local minima.

1.1 General model structure

All model structures supported by pysib are special cases of

$$y(t) = \frac{B(q^{-1})}{A(q^{-1})F(q^{-1})}\,u(t) + \frac{C(q^{-1})}{A(q^{-1})D(q^{-1})}\,e(t), \tag{1}$$

where $u(t)$ is the input, $y(t)$ is the output, $e(t)$ is white noise, and q^{-1} is the backward-shift operator. The polynomials are written in non-negative powers of q^{-1}:

$$\begin{aligned}
A(q^{-1}) &= 1 + a_1 q^{-1} + \cdots + a_{n_a} q^{-n_a},\\
B(q^{-1}) &= b_0 + b_1 q^{-1} + \cdots + b_{n_b-1} q^{-(n_b-1)},\\
C(q^{-1}) &= 1 + c_1 q^{-1} + \cdots + c_{n_c} q^{-n_c},\\
D(q^{-1}) &= 1 + d_1 q^{-1} + \cdots + d_{n_d} q^{-n_d},\\
F(q^{-1}) &= 1 + f_1 q^{-1} + \cdots + f_{n_f} q^{-n_f}.
\end{aligned}$$

Each model structure fixes some polynomials to unity:

Structure	A	B	C	D	F
ARX	free	free	1	1	1
ARMAX	free	free	free	1	1
OE	1	free	1	1	free
BJ	1	free	free	free	free

An input delay of n_k samples is incorporated by prepending n_k zeros to the B polynomial.

1.2 Return convention

Every identification routine returns a pair `(theta, m)`:

- `theta`: 1-D `ndarray` of estimated free parameters, in the order listed in the API reference for each function.
- `m`: `dict` with keys `A`, `B`, `C`, `D`, `F`, each a 1-D `ndarray` of polynomial coefficients with the leading 1 included where applicable.

The `m` dictionary can be passed directly to `predict` and `simulate`, regardless of which estimator produced it.

2 Installation

2.1 Requirements

Dependency	Minimum version
Python	3.9
NumPy	1.20
SciPy	1.7
matplotlib	3.4
C compiler	GCC $\geq$ 9 or Clang $\geq$ 11
LAPACK	Accelerate (macOS) or `liblapack-dev` (Linux)

The C compiler and LAPACK are required only when building from source, for example on platforms for which a binary wheel is not available. On macOS, the Accelerate framework is available by default. On Linux (Debian/Ubuntu), install the build dependencies with:

```
sudo apt-get install gcc liblapack-dev
```

2.2 Installing via pip

```
pip install pysib
```

On supported platforms, pip installs a binary wheel containing the compiled extension modules. If no compatible wheel is available, pip falls back to building from the source distribution; in that case a working C compiler and LAPACK must be available before running this command.

2.3 Verifying the installation

```
python -c "import pysib; print('pysib OK')"
```

To run the test suite:

```
pip install pytest
python -m pytest tests/ -v
```

2.4 Running examples

Example scripts are available in the repository under `examples/`. They are self-contained and can be run directly from the repository root, for example:

```
python examples/example_arx.py
```

The main user-facing documentation is available at `https://pysib.net/`.

3 API Reference

All functions are accessible directly from the `pysib` namespace after `import pysib`.

3.1 Methods without optimization

These estimators do not require nonlinear optimization. ARX and correlation solve a single linear least-squares problem. IV uses the same structure with an instrument matrix. SM solves a sequence of linear problems iteratively (100 steps). They are generally fast, but can be biased when the noise model does not match the assumed structure.

3.1.1 `arx` — ARX identification

```
theta, m = pysib.arx(u, y, na, nb, nz)
```

Identifies an ARX model by ordinary least squares. The model structure is

$$A(q^{-1})\,y(t) = B(q^{-1})\,u(t - n_k) + e(t),$$

where $e(t)$ is white noise. The one-step-ahead predictor is linear in the parameters:

$$\hat{y}(t \mid t-1) = -\sum_{i=1}^{n_a} a_i\, y(t-i) + \sum_{j=0}^{n_b-1} b_j\, u(t-n_k-j),$$

so the prediction-error criterion $V(\theta) = \frac{1}{N}\sum_t [y(t) - \hat{y}(t \mid t-1)]^2$ is quadratic in θ and has a unique global minimum found by a single linear least-squares solve.

Arguments

Name	Type	Description
`u`	array_like	Input signal, length N.
`y`	array_like	Output signal, length N.
`na`	int	Number of A parameters: $a_1, \ldots, a_{n_a}$.
`nb`	int	Number of B parameters $b_0, \ldots, b_{n_b-1}$ (degree of B is $n_b - 1$).
`nz`	int	Input delay $n_k \geq 0$.

Returns

Name	Type	Value
`theta`	ndarray	$[a_1, \ldots, a_{n_a},\ b_0, \ldots, b_{n_b-1}]$
`m`	dict	`A`, `B` free; `C`=`D`=`F`= [1]

3.1.2 `sm` — Stieglitz–McBride

```
theta, m = pysib.sm(u, y, nb, nf, nz)
```

Identifies an OE-structure model by the Stieglitz–McBride iterative method. The model structure is

$$y(t) = \frac{B(q^{-1})}{F(q^{-1})}\,u(t - n_k) + e(t),$$

where $e(t)$ is white noise. At each iteration the input and output are filtered by the current $\hat{F}$ estimate,

$$u_f(t) = \frac{1}{\hat{F}(q^{-1})}\,u(t), \qquad y_f(t) = \frac{1}{\hat{F}(q^{-1})}\,y(t),$$

and an ARX problem is solved on the filtered data to update the estimate. The procedure runs for a fixed 100 iterations; no convergence criterion is applied. SM is computationally cheaper than OE but may converge to a different solution in the presence of local minima.

Arguments

Name	Type	Description
`u`	array_like	Input signal.
`y`	array_like	Output signal.
`nb`	int	Number of B parameters.
`nf`	int	Number of F parameters.
`nz`	int	Input delay $n_k \geq 0$.

Returns

Name	Type	Value
`theta`	ndarray	$[b_0, \ldots, b_{n_b-1},\ f_1, \ldots, f_{n_f}]$
`m`	dict	`B`, `F` free; `A`=`C`=`D`= [1]

3.1.3 `correlation` — Correlation method

```
theta, m = pysib.correlation(u, y, na, nb, nz, M=20, z=None)
```

Identifies an ARX model structure by requiring the prediction error $\varepsilon(t;\theta) = y(t) - \hat{y}(t \mid t-1;\theta)$ to be uncorrelated with the instrument z at lags $\tau = 0, 1, \ldots, M-1$. This method does *not* minimize the sum of squared prediction errors. Instead it seeks θ satisfying the M linear correlation conditions

$$\frac{1}{N}\sum_{t} z(t-\tau)\,\varepsilon(t;\theta) = 0, \quad \tau = 0, 1, \ldots, M-1.$$

For $M = n_a + n_b$ the system is exactly determined and solved directly. For $M > n_a + n_b$ it is overdetermined; the least-squares solution minimizes the sum of squared cross-correlations,

$$\sum_{\tau=0}^{M-1} \left[\frac{1}{N}\sum_{t} z(t-\tau)\,\varepsilon(t;\theta)\right]^2,$$

which is a quadratic in θ (not the sum of squared prediction errors). If `z` is not supplied, the input `u` is used as the instrument. The estimator is consistent whenever the instrument is uncorrelated with the noise, regardless of whether the noise is white.

Arguments

Name	Type	Description
`u`	array_like	Input signal.
`y`	array_like	Output signal.
`na`	int	Number of A parameters.
`nb`	int	Number of B parameters.
`nz`	int	Input delay $n_k \geq 0$.
`M`	int	Number of correlation lags (default 20).
`z`	array_like or None	External instrument; if `None`, `u` is used (default).

Returns

Name	Type	Value
`theta`	ndarray	$[a_1, \ldots, a_{n_a},\ b_0, \ldots, b_{n_b-1}]$
`m`	dict	`A`, `B` free; `C`=`D`=`F`= [1]

3.1.4 `iv` — Instrumental variables

```
theta, m = pysib.iv(u, y1, y2, na, nb, nz)
```

Identifies an ARX model structure using the instrumental variable (IV) method. Two independent experiments are required: both are driven by the same input `u`, but the output noise sequences are independent. The model structure is the same ARX form used by `arx`:

$$y_1(t) = -\sum_{i=1}^{n_a} a_i\, y_1(t-i) + \sum_{j=0}^{n_b-1} b_j\, u(t-n_k-j) + e_1(t).$$

At each time step, the regression vector is

$$\varphi(t) = \big[-y_1(t-1),\ \ldots,\ -y_1(t-n_a),\ u(t-n_k),\ \ldots,\ u(t-n_k-n_b+1)\big]^\top,$$

and the instrument vector replaces the lagged `y1` terms with the corresponding lagged `y2` terms:

$$\zeta(t) = \big[-y_2(t-1),\ \ldots,\ -y_2(t-n_a),\ u(t-n_k),\ \ldots,\ u(t-n_k-n_b+1)\big]^\top.$$

Stacking these vectors into matrices Φ_1 and Z (one row per time step), the IV estimate is

$$\hat{\theta} = (Z^\top \Phi_1)^{-1} Z^\top y_1.$$

This estimator does *not* minimize a prediction-error criterion. The ordinary least-squares solution

$$\hat{\theta}_{\mathrm{OLS}} = (\Phi_1^\top \Phi_1)^{-1} \Phi_1^\top y_1$$

is biased when the noise enters through $1/A(q^{-1})$, because $\varphi(t)$ contains lagged y_1 values that are correlated with $e_1(t)$. The instrument $\zeta(t)$ shares the input-driven component of $\varphi(t)$ but is correlated with a different, independent noise sequence, so $Z^\top e_1 \to 0$ as $N \to \infty$ and $\hat{\theta}$ is consistent.

Arguments

Name	Type	Description
`u`	array_like	Input signal (same in both experiments).
`y1`	array_like	Output from the first experiment.
`y2`	array_like	Output from the second experiment.
`na`	int	Number of A parameters.
`nb`	int	Number of B parameters.
`nz`	int	Input delay $n_k \geq 0$.

Returns

Name	Type	Value
`theta`	ndarray	$[a_1, \ldots, a_{n_a},\ b_0, \ldots, b_{n_b-1}]$
`m`	dict	`A`, `B` free; `C`=`D`=`F`= [1]

3.2 Methods with optimization

These estimators minimize the prediction-error criterion

$$V(\theta) = \frac{1}{N} \sum_t \varepsilon(t;\theta)^2.$$

The criterion is nonlinear in θ because the denominator polynomials (F, or C and D) appear in the predictor. An ARX estimate provides the initial guess in all cases; the optimizer then runs two phases implemented in a C extension.

Sensitivity signals. At each θ, the sensitivity vector $\psi(t;\theta) = -\partial\varepsilon(t;\theta)/\partial\theta$ is computed by filtering. For the OE structure, for example,

$$\psi_j^B(t) = \frac{u(t-n_k)}{F(q^{-1})} \cdot q^{-j}, \qquad \psi_j^F(t) = \frac{\hat{y}(t)}{F(q^{-1})} \cdot q^{-j},$$

where $\hat{y}(t) = (B/F)\,u(t-n_k)$ is the current simulated output. ARMAX and BJ use analogous filtered signals for their respective polynomials.

Gradient and Gauss–Newton Hessian. The gradient and an approximate Hessian are assembled from the sensitivity signals and the current prediction error $\varepsilon(t)$:

$$g_j = -\sum_t \varepsilon(t)\,\psi_j(t), \qquad H_{jk} = \sum_t \psi_j(t)\,\psi_k(t).$$

H is the Gauss–Newton approximation to the Hessian of V: it retains only the outer-product term $\Psi^\top\Psi$ and drops the second-derivative term (which vanishes near the optimum).

Phase 1 — steepest descent with smoothed direction. The gradient direction is smoothed by an exponential moving average before each step:

$$d \leftarrow \tfrac{4}{5}\,d + \tfrac{1}{5}\,g.$$

The update is $\theta \leftarrow \theta - \alpha\,d/\|d\|$, where α is adapted per step (multiplied by 1.01 on a successful reduction, by 0.99 otherwise). This phase runs for up to 100×100 gradient evaluations.

Phase 2 — Gauss–Newton with backtracking. The Newton direction d is obtained by solving $H\,d = g$ via LAPACK `dposv` (Cholesky factorisation for symmetric positive definite matrices). The step length grows linearly with iteration i: $\theta \leftarrow \theta - (i/1000)\,d$. If the step increases V, the candidate is bisected up to 10 times. This phase runs for up to 1 000 iterations.

3.2.1 `oe` — Output Error

```
theta, m = pysib.oe(u, y, nb, nf, nz)
```

Identifies an OE model by minimizing the prediction-error criterion. The model structure is

$$y(t) = \frac{B(q^{-1})}{F(q^{-1})}\,u(t - n_k) + e(t),$$

where $e(t)$ is white noise. For this structure the one-step-ahead predictor equals the noise-free simulated output, so the criterion is

$$V(\theta) = \frac{1}{N}\sum_{t}\left[y(t) - \frac{B(q^{-1})}{F(q^{-1})}\,u(t - n_k)\right]^2.$$

This is nonlinear in θ because F appears in the denominator. An ARX estimate provides the initial guess; the optimizer then alternates steepest-descent and Newton steps (implemented in a C extension, with LAPACK used for the Newton step).

Arguments

Name	Type	Description
`u`	array_like	Input signal.
`y`	array_like	Output signal.
`nb`	int	Number of B parameters.
`nf`	int	Number of F parameters.
`nz`	int	Input delay $n_k \geq 0$.

Returns

Name	Type	Value
`theta`	ndarray	$[b_0, \ldots, b_{n_b-1},\ f_1, \ldots, f_{n_f}]$
`m`	dict	`B`, `F` free; `A`=`C`=`D`= [1]

3.2.2 `armax` — ARMAX

```
theta, m = pysib.armax(u, y, na, nb, nc, nz)
```

Identifies an ARMAX model by minimizing the prediction-error criterion. The model structure is

$$A(q^{-1})\,y(t) = B(q^{-1})\,u(t-n_k) + C(q^{-1})\,e(t),$$

where $e(t)$ is white noise. The one-step-ahead prediction error is

$$\varepsilon(t;\theta) = \frac{A(q^{-1})}{C(q^{-1})}\,y(t) - \frac{B(q^{-1})}{C(q^{-1})}\,u(t-n_k),$$

and the criterion $V(\theta) = \frac{1}{N}\sum_t \varepsilon(t;\theta)^2$ is minimized. Because C appears in the denominator, the predictor depends on past residuals and the problem is nonlinear in θ. An ARX estimate (with C initialized to zero) provides the initial guess; the optimizer uses steepest-descent and Newton steps implemented in a C extension, with LAPACK used for the Newton step.

Arguments

Name	Type	Description
`u`	array_like	Input signal.
`y`	array_like	Output signal.
`na`	int	Number of A parameters.
`nb`	int	Number of B parameters.
`nc`	int	Number of C parameters.
`nz`	int	Input delay $n_k \geq 0$.

Returns

Name	Type	Value
`theta`	ndarray	$[a_1, \ldots, a_{n_a},\ b_0, \ldots, b_{n_b-1},\ c_1, \ldots, c_{n_c}]$
`m`	dict	`A`, `B`, `C` free; `D`=`F`= [1]

3.2.3 `bj` — Box–Jenkins

```
theta, m = pysib.bj(u, y, nb, nc, nd, nf, nz)
```

Identifies a Box–Jenkins model by minimizing the prediction-error criterion. The model structure is

$$y(t) = \frac{B(q^{-1})}{F(q^{-1})}\,u(t-n_k) + \frac{C(q^{-1})}{D(q^{-1})}\,e(t),$$

where $e(t)$ is white noise. The one-step-ahead prediction error is

$$\varepsilon(t;\theta) = \frac{D(q^{-1})}{C(q^{-1})}\left[y(t) - \frac{B(q^{-1})}{F(q^{-1})}\,u(t-n_k)\right],$$

and the criterion $V(\theta) = \frac{1}{N}\sum_t \varepsilon(t;\theta)^2$ is minimized. The problem is nonlinear in θ because C, D, and F all appear in denominators. An ARX estimate provides the initial guess for B and F; C and D are initialized from the ARX denominator. The optimizer uses steepest-descent and Newton steps implemented in a C extension, with LAPACK used for the Newton step.

Arguments

Name	Type	Description
`u`	array_like	Input signal.
`y`	array_like	Output signal.
`nb`	int	Number of B parameters.
`nc`	int	Number of C parameters.
`nd`	int	Number of D parameters.
`nf`	int	Number of F parameters.
`nz`	int	Input delay $n_k \geq 0$.

Returns

Name	Type	Value
`theta`	ndarray	$[b_0, \ldots, b_{n_b-1},\ c_1, \ldots, c_{n_c},\ d_1, \ldots, d_{n_d},\ f_1, \ldots, f_{n_f}]$
`m`	dict	`B`, `C`, `D`, `F` free; `A`= [1]

3.3 Filtered estimators

The filtered estimators address the sensitivity of the nonlinear prediction-error criterion to local minima by a cost-function shaping strategy: both `u` and `y` are low-pass filtered before the optimization, which removes high-frequency content and reshapes the objective to be more convex in the low-frequency region. The optimization is repeated 9 times with progressively wider passbands, then once more on the original unfiltered data to obtain the final estimate.

Filter sequence for `oe_filtered`. A first-order DC-normalized IIR filter is used:

$$L_i(q^{-1}) = \frac{1 - a_i}{1 - a_i\, q^{-1}}, \quad i = 1, \ldots, 9.$$

The poles a_i are spaced so that the impulse response decays to 5% in τ_i samples, with τ_i decreasing from 36 to 4:

$$a_i = 0.05^{1/\tau_i}, \quad \tau_i \in \{36,\, 32,\, 28,\, 24,\, 20,\, 16,\, 12,\, 8,\, 4\}.$$

The resulting poles decrease from $a_1 \approx 0.920$ to $a_9 \approx 0.473$, progressively widening the passband. The initial guess for stage 1 is an ARX estimate computed on the most-narrowly filtered data; each subsequent stage warm-starts from the previous estimate.

Filter sequence for `armax_filtered` and `bj_filtered`. A first-order Butterworth low-pass filter is used at each stage:

$$L_i = \text{Butterworth}(1,\, \omega_c = 0.1\, i\, \pi), \quad i = 1, \ldots, 9,$$

where ω_c is the $-3\,\text{dB}$ cutoff in rad/sample. The cutoff increases from 0.1π to 0.9π, covering progressively wider bands. The initial guess is an ARX estimate on the unfiltered data; each subsequent stage warm-starts from the previous estimate.

3.3.1 `oe_filtered`

```
theta, m = pysib.oe_filtered(u, y, nb, nf, nz)
```

Identifies an OE model using a cost-function shaping strategy to improve convergence. The model structure and the criterion minimized at each stage are the same as for `oe`:

$$y(t) = \frac{B(q^{-1})}{F(q^{-1})}\,u(t-n_k) + e(t), \qquad V(\theta) = \frac{1}{N}\sum_{t}\left[y(t) - \frac{B(q^{-1})}{F(q^{-1})}\,u(t-n_k)\right]^2.$$

Before the final unfiltered OE step, 9 auxiliary OE problems are solved on filtered versions of the data. At stage i, both `u` and `y` are passed through a first-order low-pass filter $(1-a_i)/(1-a_i\,q^{-1})$, and the OE criterion is minimized on the filtered data. The pole a_i decreases from $\approx$0.92 to $\approx$0.47, progressively widening the passband from a narrow low-frequency window toward the full spectrum. Filtering reshapes the cost function, reducing the attraction of undesirable local minima before the estimate is refined on the original data.

Arguments

Name	Type	Description
`u`	array_like	Input signal.
`y`	array_like	Output signal.
`nb`	int	Number of B parameters.
`nf`	int	Number of F parameters.
`nz`	int	Input delay $n_k \geq 0$.

Returns

Name	Type	Value
`theta`	ndarray	$[b_0, \ldots, b_{n_b-1},\ f_1, \ldots, f_{n_f}]$
`m`	dict	`B`, `F` free; `A`=`C`=`D`= [1]

3.3.2 `armax_filtered`

```
theta, m = pysib.armax_filtered(u, y, na, nb, nc, nz)
```

Identifies an ARMAX model using a cost-function shaping strategy to improve convergence. The model structure and the criterion minimized at each stage are the same as for `armax`:

$$A(q^{-1})\,y(t) = B(q^{-1})\,u(t-n_k) + C(q^{-1})\,e(t), \qquad V(\theta) = \frac{1}{N}\sum_{t}\varepsilon(t;\theta)^2.$$

Before the final unfiltered ARMAX step, 9 auxiliary ARMAX problems are solved on filtered versions of the data. At stage i, both `u` and `y` are passed through a first-order low-pass Butterworth filter with cutoff frequency $0.1i\,\pi$ rad/sample ($i = 1, \ldots, 9$), and the ARMAX criterion is minimized on the filtered data. The cutoff increases from 0.1π to 0.9π, progressively widening the passband and reshaping the cost function to reduce the attraction of undesirable local minima before the estimate is refined on the original data.

Arguments

Name	Type	Description
`u`	array_like	Input signal.
`y`	array_like	Output signal.
`na`	int	Number of A parameters.
`nb`	int	Number of B parameters.
`nc`	int	Number of C parameters.
`nz`	int	Input delay $n_k \geq 0$.

Returns

Name	Type	Value
`theta`	ndarray	$[a_1, \ldots, a_{n_a},\ b_0, \ldots, b_{n_b-1},\ c_1, \ldots, c_{n_c}]$
`m`	dict	`A`, `B`, `C` free; `D`=`F`= [1]

3.3.3 `bj_filtered`

```
theta, m = pysib.bj_filtered(u, y, nb, nc, nd, nf, nz)
```

Identifies a Box–Jenkins model using a cost-function shaping strategy to improve convergence. The model structure and the criterion minimized at each stage are the same as for `bj`:

$$y(t) = \frac{B(q^{-1})}{F(q^{-1})}\,u(t-n_k) + \frac{C(q^{-1})}{D(q^{-1})}\,e(t), \qquad V(\theta) = \frac{1}{N}\sum_{t}\varepsilon(t;\theta)^2.$$

Before the final unfiltered BJ step, 9 auxiliary BJ problems are solved on filtered versions of the data. At stage i, both `u` and `y` are passed through a first-order low-pass Butterworth filter with cutoff frequency $0.1i\,\pi$ rad/sample ($i = 1, \ldots, 9$), and the BJ criterion is minimized on the filtered data. The cutoff increases from 0.1π to 0.9π, progressively widening the passband and reshaping the cost function to reduce the attraction of undesirable local minima before the estimate is refined on the original data.

Arguments

Name	Type	Description
`u`	array_like	Input signal.
`y`	array_like	Output signal.
`nb`	int	Number of B parameters.
`nc`	int	Number of C parameters.
`nd`	int	Number of D parameters.
`nf`	int	Number of F parameters.
`nz`	int	Input delay $n_k \geq 0$.

Returns

Name	Type	Value
`theta`	ndarray	$[b_0, \ldots, b_{n_b-1},\ c_1, \ldots, c_{n_c},\ d_1, \ldots, d_{n_d},\ f_1, \ldots, f_{n_f}]$
`m`	dict	`B`, `C`, `D`, `F` free; `A`= [1]

3.4 Extra functions

After identification, three utility functions support model validation and analysis. `predict` and `simulate` both take the model dictionary `m` returned by any estimator. `predict` computes the one-step-ahead prediction $\hat{y}(t \mid t-1)$ using both the input and the observed output; residuals $\varepsilon(t) = y(t) - \hat{y}(t \mid t-1)$ should be white and uncorrelated with the input for a correct model. `simulate` computes the open-loop output $G(q^{-1})\,u(t)$ using only the input signal; it does not use output observations or the noise model (C, D are ignored). Both functions can be applied to validation data; they assess complementary aspects of model quality. `plota` is a Monte Carlo diagnostic: given a matrix of parameter estimates from repeated experiments, it sorts the runs and plots all parameters together so that the spread and bias of the estimates can be assessed visually.

3.4.1 `predict` — One-step-ahead prediction

```
yp = pysib.predict(u, y, m)
```

Computes the one-step-ahead prediction:

$$\hat{y}(t \mid t-1) = y(t) + H^{-1}(q^{-1})\big[G(q^{-1})\,u(t) - y(t)\big],$$

where $G = B/(AF)$ and $H = C/(AD)$.

Arguments

Name	Type	Description
`u`	array_like	Input signal.
`y`	array_like	Output signal.
`m`	dict	Model dictionary with keys `A, B, C, D, F`.

Returns

`yp`: ndarray of predicted output values, same length as `u`.

3.4.2 `simulate` — Noise-free simulation

```
ys = pysib.simulate(u, m)
```

Computes the open-loop, noise-free model output

$$y_s(t) = G(q^{-1})\,u(t) = \frac{B(q^{-1})}{A(q^{-1})\,F(q^{-1})}\,u(t).$$

Only the input signal is required; the noise polynomials C and D are not used. Comparing y_s against measured outputs on a validation experiment assesses the quality of the plant dynamics, independently of the noise model.

Arguments

Name	Type	Description
`u`	array_like	Input signal.
`m`	dict	Model dictionary with keys `A, B, C, D, F`.

Returns

`ys`: ndarray of simulated output values, same length as `u`.

3.4.3 `plota` — Monte Carlo plot

```
Torg = pysib.plota(T, a)
```

Sorts the columns of a Monte Carlo parameter matrix by the value in row `a` and plots all rows against the run index.

Arguments

Name	Type	Description
`T`	ndarray, shape $(n_\theta, n_{\text{runs}})$	Matrix of parameter estimates from Monte Carlo runs.
`a`	int	Row index used for sorting (0-based).

Returns

`Torg`: ndarray — parameter matrix sorted by row `a`.

4 Examples

4.1 ARX

Based on `examples/example_arx.py`.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lfilter
import pysib

N = 1000
t = np.arange(N)
u = np.sign(np.sin(2 * np.pi * t / 100))  # square wave

# True model:  y(t) = 0.9*y(t-1) + u(t-1) + e(t)
y = lfilter([0, 1], [1, -0.9], u) + lfilter([1], [1, -0.9], np.random.randn(N))

theta, m = pysib.arx(u, y, na=1, nb=1, nz=1)
print("theta:", theta)

yp = pysib.predict(u, y, m)
ys = pysib.simulate(u, m)

plt.plot(t, y, label="Data")
plt.plot(t, yp, label="Prediction")
plt.plot(t, ys, label="Simulation")
plt.legend(); plt.show()
```

4.2 OE and Stieglitz–McBride

Based on `examples/example_oe.py` and `examples/example_sm.py`.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lfilter
import pysib

N = 100
t = np.arange(N)
u = np.sign(np.sin(2 * np.pi * t / 100))

# True OE model:  y(t) = u(t-1) / (1 - 0.9*z^-1) + e(t)
y = lfilter([0, 1], [1, -0.9], u) + 0.01 * np.random.randn(N)

theta_oe, m_oe = pysib.oe(u, y, nb=1, nf=1, nz=1)
theta_sm, m_sm = pysib.sm(u, y, nb=1, nf=1, nz=1)
print("OE theta:", theta_oe)
print("SM theta:", theta_sm)

yp = pysib.predict(u, y, m_oe)
ys = pysib.simulate(u, m_oe)

plt.plot(t, y, label="Data")
plt.plot(t, yp, label="Prediction (OE)")
plt.plot(t, ys, label="Simulation (OE)")
plt.legend(); plt.show()
```

4.3 ARMAX

Based on `examples/example_armax.py`.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lfilter
import pysib

N = 1000
t = np.arange(N)
u = np.sign(np.sin(2 * np.pi * t / 100))

# True ARMAX:  y = u(t-1)/(1-0.9z^-1) + (1-0.5z^-1)/(1-0.9z^-1)*e(t)
y = (lfilter([0, 1], [1, -0.9], u)
   + lfilter([1, -0.5], [1, -0.9], np.random.randn(N)))

theta, m = pysib.armax(u, y, na=1, nb=1, nc=1, nz=1)
print("theta:", theta)

yp = pysib.predict(u, y, m)
ys = pysib.simulate(u, m)

plt.plot(t, y, label="Data")
plt.plot(t, yp, label="Prediction")
plt.plot(t, ys, label="Simulation")
plt.legend(); plt.show()
```

4.4 Box–Jenkins

Based on `examples/example_bj.py`.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lfilter
import pysib

N = 1000
t = np.arange(N)
u = np.sign(np.sin(2 * np.pi * t / 100))

# True BJ:  y = u(t-1)/(1-0.9z^-1) + (1-0.5z^-1)/(1-1.5z^-1+0.7z^-2)*e(t)
y = (lfilter([0, 1], [1, -0.9], u)
   + lfilter([1, -0.5], [1, -1.5, 0.7], 0.05 * np.random.randn(N)))

theta, m = pysib.bj(u, y, nb=1, nc=1, nd=2, nf=1, nz=1)
print("theta:", theta)

yp = pysib.predict(u, y, m)
ys = pysib.simulate(u, m)

plt.plot(t, y, label="Data")
plt.plot(t, yp, label="Prediction")
plt.plot(t, ys, label="Simulation")
plt.legend(); plt.show()
```

4.5 Correlation error minimization

Based on `examples/example_correlation.py`.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lfilter
import pysib

N = 1000
t = np.arange(N)
u = np.sign(np.sin(2 * np.pi * t / 100))

# True ARX: y(t) = 0.9*y(t-1) + u(t-1) + e(t)
y = lfilter([0, 1], [1, -0.9], u) + np.random.randn(N)

theta_ls,   m_ls   = pysib.arx(u, y, na=1, nb=1, nz=1)
theta_corr, m_corr = pysib.correlation(u, y, na=1, nb=1, nz=1, M=30)
print("ARX        :", theta_ls)
print("Correlation:", theta_corr)

yp = pysib.predict(u, y, m_corr)
ys = pysib.simulate(u, m_corr)

plt.plot(t, y, label="Data")
plt.plot(t, yp, label="Prediction (correlation)")
plt.plot(t, ys, label="Simulation (correlation)")
plt.legend(); plt.show()
```

4.6 Instrumental variables

Based on `examples/example_iv.py`.

```
import numpy as np
from scipy.signal import lfilter
import pysib

N = 1000
t = np.arange(N)
u = np.sign(np.sin(2 * np.pi * t / 100))

# Two independent noise realizations, same input
y1 = lfilter([0, 1], [1, -0.9], u) + np.random.randn(N)
y2 = lfilter([0, 1], [1, -0.9], u) + np.random.randn(N)

theta_arx, _ = pysib.arx(u, y1, na=1, nb=1, nz=1)
theta_iv,  _ = pysib.iv(u, y1, y2, na=1, nb=1, nz=1)
print("ARX:", theta_arx)
print("IV :", theta_iv)
```

4.7 OE with filtered initialization

Based on `examples/example_oe_filtered.py`.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lfilter
import pysib
```

```
N = 100
t = np.arange(N)
u = np.sign(np.sin(2 * np.pi * t / 100))

y = lfilter([0, 1], [1, -0.9], u) + 0.1 * np.random.randn(N)

theta_oe,   m_oe   = pysib.oe(u, y, nb=1, nf=1, nz=1)
theta_filt, m_filt = pysib.oe_filtered(u, y, nb=1, nf=1, nz=1)
print("OE         :", theta_oe)
print("OE filtered:", theta_filt)

yp = pysib.predict(u, y, m_filt)
ys = pysib.simulate(u, m_filt)

plt.plot(t, y, label="Data")
plt.plot(t, yp, label="Prediction (filtered)")
plt.plot(t, ys, label="Simulation (filtered)")
plt.legend(); plt.show()
```

4.8 ARMAX with filtered initialization

Based on `examples/example_armax_filtered.py`.

```
import numpy as np
from scipy.signal import lfilter
import pysib

N = 1000
t = np.arange(N)
u = np.sign(np.sin(2 * np.pi * t / 100))

y = (lfilter([0, 1], [1, -0.9], u)
   + lfilter([1, -0.5], [1, -0.9], np.random.randn(N)))

theta_armax, _ = pysib.armax(u, y, na=1, nb=1, nc=1, nz=1)
theta_filt,  _ = pysib.armax_filtered(u, y, na=1, nb=1, nc=1, nz=1)
print("ARMAX         :", theta_armax)
print("ARMAX filtered:", theta_filt)
```

4.9 Box–Jenkins with filtered initialization

Based on `examples/example_bj_filtered.py`.

```
import numpy as np
from scipy.signal import lfilter
import pysib

N = 1000
t = np.arange(N)
u = np.sign(np.sin(2 * np.pi * t / 100))

y = (lfilter([0, 1], [1, -0.9], u)
   + lfilter([1, -0.5], [1, -1.5, 0.7], 0.05 * np.random.randn(N)))

theta_bj,   _ = pysib.bj(u, y, nb=1, nc=1, nd=2, nf=1, nz=1)
theta_filt, _ = pysib.bj_filtered(u, y, nb=1, nc=1, nd=2, nf=1, nz=1)
```

```
print("BJ         :", theta_bj)
print("BJ filtered:", theta_filt)
```

4.10 Monte Carlo plot

Based on `examples/example_monte_carlo.py`.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import lfilter
import pysib

M = 100
N = 100
T = np.full((2, M), np.nan)

for run in range(M):
    t = np.arange(N)
    u = np.sign(np.sin(2 * np.pi * t / 100))
    y = lfilter([0, 1], [1, -0.9], u) + 10 * np.random.randn(N)
    theta, _ = pysib.oe(u, y, nb=1, nf=1, nz=1)
    T[:, run] = theta

T_sorted = pysib.plota(T, a=0)
plt.title("OE Monte-Carlo (100 runs)")
plt.show()
```

5 Parameter selection guide

5.1 Model orders

`na` Number of A parameters $(a_1, \ldots, a_{n_a})$. Start with `na = 1` or `2` for first- and second-order systems.

`nb` Number of B parameters $(b_0, \ldots, b_{n_b-1})$. `nb = 1` gives a single gain b_0; the degree of B is $n_b - 1$.

`nf` Number of F parameters $(f_1, \ldots, f_{n_f})$, equal to the degree of F (poles of the noise-free model in OE and BJ). Typically equal to the system order.

`nc` Number of C parameters $(c_1, \ldots, c_{n_c})$, equal to the degree of C (noise MA polynomial in ARMAX and BJ). Start with `nc = 1` and increase only if residuals remain correlated.

`nd` Number of D parameters $(d_1, \ldots, d_{n_d})$, equal to the degree of D (noise AR polynomial in BJ only). Start with `nd = 1`.

`nz` Input delay in samples. `nz = 1` means the input affects the output one step later (b_0 multiplies $u(t-1)$). Set `nz` equal to the known or estimated dead time of the process.

5.2 Which estimator to use

The central question is *where the noise enters the system.* All model structures are special cases of $y = G\,u + H\,e$ where $G = B/(AF)$ is the plant and $H = C/(AD)$ is the noise model. Different assumptions about H lead to different estimators.

Process noise (ARX, ARMAX). If the disturbance enters at the plant input, it is filtered by the plant before reaching the output. The output noise therefore has the same poles as the plant, giving the ARX noise model $H = 1/A$. In this case ordinary least squares on the ARX structure is consistent. If the residuals from ARX are correlated (the noise has additional moving-average dynamics), extend to ARMAX by adding the C polynomial, making $H = C/A$.

Measurement noise (OE, BJ). If the disturbance is additive at the output (sensor or measurement noise), the noise is independent of the plant dynamics. The OE structure captures this with $H = 1$ (white noise). If the measurement noise is itself colored, BJ provides the most general description: $H = C/D$ with poles and zeros independent of the plant $G = B/F$. Applying ARX to data with additive output noise yields biased estimates because the lagged output regressors are then correlated with the noise.

Moment-based methods for the ARX structure (IV, correlation). When ARX ordinary least squares is expected to be biased, two moment-based alternatives avoid nonlinear optimization entirely. The *instrumental variable* method requires two independent experiments with the same input: it uses the output of the second experiment as an instrument for the first, yielding a consistent estimate because the instrument shares the input-driven component but has independent noise. The *correlation* method works with a single experiment: it finds θ such that the prediction error is uncorrelated with the instrument z at M lags. By default $z = u$, which is a valid instrument whenever the input is uncorrelated with the noise (open-loop operation). An external instrument can be supplied when this condition does not hold. Using $M > n_a + n_b$ lags over-determines the system and improves robustness.

Situation	Estimator	Notes
Quick estimate; disturbance enters at plant input (process noise)	ARX	Closed-form; consistent when $H = 1/A$. Biased under additive output noise.
Additive output noise (measurement or sensor noise)	OE	Consistent plant model B/F; nonlinear optimization.
OE structure; faster computation preferred	SM	Iterative ARX steps (100 iterations); cheaper but may converge to a local minimum.
ARX residuals are correlated; noise shares plant denominator	ARMAX	Extends ARX with MA noise C/A; nonlinear optimization.
Plant and noise have independent pole structures	BJ	Fully separate plant B/F and noise C/D; most parameters; needs most data.
Two independent experiments with the same input available	IV	Consistent under ARX noise structure; no nonlinear optimization.
Single experiment; input uncorrelated with noise (open-loop); or external instrument available	Correlation	Consistent ARX estimate without a second experiment; $M > n_a + n_b$ lags improve robustness.

5.3 Filtered vs. standard estimators

The prediction-error criterion $V(\theta)$ is nonlinear in θ whenever denominator polynomials are free (F in OE, C or D in ARMAX and BJ). Nonlinear optimizers — including the steepest-descent and Gauss–Newton algorithm implemented here — can converge to a local minimum rather than the global one, particularly when the system has slow dynamics or the data record is short.

The filtered variants reduce the likelihood of this happening by reshaping the cost function before the final optimization. Low-pass filtering the data at progressively wider bandwidths produces a sequence of modified criteria, each of which tends to have fewer local minima than the full-bandwidth criterion. Each stage is warm-started from the solution of the previous one, so the optimizer is guided toward a region close to the global minimum before the final unfiltered step. This does not *guarantee* convergence to the global minimum, but it substantially reduces the probability of failure in practice.

The computational cost of a filtered variant is approximately $10\times$ that of the corresponding standard estimator (9 filtered stages plus the final unfiltered run, each requiring a full optimization). Use the standard estimator first; switch to the filtered version when the standard one returns unstable polynomials, produces estimates that vary across runs, or fails to reduce V to a plausible level.