pysib: A Python System Identification toolBox

Diego Eckhard *

* *Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil (e-mail: diegoeck@ufrgs.br)*

Abstract: `pysib` is an open-source Python package for parameter identification of discrete-time SISO systems described by polynomial input–output model structures. It implements classical estimators for ARX, Stieglitz–McBride, instrumental-variable, correlation, output-error, ARMAX, and Box–Jenkins models. All routines share a unified interface and a common polynomial dictionary convention, so models estimated by different algorithms can be evaluated through the same prediction and simulation functions. Linear estimators are implemented in Python using NumPy and SciPy. The nonlinear prediction-error estimators use a specialized conservative optimization core for OE, ARMAX, and Box–Jenkins structures. This optimizer deliberately uses many small parameter updates, rather than relying on a few large steps, to obtain a more controlled search trajectory on non-convex criteria. The large number of error, gradient, and sensitivity evaluations required by this strategy is made practical by compiled C extensions and LAPACK routines for the Gauss–Newton step. The package also provides open implementations of filtered continuation methods for the nonlinear estimators, progressively relaxing data filters across optimization stages to reduce sensitivity to local minima. The aim of this paper is not to introduce a new identification criterion, but to present an open Python implementation that makes robust nonlinear polynomial-model identification and filtered continuation methods directly available to users. Monte Carlo experiments show that the SM and OE estimates produced by `pysib` are tightly centered around the true parameters under additive output noise, and that filtered continuation raises the empirical success rate of the OE estimator from 60 % to 100 % on a non-convex identification problem.

Keywords: system identification, Python, prediction-error method, polynomial models, mathematical software

1. INTRODUCTION

Identifying dynamic models from measured input–output records is a central problem in control, simulation, and signal processing. Among the available model classes, discrete-time polynomial input–output structures—ARX, ARMAX, output-error, and Box–Jenkins—remain widely used because they are interpretable, compact, and well supported by the prediction-error framework Ljung (1999); Söderström and Stoica (1989). Estimating such models from data requires solving linear or nonlinear least-squares problems, and for the nonlinear structures the quality of the estimates also depends on initialization and the optimization trajectory.

The MATLAB System Identification Toolbox and the SIB toolbox provide mature, feature-rich implementations of these classical estimators, but they are either proprietary or tied to a commercial platform The MathWorks, Inc. (2026); Eckhard (2026). In the open Python ecosystem, the SIPPY package offers a broad collection of input–output and state-space routines CPCLAB-UNIPI (2026), while SysIdentPy focuses on nonlinear autoregressive models with exogenous inputs Lacerda et al. (2020). A gap remains for an open-source Python package that is specifically designed for classical polynomial prediction-error structures, with a specialized optimization core, filtered continuation variants, and a unified model convention that allows seamless switching between estimators.

`pysib` fills this gap. It implements the main classical estimators—ARX, Stieglitz–McBride, instrumental variables, correlation error minimization, output error, ARMAX, and Box–Jenkins—together with prediction and simulation routines that share a common polynomial dictionary (keys `A`, `B`, `C`, `D`, `F`). Linear estimators are written in Python with NumPy and SciPy; the nonlinear prediction-error methods use compiled C extensions that interface with LAPACK for the Newton step. Filtered variants of the nonlinear estimators implement a continuation strategy that makes the optimization less sensitive to local minima.

The package makes three contributions whose combination distinguishes it from broader Python system-identification tools. First, it is a focused, open-source, and well-tested toolbox for polynomial input–output structures, with a simple API and a layered Python/C architecture. Second, the nonlinear OE, ARMAX, and BJ estimators employ a conservative two-phase optimizer that deliberately uses many small parameter updates rather than a few aggressive steps; the resulting computational cost of repeated cost, gradient, and sensitivity evaluations is made practical by compiled C extensions and LAPACK linear solvers.

Third, `pysib` provides open implementations of filtered continuation methods Eckhard et al. (2013, 2017) for the nonlinear estimators. These methods have appeared previously in the identification literature, but their availability in a compact Python toolbox makes them easier to use, compare, and reproduce; their effect is evaluated below on a non-convex OE problem.

The remainder of the article is organized as follows. Section 2 defines the prediction-error problem and the four polynomial model structures. Section 3 describes the linear and nonlinear estimators, the specialized optimization core, and the filtered continuation strategy. Section 4 outlines the software architecture and the unified model convention. Section 5 presents Monte Carlo experiments under noisy data and a study of the filtered continuation strategy on a non-convex output-error problem. Section 6 concludes the article.

2. PROBLEM AND MODEL CLASSES

2.1 The Identification Problem

Consider a discrete-time single-input single-output (SISO) dynamical system whose internal structure is unknown. The system is driven by an input $u(t) \in \mathbb{R}$ and produces a true output $y_0(t) \in \mathbb{R}$, where t indexes discrete-time samples. The relationship is dynamic: $y_0(t)$ depends on the current and past values of u, and possibly on past values of the output itself.

An identification experiment applies an input signal $u(t)$ to the system for N time steps. The output is not measured exactly; what is recorded is

$$y(t) = y_0(t) + v(t), \qquad t = 1, \dots, N, \tag{1}$$

where $v(t)$ is an additive disturbance. No specific structure is assumed for $v(t)$ at this stage.

The identification problem is: given the data record $\mathcal{D}_N = \{u(t),\, y(t)\}_{t=1}^{N}$, determine a parametric model $\hat{y}(t;\theta)$—a function of past inputs and outputs, indexed by a finite-dimensional parameter vector θ—such that the prediction errors $\varepsilon(t;\theta) = y(t) - \hat{y}(t;\theta)$ are small in a well-defined sense. The choice of model structure and the criterion for "small" are the subjects of the following sections.

2.2 Model Parameterization

The unknown system is parameterized by the general polynomial model

$$y(t;\theta) = \frac{B(q^{-1};\theta)}{A(q^{-1};\theta)\,F(q^{-1};\theta)}\,u(t-n_k) + \frac{C(q^{-1};\theta)}{A(q^{-1};\theta)\,D(q^{-1};\theta)}\,e(t) \tag{2}$$

where θ is the vector of free polynomial coefficients to be estimated, $e(t)$ is a zero-mean white-noise sequence, q^{-1} is the backward-shift operator ($q^{-1}x(t) = x(t-1)$), and $n_k \geq 0$ is the input delay in samples. The polynomials are monic and defined in non-negative powers of q^{-1}:

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

Table 1. Polynomial model structures supported by `pysib`. A free polynomial is estimated from data; a fixed polynomial is set to unity.

Structure	A	B	C	D	F	Noise model H
ARX	free	free	1	1	1	$1/A$
ARMAX	free	free	free	1	1	C/A
OE	1	free	1	1	free	1
BJ	1	free	free	free	free	C/D

For a given model structure, θ contains the coefficients of the *free* polynomials only (those not fixed to unity). For example, for the BJ structure all five polynomials contribute free coefficients, so $\theta = [b_0, \dots, b_{n_b-1},\, c_1, \dots, c_{n_c},\, d_1, \dots, d_{n_d},\, f_1, \dots$ for ARX, only A and B are free, giving $\theta = [a_1, \dots, a_{n_a},\, b_0, \dots, b_{n_b-}$ The dimension n_θ depends on the chosen orders and structure. The four standard structures are special cases of (2) obtained by fixing some polynomials to unity, as listed in Table 1. The plant transfer function and noise model are, respectively,

$$G(q^{-1};\theta) = \frac{B(q^{-1};\theta)}{A(q^{-1};\theta)\,F(q^{-1};\theta)}, \qquad H(q^{-1};\theta) = \frac{C(q^{-1};\theta)}{A(q^{-1};\theta)\,D(q^{-1}} \tag{3}$$

so (2) reads compactly as $y(t;\theta) = G(q^{-1};\theta)\,u(t-n_k) + H(q^{-1};\theta)\,e(t)$. At the true parameter vector θ_0 this gives $y_0(t) = G(q^{-1};\theta_0)\,u(t-n_k)$ and $v(t) = H(q^{-1};\theta_0)\,e(t)$, making precise the decomposition (1) of Section 2.1. A key structural distinction separates the four models: in ARX and ARMAX, the plant denominator and the noise model share the same polynomial A, so the dynamics of the noise and the plant are coupled; in OE and Box–Jenkins, the plant denominator F and the noise model C/D are independent, allowing the two components to be parameterized separately.

Given the polynomial assignments in Table 1, the one-step-ahead predictor for the general structure (2) is

$$\hat{y}(t \mid t-1;\theta) = y(t) + H(q^{-1};\theta)^{-1}\big[G(q^{-1};\theta)\,u(t-n_k) - y(t)\big], \tag{4}$$

where $H(q^{-1};\theta)^{-1} = A(q^{-1};\theta)D(q^{-1};\theta)/C(q^{-1};\theta)$. Substituting the structure assignments into (4) yields the structure-specific predictor. For the ARX structure,

$$\hat{y}(t \mid t-1;\theta) = \sum_{j=0}^{n_b-1} b_j\,u(t-n_k-j) - \sum_{i=1}^{n_a} a_i\,y(t-i), \tag{5}$$

which is linear in θ. For the OE, ARMAX, and Box–Jenkins structures:

$$\hat{y}(t \mid t-1;\theta) = \frac{B(q^{-1};\theta)}{F(q^{-1};\theta)}\,u(t-n_k), \tag{6}$$

$$\hat{y}(t \mid t-1;\theta) = \frac{B(q^{-1};\theta)}{C(q^{-1};\theta)}\,u(t-n_k) + \frac{C(q^{-1};\theta) - A(q^{-1};\theta)}{C(q^{-1};\theta)}\,y(t), \tag{7}$$

$$\hat{y}(t \mid t-1;\theta) = y(t) + \frac{D(q^{-1};\theta)}{C(q^{-1};\theta)}\left[\frac{B(q^{-1};\theta)}{F(q^{-1};\theta)}\,u(t-n_k) - y(t)\right], \tag{8}$$

Thus, for Box–Jenkins models, prediction is evaluated by filtering the plant-output mismatch through the inverse noise model $D(q^{-1};\theta)/C(q^{-1};\theta)$ and adding the result to the measured output.

2.3 Parameter Estimation

The prediction-error method (PEM) estimates θ by minimizing the mean squared prediction error Ljung (1999); Söderström and Stoica (1989)

$$V(\theta) = \frac{1}{N}\sum_{t=1}^{N} \varepsilon(t;\theta)^2, \tag{9}$$

where $\varepsilon(t;\theta) = y(t) - \hat{y}(t \mid t-1;\theta)$ is the one-step-ahead prediction error. For the ARX predictor (5), writing $\hat{y}(t \mid t-1;\theta) = \varphi(t)^\top\theta$ with the regression vector

$$\varphi(t) = \big[-y(t-1),\dots,-y(t-n_a),\, u(t-n_k),\dots,u(t-n_k-n_b+1)\big]^\top, \tag{10}$$

and stacking N rows into $\Phi \in \mathbb{R}^{N\times(n_a+n_b)}$ gives $V(\theta) \propto \|\Phi\theta - y\|^2$, whose minimum is the ordinary least-squares (OLS) solution $\hat{\theta} = (\Phi^\top\Phi)^{-1}\Phi^\top y$.

The gradient of (9) with respect to θ is determined by the *sensitivity matrix* $\Psi(\theta) \in \mathbb{R}^{N\times n_\theta}$, whose j-th column collects

$$\psi_j(t;\theta) = \frac{\partial \hat{y}(t \mid t-1;\theta)}{\partial \theta_j}. \tag{11}$$

Since $\varepsilon(t;\theta) = y(t) - \hat{y}(t \mid t-1;\theta)$, the chain rule gives

$$\frac{\partial V}{\partial \theta} = -\frac{2}{N}\,\Psi(\theta)^\top \varepsilon(\theta), \tag{12}$$

where $\varepsilon(\theta) = [\varepsilon(1;\theta),\dots,\varepsilon(N;\theta)]^\top$. For the ARX structure, $\psi_j(t;\theta) = \varphi_j(t)$ (the j-th entry of the regression vector (10)), and (12) vanishes exactly at the OLS solution. For the OE, ARMAX, and Box–Jenkins structures, ψ_j requires filtering through $1/F(q^{-1})$ or $1/C(q^{-1})$—polynomials that depend on θ itself—making Ψ a nonlinear function of θ and (9) non-convex in general; the existence of multiple local minima for these structures is well documented Eckhard and Bazanella (2011); Eckhard et al. (2012). The Gauss–Newton approximation to the Hessian,

$$\frac{\partial^2 V}{\partial \theta^2} \approx \frac{2}{N}\,\Psi^\top\Psi, \tag{13}$$

retains only the outer-product term (dropping second-order sensitivity terms that vanish near the minimum). It is symmetric positive semidefinite by construction; when the sensitivity matrix Ψ has full column rank the Gram matrix $\Psi^\top\Psi$ becomes positive definite and the Gauss–Newton direction is well defined and points downhill. Under standard regularity conditions on the input and noise, global minimizers of the PEM criterion are consistent when the true system belongs to the model class Ljung (1999); the implemented nonlinear routines seek such minima through local optimization, so the numerical outcome also depends on initialization and the optimizer.

Three estimators in `pysib` identify an ARX-structure model without minimizing $V(\theta)$, avoiding the iterative nonlinear optimization altogether.

Stieglitz–McBride (SM). The SM method Stieglitz and McBride (1965) approximates the OE solution by replacing the single nonlinear problem with a sequence of linear ones. At each iteration k, the current denominator estimate $\hat{F}^{(k)}(q^{-1})$ is used to filter both signals: $u_f(t) = u(t)/\hat{F}^{(k)}(q^{-1})$ and $y_f(t) = y(t)/\hat{F}^{(k)}(q^{-1})$. An ARX model is then estimated on the filtered pair (u_f, y_f), and the new $\hat{F}^{(k+1)}$ is taken from the feedback polynomial of that ARX solution. The procedure is repeated for a fixed number of iterations (100 in `pysib`). Although a fixed point of this iteration satisfies the same normal equations as the OE solution, the method is not guaranteed to reach the global minimum of $V(\theta)$ Ljung (1999).

Instrumental Variables (IV). The IV method estimates an ARX model consistently even when the noise is not of the form $1/A(q^{-1};\theta)\,e(t)$, at the cost of requiring two independent experiments driven by the same input $u(t)$. Let $y_i(t) = y_0(t) + e_i(t)$, $i = 1, 2$, denote the noisy outputs, where $y_0(t)$ is the true plant output and $e_1(t)$, $e_2(t)$ are independent noise sequences. Define the regression matrix $\Phi_1 \in \mathbb{R}^{N\times(n_a+n_b)}$ with rows

$$\varphi_1(t)^\top = \big[-y_1(t-1),\dots,-y_1(t-n_a),\, u(t-n_k),\dots,u(t-n_k-n_b+1)\big]$$

and the instrument matrix $Z \in \mathbb{R}^{N\times(n_a+n_b)}$ with rows

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

where the lagged y_1 entries of φ_1 are replaced by the corresponding lagged y_2 entries. The IV estimate is

$$\hat{\theta} = (Z^\top\Phi_1)^{-1}Z^\top y_1. \tag{14}$$

Because Z shares the input-driven part of Φ_1 but $Z^\top e_1 \to 0$ as $N \to \infty$ (since e_2 and e_1 are independent), $\hat{\theta}$ is consistent Söderström and Stoica (1989).

Correlation method. The correlation method finds θ by requiring the prediction error $\varepsilon(t;\theta)$ to be uncorrelated with an instrument $z(t)$ at M lags:

$$\frac{1}{N}\sum_{t} z(t-\tau)\,\varepsilon(t;\theta) = 0, \qquad \tau = 0, 1, \dots, M-1. \tag{15}$$

Since $\varepsilon(t;\theta)$ is linear in θ for the ARX structure, (15) constitutes a system of M linear equations. For $M = n_a + n_b$ the system is exactly determined; for $M > n_a + n_b$ it is overdetermined and solved by least squares, which minimizes the sum of squared cross-correlations. The default instrument is $z = u$, which is valid in open-loop experiments where the input is uncorrelated with the noise.

These three estimators serve a dual role in the package. They can be applied directly when their respective conditions are satisfied. They also provide the initial parameter vector for every nonlinear PEM estimator in `pysib`: each nonlinear method is warm-started from an ARX solution, so the iterative optimizer begins from a meaningful point rather than an arbitrary initial guess.

3. ALGORITHMS

3.1 Linear Estimators

ARX The ARX estimator is the fundamental linear routine in the package and the warm-start for every nonlinear method. The ARX predictor (5) is linear in θ: $\hat{y}(t \mid t-1;\theta) = \varphi(t)^\top\theta$, where $\varphi(t)$ is the regression vector (10). Stacking N rows into the regression matrix $\Phi \in \mathbb{R}^{N\times n_\theta}$ and the outputs into $y = [y(1),\dots,y(N)]^\top \in \mathbb{R}^N$, the PEM criterion (9) reduces to

$$V(\theta) = \frac{1}{N}\,\|\Phi\theta - y\|^2,$$

Algorithm 1. Stieglitz–McBride

Require: u, y, orders n_b, n_f, n_k, iterations $K = 100$

1: $\hat{\theta} \leftarrow \texttt{arx}(u, y, n_f, n_b, n_k)$ ▷ ARX warm-start: $\hat{F}^{(0)}$ from $\hat{A}$
2: **for** $k = 1, \ldots, K$ **do**
3: $\quad u_f \leftarrow u \,/\, \hat{F}^{(k-1)}(q^{-1}; \theta)$
4: $\quad y_f \leftarrow y \,/\, \hat{F}^{(k-1)}(q^{-1}; \theta)$
5: $\quad \hat{\theta} \leftarrow \texttt{arx}(u_f, y_f, n_f, n_b, n_k)$ ▷ OLS on filtered pair
6: $\quad \hat{F}^{(k)} \leftarrow \hat{A}$ from $\hat{\theta}$
7: **end for**
8: **return** $\hat{\theta}$

which is a convex quadratic function of θ (the Hessian $\frac{2}{N}\Phi^\top\Phi$ is positive semidefinite). Differentiating and setting $\partial V/\partial\theta = 0$ yields the *normal equations*

$$\Phi^\top\Phi\,\hat{\theta} = \Phi^\top y.$$

The solution is unique if and only if Φ has full column rank, i.e., $\Phi^\top\Phi$ is invertible, in which case $\hat{\theta} = (\Phi^\top\Phi)^{-1}\Phi^\top y$ is the unique global minimum of $V(\theta)$. Full column rank is obtained in open-loop identification when the input $u(t)$ is *persistently exciting* of order n_θ: informally, the input must contain enough distinct frequency components to excite all modes of the regression matrix Ljung (1999). When this condition fails (e.g., a constant or insufficiently rich input), Φ is rank-deficient and $V(\theta)$ has infinitely many minimizers forming an affine subspace.

Computing $(\Phi^\top\Phi)^{-1}$ directly is numerically inadvisable even in the full-rank case: the condition number satisfies $\kappa(\Phi^\top\Phi) = \kappa(\Phi)^2$, so any ill-conditioning in Φ is squared, amplifying rounding errors. `pysib` instead calls `numpy.linalg.lstsq`, which invokes the LAPACK driver `dgelsd`. This routine solves the least-squares problem via the divide-and-conquer SVD of Φ directly, avoiding the normal equations entirely. When Φ is rank-deficient, `dgelsd` returns the minimum-norm solution among all minimizers of $V(\theta)$.

Stieglitz–McBride The OE predictor $\hat{y}(t \mid t-1;\theta) = B(q^{-1};\theta)/F(q^{-1};\theta)\,u(t-n_k)$ is nonlinear in the F coefficients, making $V(\theta)$ non-convex. The SM method approximates the OE solution by replacing the single nonlinear problem with a sequence of linear ARX problems.

At each iteration k, the current denominator estimate $\hat{F}^{(k)}(q^{-1};\theta)$ is used to prefilter both signals:

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

An ARX model of orders (n_f, n_b) is then estimated on the filtered pair (u_f, y_f) using the procedure of Section 3.1.1, and the new denominator estimate $\hat{F}^{(k+1)}$ is taken from the feedback polynomial of that ARX solution. At a fixed point $\hat{F}^{(k+1)} = \hat{F}^{(k)}$, the ARX normal equations on the filtered data reduce to the OE optimality conditions, so the fixed point is an OE solution. However, convergence to the global minimum is not guaranteed Ljung (1999); the result depends on the initialization. `pysib` initializes $\hat{F}^{(0)}$ from a plain ARX solution on the unfiltered data.

The IV and correlation estimators follow the constructions of Section 2.3 and are implemented in Python using NumPy.

Instrumental Variables The IV estimator solves the linear system $(Z^\top\Phi_1)\,\hat{\theta} = Z^\top y_1$, where Φ_1 and Z are the regression and instrument matrices defined in Section 2.3. The system is solved by least squares via `numpy.linalg.lstsq`. The method requires two independent experiments driven by the same input $u(t)$, and it is consistent when the noise sequences in the two experiments are uncorrelated with each other and with the input. In `pysib`, the IV routine accepts a single pair (u, y_1) and a second output record y_2; when the two outputs are identical, the method reduces to ordinary least-squares ARX.

Correlation The correlation estimator assembles the $M \times (n_a + n_b)$ matrix of cross-correlations between the instrument $z(t)$ and the lagged signals that appear in the ARX predictor. The default instrument is $z = u$, which is valid in open-loop experiments where the input is uncorrelated with the noise. When $M = n_a + n_b$, the linear system (15) is exactly determined; when $M > n_a + n_b$, it is overdetermined and solved via `numpy.linalg.lstsq`, which finds the least-squares solution that minimizes the sum of squared cross-correlations. The implementation follows the same pattern as ARX: the regressor matrix is assembled once and the result returned without iteration.

3.2 Nonlinear Prediction-Error Estimators and Optimization Core

The OE, ARMAX, and Box–Jenkins estimators minimize the nonlinear prediction-error criterion $V(\theta)$ using the gradient (12) and the Gauss–Newton Hessian (13). Since the filters in these structures depend on the parameters, the resulting criteria are generally non-convex. For this reason, `pysib` uses a specialized optimization core rather than treating the estimators as generic black-box minimization problems. The core combines structure-specific sensitivity recursions with a conservative search strategy based on many small parameter updates, followed by Gauss–Newton refinement. This design increases the number of error, gradient, and sensitivity evaluations, and the performance-critical routines are therefore implemented as compiled C extensions; the Newton system is solved through LAPACK. Each estimator is initialized from an ARX solution; noise-model polynomials (C, D) are initialized to zero free coefficients.

Structure-specific sensitivities. The nonlinear estimators use analytic, structure-specific sensitivity recursions rather than numerical differentiation. Separate C modules are used for OE, ARMAX, and Box–Jenkins because the free polynomials and the filters appearing in the sensitivity equations differ across structures. For the OE predictor $\hat{y}(t) = B(q^{-1})/F(q^{-1})\,u(t-n_k)$, for example, the sensitivities (11) are

$$\psi_{b_j}(t) = \frac{q^{-j}\, u(t-n_k)}{F(q^{-1})}, \tag{16}$$

$$\psi_{f_j}(t) = -\frac{q^{-j}\, \hat{y}(t)}{F(q^{-1})}. \tag{17}$$

For ARMAX the denominator filter is $1/C(q^{-1})$; for Box–Jenkins both $1/F(q^{-1})$ and $1/C(q^{-1})$ appear, together with the corresponding noise-model polynomials. Thus each C module implements the sensitivity recursions for its own polynomial structure.

Reuse of filtered sensitivity sequences. The C implementation also exploits the shift structure of polynomial models. Many sensitivity columns differ only by a delay. In these cases, the code first computes a filtered base sequence and then forms the corresponding columns by time shifts, instead of applying the same IIR filter separately for every coefficient. In the OE case, a single pass through $1/F(q^{-1})$ applied to the delayed input provides the base sequence for all numerator sensitivities, while a single pass applied to $-\hat{y}$ provides the base sequence for the denominator sensitivities; the columns associated with b_j and f_j are obtained by indexing these sequences with the appropriate lag. The same principle is used in the ARMAX and Box–Jenkins modules with their respective filters. This reuse is important because the optimizer evaluates the cost, gradient, and Gauss–Newton matrices many times along a single identification run.

Selective cost, gradient, and Hessian evaluation. The optimization core distinguishes between three levels of computation. When only a candidate parameter vector must be accepted or rejected, it evaluates the cost $V(\theta)$ without forming sensitivities. During the initial descent phase, it evaluates the cost and gradient but does not build the Gauss–Newton matrix. During the Newton phase, it evaluates the cost, gradient, and the matrix $\Psi^\top\Psi$ needed in (18). This separation avoids unnecessary gradient or Hessian assembly during the many small trial steps used by the optimizer. For scalar cost comparisons the C core internally uses the root-mean-square prediction error $\sqrt{V(\theta)}$, which does not change minimizers or accept/reject ordering because it is strictly increasing in V; the gradient and Gauss–Newton quantities retain the standard least-squares scaling and are unaffected by a constant factor in the cost.

Two-phase optimizer. *Phase 1* is a conservative smoothed steepest-descent loop intended to stabilize the trajectory from the ARX warm start before local Newton refinement. At each of up to 100×100 gradient evaluations, the gradient $g = \partial V/\partial\theta$ (12) is accumulated by an exponential moving average, $d \leftarrow (4d + g)/5$, which damps iteration-to-iteration changes in the descent direction. The direction is normalized to unit length, separating the direction of motion from the step size, and the parameter vector is updated by $\theta \leftarrow \theta - \alpha\, d/\|d\|$. The step size α adapts slowly and multiplicatively: $\alpha \leftarrow 1.01\,\alpha$ on a cost decrease and $\alpha \leftarrow 0.99\,\alpha$ on an increase, starting from $\alpha = 10^{-5}$. The phase terminates early when $\alpha < 10^{-7}$.

Phase 2 performs local Gauss–Newton refinement using the structured sensitivities computed by the C module. At each of 1000 iterations the Newton direction δ is obtained by solving

$$(\Psi^\top\Psi)\,\delta = -\Psi^\top\varepsilon \tag{18}$$

via the LAPACK routine `dposv`, which solves the Gauss–Newton normal system by Cholesky factorization under the assumption of positive definiteness. The candidate $\theta \leftarrow \theta - \alpha_k\,\delta$ uses a growing step $\alpha_k = (k+1)/1000$, ranging from 0.002 at the first iteration to approximately 1 at the last, so full Newton steps are approached gradually. If the candidate does not reduce the cost or is non-finite, a bisection safeguard is applied up to ten times: $\theta^{\text{new}} \leftarrow (\theta^{\text{new}} + \theta)/2$.

3.3 Filtered Continuation Estimators

Nonlinear prediction-error criteria may have multiple local minima, and even a conservative optimizer may converge to a nonglobal stationary point. The filtered estimators in `pysib` provide open implementations of the continuation and cost-function-shaping strategies developed for this setting Eckhard et al. (2013, 2017). At each continuation stage, filtering the input–output data modifies the shape of the cost function; progressively relaxing the filter then drives the estimate toward the original unfiltered criterion. Filtered variants are provided for the OE, ARMAX, and Box–Jenkins estimators; in each case the final stage runs the corresponding unfiltered estimator on the original PEM criterion.

Stage sequence and warm-starting. The procedure runs nine filtered optimization stages followed by one final stage on the unfiltered data. Each stage calls the same nonlinear optimization core used by the corresponding unfiltered estimator, but on a filtered version of the input–output record. The solution returned by one stage is used as the initial condition for the next stage, so the continuation path begins from a strongly shaped criterion and ends at the original PEM problem. The total computational cost is approximately ten times that of the corresponding unfiltered estimator.

Filter schedules. For the OE structure, each stage uses a first-order IIR filter $(1 - a_i)/(1 - a_i q^{-1})$ with unit DC gain, whose pole is $a_i = 0.05^{1/\tau_i}$ with $\tau_i = 4(10 - i)$ for stages $i = 1, \ldots, 9$. The pole decreases from $a_1 \approx 0.920$ (aggressive low-pass, stage 1) to $a_9 \approx 0.473$ (mild low-pass, stage 9), progressively restoring high-frequency content.

For the ARMAX and Box–Jenkins structures, each stage uses a first-order Butterworth filter with normalized cutoff $\omega_i = 0.1\,i$ (as a fraction of the Nyquist frequency), $i = 1, \ldots, 9$. Stage 1 passes only the lowest 10 % of the spectrum; stage 9 passes 90 %. In both schedules the final stage is the original unfiltered estimator applied to the same PEM criterion; the only difference between the standard and filtered methods is the initial condition—the standard method starts from an ARX estimate, while the filtered method starts from the solution of the previous filtered stage.

Table 2. Main software components of `pysib`.

Component	Implementation	Role
ARX, SM, IV, Correlation	Python, NumPy, SciPy	Linear estimators
OE, ARMAX, BJ	C extension, LAPACK	Nonlinear prediction-error estimators
Filtered variants	Python over C optimizer	Continuation strategy
Prediction and simulation	SciPy `lfilter`	Model evaluation

The numerical experiments in Section 5 focus on the OE structure because the theoretical analysis of cost-function shaping was developed specifically for the output-error case Eckhard et al. (2013, 2017). The same continuation strategy is implemented for ARMAX and Box–Jenkins in `pysib` under the expectation that the mechanism of progressively relaxing data filters should also improve convergence in those structures, although formal guarantees have not been established for them.

4. SOFTWARE ORGANIZATION

Python/C separation. The package separates Python-level orchestration from performance-critical compiled routines. Linear estimators (ARX, SM, IV, correlation) are implemented in Python using NumPy and SciPy. The nonlinear PEM optimizers are compiled as three separate C extension modules (`sib_oe_module.c`, `sib_armax_module.c`, `sib_bj_module.c`), each providing the structure-specific sensitivity formulas and sharing the common two-phase optimizer from `sib_optimize.c`. The Newton step in Phase 2 links against LAPACK (`dposv`) without requiring an external system identification framework. The filtered variants are Python wrappers that sequence the filter stages and delegate each optimization to the corresponding C module.

Unified model convention. Every estimation routine returns a pair $(\hat{\theta}, m)$. The vector $\hat{\theta}$ contains the free parameter estimates in structure-specific order, and the dictionary m stores the polynomial coefficient vectors under the keys `A`, `B`, `C`, `D`, and `F` (fixed polynomials contain only [1]). The `predict` and `simulate` routines share the same dictionary convention, so any estimated model can be evaluated through a common model representation.

Article/manual boundary. This article describes the algorithms and the high-level organization of the software. Installation instructions, dependency versions, complete API documentation, callable-procedure examples, and expected driver outputs are provided in the accompanying user manual, as required for the CALGO software component.

Table 2 summarizes the main components and their implementations.

Table 3. Mean and standard deviation of the OE-structure parameter estimates under noisy data ($M = 100$).

Method	Parameter	Mean	Std
SM	b_1 (true 1)	1.0005	0.0030
SM	f_1 (true -2.4)	-2.3997	0.0018
SM	f_2 (true 1.91)	1.9094	0.0033
SM	f_3 (true -0.504)	-0.5037	0.0015
OE	b_1 (true 1)	1.0005	0.0030
OE	f_1 (true -2.4)	-2.3997	0.0018
OE	f_2 (true 1.91)	1.9094	0.0033
OE	f_3 (true -0.504)	-0.5037	0.0015
SIPPY	b_1 (true 1)	1.0439	0.0586
SIPPY	f_1 (true -2.4)	-2.3731	0.0342
SIPPY	f_2 (true 1.91)	1.8604	0.0630
SIPPY	f_3 (true -0.504)	-0.4810	0.0292

5. NUMERICAL RESULTS

The accompanying software artifact includes deterministic unit tests for noise-free exact recovery and for the polynomial convention of Section 2. These tests cover all public estimators—ARX, SM, IV, correlation, OE, ARMAX, BJ, and the filtered variants—on low-order systems with known parameters. They are part of the software validation and are not presented as scientific results. The numerical experiments reported here focus instead on the behavior of the estimators under noisy data and on the effect of filtered continuation in non-convex identification problems.

5.1 Robustness under Noisy Data

The first experiment evaluates parameter accuracy and simulation quality under additive output noise. The true system is a third-order OE plant with $B(q^{-1}) = q^{-1}$ and $F(q^{-1}) = 1 - 2.4q^{-1} + 1.91q^{-2} - 0.504q^{-3}$, corresponding to poles at 0.9, 0.8, and 0.7. The plant is driven by a three-sine input of length $N = 1000$. White Gaussian noise with standard deviation $\sigma_v = 1$ is added to the output, and $M = 100$ independent realizations are used. The SM and OE estimators in `pysib` use the correct orders $(n_b, n_f, n_k) = (1, 3, 1)$. To place the implementation in context, the same Monte Carlo experiment is also run with the OE routine from SIPPY using the same data and model order.

Table 3 reports the mean and sample deviation of the OE-structure parameter estimates. The SM and OE estimates produced by `pysib` are tightly centered around the true parameters. SIPPY also returns stable models on this example, but with visibly larger parameter bias and dispersion. The notation b_1 refers to the first non-zero B coefficient after the delay $n_k = 1$, following the definitions in Section 2. Figure 1 shows the corresponding histograms per parameter. Table 4 reports the relative simulation error $\|y_0 - \hat{y}\|_2 / \|y_0\|_2$ on the noise-free output. All methods satisfy the 5 % success threshold in every run, while the `pysib` SM and OE estimates yield an order-of-magnitude smaller mean simulation error in this experiment. The corresponding simulation drivers and expected outputs are included in the software artifact.

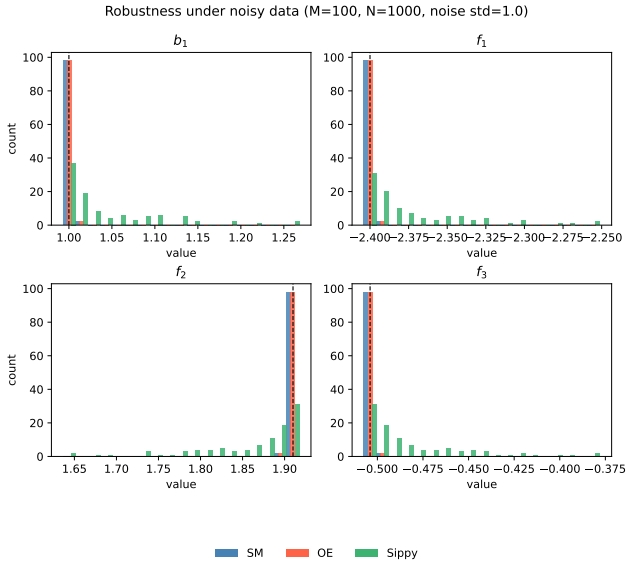

Fig. 1. Monte Carlo histograms of the OE-structure parameter estimates for SM, OE, and SIPPY. Dashed vertical lines mark true values.

Table 4. Relative simulation error under noisy data ($M = 100$).

Method	Mean error	Std error	Success rate ($< 5\%$)
SM	1.05×10^{-3}	3.88×10^{-4}	100%
OE	1.05×10^{-3}	3.89×10^{-4}	100%
SIPPY	9.71×10^{-3}	8.97×10^{-3}	100%

5.2 Effect of Filtered Continuation

The second experiment tests whether the filtered continuation strategy improves the convergence of the OE estimator when the PEM criterion has multiple local minima. The true system is a third-order OE plant with $B(q^{-1}) = q^{-1}$ and $F(q^{-1}) = 1 - 2.4q^{-1} + 1.91q^{-2} - 0.504q^{-3}$, corresponding to poles at 0.9, 0.8, and 0.7. The input is a three-sine signal of length $N = 1000$, and white Gaussian noise with $\sigma_v = 30$ is added to the output, yielding a signal-to-noise ratio of approximately $-6\,\mathrm{dB}$. A Monte Carlo of $M = 100$ independent realizations is performed; both estimators use ARX-based initializations and the same optimization core (Section 3.2): the standard OE method proceeds directly to the unfiltered criterion, whereas the filtered variant first follows the filtered continuation path before its final unfiltered stage.

Quality is measured by the relative simulation error $\|y_0 - \hat{y}\|_2 / \|y_0\|_2$ on the noise-free output, and a run is considered successful if this error is below 5 %, a typical practical tolerance for model-based control applications. Table 5 reports the mean and standard deviation of the error for both methods, as well as the success rate. The standard OE estimator succeeds in 60 % of the Monte Carlo runs (95 % Wilson confidence interval [50 %, 69 %]), with a mean error of 5.7×10^{-2}. The filtered variant succeeds in 100 % of the runs and reduces the mean error to 2.2×10^{-2}. Figure 2 shows the error distribution and a boxplot of both estimators. The tenfold increase in computational cost due to the nine filtered stages is the price paid for achieving a 100 % empirical success rate in this problem.

Table 5. Relative simulation error of OE and OE filtered on a non-convex OE identification problem ($M = 100$).

Method	Mean error	Std error	Success rate ($< 5\%$)
OE	5.69×10^{-2}	4.37×10^{-2}	60%
OE filtered	2.17×10^{-2}	7.94×10^{-3}	100%

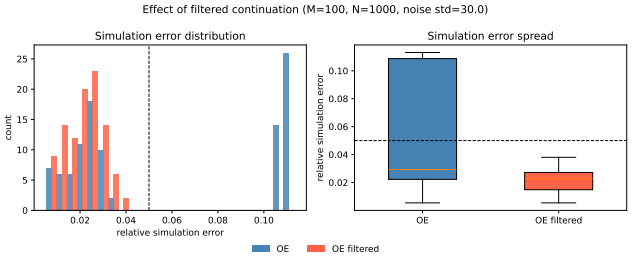

Fig. 2. Distribution of the relative simulation error for OE and OE filtered on the non-convex problem. Dashed line marks the 5% success threshold.

6. CONCLUSION

`pysib` is an open-source Python package for parameter identification of discrete-time SISO systems using classical polynomial input–output model structures. It provides a unified interface to ARX, Stieglitz–McBride, instrumental-variable, correlation, output-error, ARMAX, and Box–Jenkins estimators, all sharing a common polynomial dictionary convention so that prediction and simulation can be applied uniformly.

The package makes three main contributions. First, it fills a gap in the open Python ecosystem by delivering a focused, well-tested implementation of the key linear and nonlinear prediction-error estimators, with a simple API and a layered Python/C architecture. Second, the nonlinear OE, ARMAX, and BJ estimators use a specialized low-level optimization core that favors a conservative trajectory of many small updates over aggressive Newton-type steps; the resulting workload of cost, gradient, and sensitivity evaluations is made practical by compiled C extensions and LAPACK linear solvers. Third, the toolbox provides open implementations of filtered continuation methods Eckhard et al. (2013, 2017) for the nonlinear estimators; these methods progressively relax data filters to drive the estimate toward the original PEM criterion, substantially reducing sensitivity to local minima.

Numerical experiments show that, under noisy data, the SM and OE implementations in `pysib` produce parameter estimates tightly centered around the true OE model and lower simulation error than SIPPY in the tested setting. A second experiment shows that the filtered continuation strategy raises the success rate on a non-convex OE problem from 60 % to 100 %. The complete source code, unit tests, and reproducible drivers are available through the Collected Algorithms of the ACM and at a public software archive.

REFERENCES

CPCLAB-UNIPI (2026). SIPPY: Systems Identification Package for PYthon. Online. URL `https://github.`

com/CPCLAB-UNIPI/SIPPY. Accessed: 2026-05-25.

Eckhard, D. and Bazanella, A.S. (2011). On the global convergence of identification of output error models. *IFAC Proceedings Volumes*, 44(1), 9058–9063. doi:10.3182/20110828-6-IT-1002.03566. URL https://doi.org/10.3182/20110828-6-IT-1002.03566.

Eckhard, D., Bazanella, A.S., Rojas, C.R., and Hjalmarsson, H. (2012). On the convergence of the prediction error method to its global minimum. *IFAC Proceedings Volumes*, 45(16), 698–703. doi:10.3182/20120711-3-BE-2027.00371. URL https://doi.org/10.3182/20120711-3-BE-2027.00371.

Eckhard, D., Bazanella, A.S., Rojas, C.R., and Hjalmarsson, H. (2013). Input design as a tool to improve the convergence of pem. *Automatica*, 49(11), 3282–3291. doi:10.1016/j.automatica.2013.08.027. URL https://doi.org/10.1016/j.automatica.2013.08.027.

Eckhard, D., Bazanella, A.S., Rojas, C.R., and Hjalmarsson, H. (2017). Cost function shaping of the output error criterion. *Automatica*, 76, 53–60. doi:10.1016/j.automatica.2016.10.015. URL https://doi.org/10.1016/j.automatica.2016.10.015.

Eckhard, D. (2026). SIB – System Identification Toolbox. Online. URL https://professor.ufrgs.br/diego/software/slow-better. Accessed: 2026-05-25.

Lacerda, W.R., da Andrade, L.P.C., Oliveira, S.C.P., and Martins, S.A.M. (2020). SysIdentPy: A Python package for System Identification using NARMAX models. *Journal of Open Source Software*, 5(54), 2384. doi:10.21105/joss.02384. URL https://doi.org/10.21105/joss.02384.

Ljung, L. (1999). *System Identification: Theory for the User*. Prentice Hall, Upper Saddle River, NJ, 2 edition.

Söderström, T. and Stoica, P. (1989). *System Identification*. Prentice Hall, Englewood Cliffs, NJ.

Stieglitz, K. and McBride, L.E. (1965). A technique for the identification of linear systems. *IEEE Transactions on Automatic Control*, 10(4), 461–464. doi:10.1109/TAC.1965.1098181.

The MathWorks, Inc. (2026). System Identification Toolbox. Online. URL https://www.mathworks.com/products/sysid.html. Accessed: 2026-05-25.