EFFICIENT NUMERICAL OPTIMIZATION METHODS FOR LEARNING NONLINEAR STATE-SPACE MODELS

Alberto Bemporad

imt.lu/ab

Funded by the European Union

European Research Council
Established by the European Commission

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

ODYS Advanced Controls & Optimization

33rd ERNSI Workshop on System Identification - Bordeaux, September 24, 2025

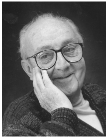
CONTENTS OF MY TALK

1. Optimization methods for **learning nonlinear state-space models**
2. **jax-sysid** : A Python package for nonlinear system identification
3. **Concurrent learning** of nonlinear models and **control invariant sets**
4. Learning (static) **parametric convex functions** from data
5. Learning **combined process and noise models** in nonlinear state-space form

LEARNING CONTROL-ORIENTED NONLINEAR MODELS

"All models are wrong, but some are useful."

(George E. P. Box)



CONTROL-ORIENTED MODELS

- A **complex model** implies a **complex model-based controller**
- We typically look for **small-scale models** (e.g., ≤ 10 states/inputs/outputs) with a **limited number of coefficients** (<1k params vs >300B of LLMs)
- **Limit nonlinearities** as much as possible (e.g., avoid very deep neural networks)
- Need to get the **best model** within a **poor model class** from a **rich dataset** (= limited risk of overfit, under proper excitation)
- **Computation constraints**: solve the learning problem using limited resources (=our laptop, no supercomputing infrastructures)

Learning control-oriented models of dynamical systems requires different algorithms than typical machine learning tasks

NONLINEAR SYS-ID BASED ON NEURAL NETWORKS

- **Neural networks** proposed for nonlinear system identification since the '90s

  (Narendra, Parthasarathy, 1990) (Hunt et al., 1992) (Suykens, Vandewalle, De Moor, 1996)

- **NNARX** models: use a **feedforward neural network** (FNN) to approximate the nonlinear difference equation $y_t \approx \mathcal{N}(y_{t-1}, \ldots, y_{t-n_a}, u_{t-1}, \ldots, u_{t-n_b})$

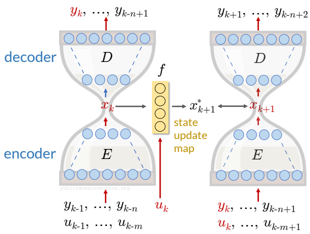
- **Neural state-space** models:

  - **w/ state data**: fit a neural network model

    $x_{t+1} \approx \mathcal{N}_x(x_t, u_t), \quad y_t \approx \mathcal{N}_y(x_t)$

  - **I/O data only**:

    - x_t = inner layer of a FNN (Prasad, Bequette, 2003)
    - **Autoencoders** (Masti, Bemporad, 2021)
    - **SUBNETS** (Beintema, Schoukens, Tóth, 2023)



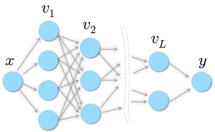
- Usually minimize the **open-loop prediction error** to get a good prediction model

RECURRENT NEURAL NETWORKS

- **Recurrent Neural Network** (RNN) model:

$$\begin{aligned} x_{k+1} &= f_x(x_k, u_k, \theta_x) \\ y_k &= f_y(x_k, \theta_y) \\ f_x, f_y &= \text{feedforward neural network} \end{aligned}$$

(e.g.: general RNNs, LSTMs, RESNETS, physics-informed NNs, ...)



$$v_j = W_j f_{j-1}(v_{j-1}) + b_j$$

$$\theta = (W_1, b_1, \ldots, W_L, b_L)$$

- **Training problem**: given an I/O dataset $\{u_0, y_0, \ldots, u_{N-1}, y_{N-1}\}$ solve

$$\begin{aligned} \min_{\substack{\theta_x, \theta_y \\ x_0, x_1, \ldots, x_{N-1}}} \quad & r(x_0, \theta_x, \theta_y) + \frac{1}{N} \sum_{k=0}^{N-1} \ell(y_k, f_y(x_k, \theta_y)) \\ \text{s.t.} \quad & x_{k+1} = f_x(x_k, u_k, \theta_x) \end{aligned}$$

- **Main issue:** x_k are **hidden states** and hence also **unknowns** of the problem

OPTIMIZATION METHODS FOR TRAINING RNNS

- **Problem condensing**: substitute $x_{k+1} = f_x(x_k, u_k, \theta_x)$ recursively and solve

$$\min_{\theta_x,\theta_y,x_0} r(x_0,\theta_x,\theta_y) + \frac{1}{N}\sum_{k=0}^{N-1} \ell(y_k, f_y(x_k,\theta_y)) = \min_{\theta_x,\theta_y,x_0} V(\theta_x,\theta_y,x_0)$$

- **Gradient descent** (**GD**) methods: update θ_x, θ_y, x_0 by setting

$$\begin{bmatrix} {\theta_x}^{t+1} \\ {\theta_y}^{t+1} \\ x_0^{t+1} \end{bmatrix} = \begin{bmatrix} {\theta_x}^{t} \\ {\theta_y}^{t} \\ x_0^{t} \end{bmatrix} - \alpha_t \nabla V({\theta_x}^t, {\theta_y}^t, x_0^t)$$

  Example: **Adam** uses adaptive moment estimation to set the learning rate α_t

  (Kingma, Ba, 2015)

- **Main issue** with GD methods: **slow convergence** (in theory and in practice)

OPTIMIZATION METHODS FOR TRAINING RNNS

- **Stochastic** gradient descent (SGD) can be even less efficient with RNNs:
  - collect a high number of short independent experiments (often impossible)
  - or create mini-batches by using **multiple-shooting** ideas

    (Forgione, Piga, 2020) (Bemporad, 2023)

- **Newton's method**: very fast ($2^{\rm nd}$-order) local convergence but difficult to implement, as we need the **Hessian** $\nabla^2 V({\theta_x}^t, {\theta_y}^t, x_0^t)$

- **Quasi-Newton methods**: good tradeoff between convergence speed / solution quality and numerical complexity. Only requires the **gradient** $\nabla V({\theta_x}^t, {\theta_y}^t, x_0^t)$

- **Extended Kalman Filtering** (EKF): a recursive Gauss-Newton method for learning nonlinear models **online**

ONLINE LEARNING VIA EXTENDED KALMAN FILTERING

TRAINING RNNS BY EKF

(Puskorius, Feldkamp, 1994) (Wang, Huang, 2011) (Bemporad, 2023)

- **Key idea**: treat the parameters of the feedforward NNs as **constant states** and iterate an EKF on the training dataset:

$$\begin{cases} x_{k+1} = f_x(x_k, u_k, \theta_{xk}) + \xi_k \\ \begin{bmatrix} \theta_{x(k+1)} \\ \theta_{y(k+1)} \end{bmatrix} = \begin{bmatrix} \theta_{xk} \\ \theta_{yk} \end{bmatrix} + \eta_k \\ y_k = f_y(x_k, \theta_{yk}) + \zeta_k \end{cases}$$

$$Q = \text{Var}[\eta_k] \quad R = \text{Var}[\zeta_k] \quad P_0 = \text{Var}\left[\begin{bmatrix} \theta_x \\ \theta_y \\ x_0 \end{bmatrix}\right]$$

- The ratio Q/R determines the **learning-rate** of the training algorithm

The model θ_x, θ_y can be learned offline by processing a given dataset multiple times, and also **adapted on line** from streaming data (u_k, y_k)

- **Generalization**: train via **Moving Horizon Estimation** (MHE) instead of EKF
  (Løwenstein, Bernardini, Bemporad, Fagiano, 2023)

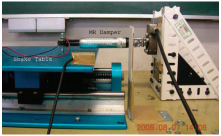
TRAINING RNNS BY EKF

(Bemporad, 2023)

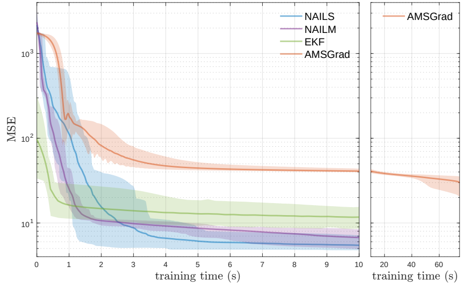
- EKF can be generalized to handle **general strongly convex and smooth** loss functions $\ell(y_k, \hat{y}_k)$
- Strongly convex smooth regularization terms $r(x_0, \theta_x, \theta_y)$ can be handled similarly
- Can handle **ℓ_1-penalties** $\lambda \left\| \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} \right\|_1$, useful to **sparsify** θ_x, θ_y by changing the EKF update into

$$\begin{bmatrix} \hat{x}(k|k) \\ \theta_x(k|k) \\ \theta_y(k|k) \end{bmatrix} = \begin{bmatrix} \hat{x}(k|k-1) \\ \theta_x(k|k-1) \\ \theta_y(k|k-1) \end{bmatrix} + M(k)e(k) - \lambda P(k|k-1) \begin{bmatrix} 0 \\ \mathrm{sign}(\theta_x(k|k-1)) \\ \mathrm{sign}(\theta_y(k|k-1)) \end{bmatrix}$$

- **Example**: **magneto-rheological fluid damper**
  N=2000 data used for training, 1499 for testing the model
  (Wang, Sano, Chen, Huang, 2009)



- RNN model: 4 states, shallow NNs w/ **4 neurons**, **I/O feedthrough**



MSE loss on training data, mean value (std) over 20 runs from different random initial weights

$$\text{BFR} = 100(1 - \frac{\|Y-\hat{Y}\|_2}{\|Y-\bar{y}\|_2})$$

BFR (Best Fit Rate)	training	test
NAILS	**94.41** (0.27)	89.35 (2.63)
NAILM	94.07 (0.38)	89.64 (2.30)
AMSGrad	84.69 (0.15)	80.56 (0.18)
EKF	91.41 (0.70)	87.17 (3.06)

NAILS = GNN method with **line search** (offline)

NAILM = GNN method with **LM steps** (offline)

(Bemporad, 2023)

TRAINING RECURRENT MODELS VIA L-BFGS

AI-generated image - DALL-E

SYSTEM IDENTIFICATION PROBLEM

- Class of nonlinear dynamical models (e.g., RNNs w/ linear bypass):

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k + f_x(x_k, u_k; \theta_x) \\ \hat{y}_k &= Cx_k + Du_k + f_y(x_k, u_k; \theta_y) \end{aligned}$$

Special cases:

linear model, RNN, ...

- Loss function (open-loop prediction error + regularization)

$$\begin{aligned} \min_{z, x_1, \ldots, x_{N-1}} \quad & r(z) + \frac{1}{N} \sum_{k=0}^{N-1} \ell(y_k, Cx_k + Du_k + f_y(x_k, u_k; \theta_y)) \\ \text{s.t.} \quad & x_{k+1} = Ax_k + Bu_k + f_x(x_k, u_k; \theta_x) \\ & k = 0, \ldots, N-2 \end{aligned}$$

$$z = \begin{bmatrix} x_0 \\ \Theta \end{bmatrix}$$

$$\Theta = \begin{bmatrix} A(:) \\ B(:) \\ C(:) \\ D(:) \\ \theta_x \\ \theta_y \end{bmatrix}$$

- Condense the problem by eliminating the hidden states x_k and get

$$\min_z f(z) + r(z)$$

(nonconvex) nonlinear programming (NLP) problem

NLP PROBLEM

- If f and r **differentiable**: use any state-of-the-art unconstrained NLP solver, e.g., **L-BFGS** (Limited-memory Broyden–Fletcher–Goldfarb–Shanno) (Liu, Nocedal, 1989)

- The gradient $\nabla f(z)$ can be computed efficiently by **automatic differentiation**

PyTorch CasADi Zygote

- However, **sparsifying** the model requires **non-smooth** regularizers:

$$r_1(z) = \tau \left\|z\right\|_1 \qquad\qquad r_g(z) = \tau_g \sum_{i=1}^{m} \|I_i z\|_2$$

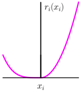
ℓ_1-regularization group-Lasso penalty

- **Group-Lasso penalties** can be used for reducing:
  - the **number of states**
  - the **number of inputs**
  - the **number of neurons** in each layer of f_x, f_y
  - ...

HANDLING NON-SMOOTH REGULARIZATION TERMS

(Bemporad, 2025)

1. If $r(x) = \sum_{i=1}^{n} r_i(x_i)$ and r_i's are **convex** & **positive semidefinite**, the ℓ_1-regularized problem can be recast as a **bound-constrained NLP**:

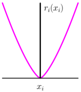


$$\min_x f(x) + \tau\|x\|_1 + r(x) \quad \rightarrow \quad \min_{y,z\geq 0} f(y-z) + \tau[1 \ \dots \ 1]\left[\begin{smallmatrix} y \\ z \end{smallmatrix}\right] + r(y) + r(-z)$$

$$x^* = y^* - z^*$$

Example: $r(x) = \|x\|_2^2$ then $r(y) + r(-z) = \|\left[\begin{smallmatrix} y \\ z \end{smallmatrix}\right]\|_2^2$ well-regularized augmented problem

2. If $r(x)$ is **convex** and **symmetric** wrt each component x_i and **increasing** for $x \geq 0$, and $\tau > 0$, then we can solve instead



$$\min_{y,z\geq 0} f(y-z) + \tau[1 \ \dots \ 1]\left[\begin{smallmatrix} y \\ z \end{smallmatrix}\right] + r(y+z)$$

if $r(x)$ differentiable for $x \neq 0$ then $r(y+z)$ differentiable if any $y_i, z_j > 0$

Example: $r(x)$ = group-Lasso penalty + constraint $y, z \geq \epsilon$ = machine precision

JAX-SYSID LIBRARY

(Bemporad, 2025)

- Python package to identify **linear/nonlinear/static** models:

jax-sysid

```
import numpy as np
from jax_sysid.models import Model

def state_fcn(x,u,params):
    ...

def output_fcn(x,u,params):
    ...

model = Model(nx, ny, nu, state_fcn=state_fcn, output_fcn=output_fcn)

model.init(params=[A,B,C,W1,W2,W3,b1,b2,W4,W5,b3,b4])
model.loss(rho_x0=1.e-4, rho_th=1.e-4, tau_th=1.e-4)
model.optimization(adam_epochs=1000, lbfgs_epochs=1000)
model.fit(Y, U)

Yhat, Xhat = model.predict(model.x0, U)
```

state-update function, $x(k+1)$

output function, $y(k)$

`pip install jax-sysid`

github.com/bemporad/jax-sysid

JAX-SYSID LIBRARY

- Python code for testing the model:

```
from jax_sysid.utils import compute_scores

x0_test = model.learn_x0(U_test, Y_test)
Yhat_test, Xhat_test = model.predict(x0_test, U_test)

R2_train, R2_test, msg = compute_scores(Y, Yhat, Y_test, Yhat_test, fit='R2')
print(msg)
```

Use multiple passes of **EKF** & **Rauch–Tung–Striebel** smoothing to estimate x_0

- Python code to identify a **linear time-invariant** model:

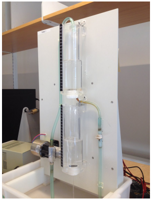
```
from jax_sysid.models import LinearModel

model = LinearModel(nx, ny, nu)
model.loss(rho_x0=1.e-3, rho_th=1.e-2)
model.optimization(lbfgs_epochs=1000)
model.fit(Y,U)
Yhat, Xhat = model.predict(model.x0, U)

A,B,C,D = model.ssdata()
```

$$x_{k+1} = Ax_k + Bu_k$$
$$y_k = Cx_k + Du_k$$

$$\theta = (A, B, C, D)$$

EXAMPLE: LINEAR SYSTEM IDENTIFICATION

- **Cascaded-Tanks benchmark**: (Schoukens, Mattson, Wigren, Noël, 2016)
  $z = (A, B, C, D, x_0)$, mean-squared error loss + ℓ_2-regularization



n_x	R^2 (training) lbfgs	sippy[1]	MATLAB[2]	R^2 (test) lbfgs	sippy	MATLAB	
1	87.43	56.24	87.06	83.22	52.38	83.18	(ssest)
2	94.07	28.97	93.81	92.16	23.70	92.17	(ssest)
3	94.07	74.09	93.63	92.16	68.74	91.56	(ssest)
4	94.07	48.34	92.34	92.16	45.50	90.33	(ssest)
5	94.07	90.70	93.40	92.16	89.51	80.22	(ssest)
6	94.07	94.00	93.99	92.17	92.32	88.49	(n4sid)
7	94.07	92.47	93.82	92.17	90.81	< 0	(ssest)
8	94.49	< 0	94.00	89.49	< 0	< 0	(n4sid)
9	94.07	< 0	< 0	92.17	< 0	< 0	(ssest)
10	94.08	93.39	< 0	92.17	92.35	< 0	(ssest)

$n_y = n_u = 1$

1024 training data

1024 test data

(standard scaling)

CPU time: 2.4 s (lbfgs), 30 ms (sippy), 50 ms (n4sid/pred.), 0.3 s (n4sid/sim.), 0.5 s (ssest) [Apple M1 Max]

1 (Armenise, Vaccari, Bacci Di Capaci, Pannocchia, 2018)

2 (Ljung, SYS-ID Toolbox)

LINEAR SYSTEM IDENTIFICATION W/ STABILITY CONSTRAINTS

- **Stability**: $\exists P \succ 0$ s.t. $P \succeq A'PA \quad \Leftrightarrow \quad I \succeq \bar{A}'\bar{A}$ for $\bar{A} = T^{-1}AT, \quad T'T = P$
- We try enforcing $\|A\|_2 \leq 1$ via the penalty $\rho_A \max\{\|A\|_2^2 - 1 + \epsilon_A, 0\}^2$ (wlog)
- Example: 1000 training + 1000 test data generated by the **unstable LTI** system

$$x_{k+1} = \begin{bmatrix} 1.0001 & 0.5 & 0.5 \\ 0 & 0.9 & -0.2 \\ 0 & 0 & 0.7 \end{bmatrix} x_k + Bu_k + \xi_k$$

$$y_k = Cx_k + z_k$$

$$B, C \in \mathcal{N}(0,1)$$

$$\xi_k \in \mathcal{N}(0, 0.01^2), \zeta_k \in \mathcal{N}(0, 0.05^2)$$

$$u_k \in \mathcal{U}[-\tfrac{1}{2}, \tfrac{1}{2}]$$

- Training setup:

```
model.force_stability(rho_A=1.e3, epsilon_A=1.e-3)
```

  - $\rho_A = 10^3$, $\epsilon_A = 10^{-3}$
  - 3000 Adam + 5000 L-BFGS iters
  - CPU time $\approx$ **3.36 s** [Apple M4 Max]

BFR (Best Fit Rate)	training	test
	98.2930	91.7369

Eigenvalues of identified matrix A:

$$0.99997, 0.92747, 0.59781$$

QUASI-LPV MODEL IDENTIFICATION

- **Quasi-Linear Parameter Varying** (qLPV, a.k.a. "**self-scheduled**" LPV) models:

$$\begin{aligned} x_{k+1} &= A(p_k)x_k + B(p_k)u_k \\ y_k &= C(p_k)x_k + D(p_k)u_k \\ p_k &= f(x_k, u_k, \theta_p) \end{aligned}$$

$$\begin{bmatrix} A(p_k) & B(p_k) \\ C(p_k) & D(p_k) \end{bmatrix} = \begin{bmatrix} A_0 & B_0 \\ C_0 & D_0 \end{bmatrix} + \sum_{i=1}^{n_p} \begin{bmatrix} A_i & B_i \\ C_i & D_i \end{bmatrix} p_{ki}$$

p_k = **scheduling parameter**

model parameters:

$$\theta = \begin{pmatrix} A_0, B_0, C_0, D_0 \\ \vdots \\ A_{n_p}, B_{n_p}, C_{n_p}, D_{n_p} \\ \theta_p \end{pmatrix}$$

Example: $p_k = f(x_k, u_k; \theta)$ = FNN with sigmoid output layer ($\Rightarrow 0 \leq p_{ki} \leq 1$)

- (q)LPV models are a very powerful class of control-oriented nonlinear models

(Shamma, Athans, 1991) (Tóth, 2010)

```
from jax_sysid.models import qLPVModel

model = qLPVModel(nx, ny, nu, npar, qlpv_fcn, qlpv_params_init)
```

EXAMPLE: QLPV MODEL IDENTIFICATION

- Generate 5000 training data and 1000 test data from the NL dynamics

$$x_{k+1} = \begin{bmatrix} 0.5\sin(x_{1k}) + 1.7\cos(0.5x_{2k})u_k \\ 0.6\sin(x_{1k}+x_{3k}) + 0.4\,\mathrm{atan}(x_{1k}+x_{2k}) \\ 0.4\,e^{-x_{2k}} + 0.9\sin(-0.5x_{1k})u_k \end{bmatrix} + \xi_k$$

$$y_k = \mathrm{atan}(2.2x_{1k}^3) + \mathrm{atan}(1.8x_{2k}^3) + \mathrm{atan}(-x_{3k}^3) + z_k$$

where $\xi_k, z_k \in \mathcal{N}(0, 0.01^2)$ and u_k uniformly generated in $[-\frac{1}{2}, \frac{1}{2}]$

- p_k = 2-layer FNN (6 neurons each) + swish activation + sigmoid output function

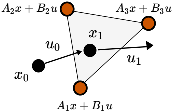
- Training results:
  - 1000 Adam + 5000 L-BFGS iters
  - CPU time measured on [Apple M4 Max]

model	n_p	Best Fit Rate training data	Best Fit Rate test data	CPU time (s)
LTI	0	71.35	71.36	1.3
qLPV	1	93.57	93.55	20.1
qLPV	2	95.54	95.51	22.6
qLPV	3	96.04	95.94	26.4

COMBINED LEARNING OF MODEL AND INVARIANT SETS

(Mulagaleti, Bemporad, 2025)

- **Goal**: learn a model for **control design** under **constraints** $y \in \mathcal{Y}, u \in \mathcal{U}$



- **Self-scheduled LPV model**:

$$x_{k+1} = A(p(x_k))x_k + B(p(x_k))u_k$$
$$y_k = Cx_k$$
$$p(x_k) = \text{softmax}(N_1(x_k), \ldots, N_{n_p}(x_k))$$

$$\left[\begin{smallmatrix} A \\ B \end{smallmatrix}\right](p) = \sum_{i=1}^{n_p} \left[\begin{smallmatrix} A_i \\ B_i \end{smallmatrix}\right] p_i$$

$$\Rightarrow \quad 0 \le p_{ki} \le 1, \quad \sum_{i=1}^{n_p} p_{ki} = 1$$

- Uncertain predictions: $x_{k+1} \in \text{conv}(A_i x_k + B_i u_k,\ i = 1, \ldots, n_p)$

- **Control invariant set** R: $\forall x \in R$, $\exists u \in \mathcal{U}$ s.t. $A_i x + B_i u \in R$, $\forall i = 1, \ldots, n_p$

COMBINED LEARNING OF MODEL AND INVARIANT SETS

- **Key idea**: add **regularization term** $r(\theta)$ in training problem (θ = model coeffs):

$$
\begin{aligned}
r(\theta) = \quad & \min_{R} \text{conservativeness}(R) \\
\text{s.t.} \quad & C \cdot R \oplus W \subseteq \mathcal{Y} \\
& R \text{ control invariant}
\end{aligned}
$$

- $r(\theta) < \infty \Rightarrow \exists$ **control invariant set** R

- small $r(\theta) \Rightarrow$ less conservative R

- qLPV + polytopic sets $\Rightarrow r(\theta)$ differentiable

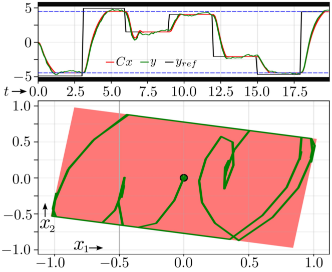
- **Example:** $1.5\ddot{y} + \dot{y} + y + 1000y^3 = u$

10,000 training + 2,000 disturbance-set estimation data

learned model: $n_x = 4$ states, $n_p = 6$ scheduling params

p-function = shallow FNN with 3 neurons

$r(\theta)$ almost does not perturb quality of fit



INDUSTRIAL ROBOT BENCHMARK

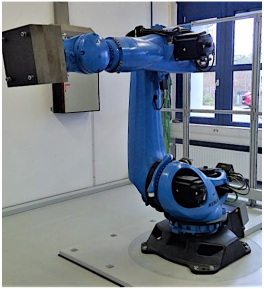
(Weigand, Götz, Ulmen, Ruskowski, 2022)

- KUKA KR300 R2500 ultra SE industrial robot, full robot movement

- **6 inputs** (torques), **6 outputs** (joint angles), w/ backlash, highly **nonlinear** and coupled, **slightly oversampled** ($\|y_k - y_{k-1}\|$ is often very small)

- Identification benchmark dataset (forward model):
  - Sample time: $T_s = 100$ ms
  - N = 39988 training samples
  - N_{test} = 3636 test samples

- Very challenging NL-SYSID benchmark on **`nonlinearbenchmark.org`**



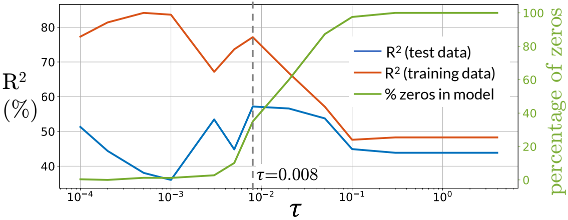
`nonlinearbenchmark.org`

INDUSTRIAL ROBOT BENCHMARK: RESULTS

(Bemporad, 2024)

- State $x \in \mathbb{R}^{12}$, f_x, f_y with **36** and **24** neurons, **swish** activation fcn $\frac{x}{1+e^{-x}}$

- Total number of training parameters: $\dim(\theta_x) + \dim(\theta_y) = 1590$



(best R^2 in 30 runs)

- Model quality measured by **average R^2-score** on all outputs:

$$R^2 = \frac{1}{n_y} \sum_{i=1}^{n_y} 100 \left(1 - \frac{\sum_{k=1}^{N} (y_{k,i} - \hat{y}_{k,i|0})^2}{\sum_{k=1}^{N} (y_{k,i} - \frac{1}{N} \sum_{i=1}^{N} y_{k,i})^2} \right)$$

- Training time $\approx$ 12 min per run on a single core [Apple M1 Max]

  (3192 variables, 2000 Adam iterations + 2000 L-BFGS-B iterations)

INDUSTRIAL ROBOT BENCHMARK: RESULTS

- **Open-loop simulation** errors ($\rho = 0.01, \tau = 0.008$):

	R^2 (training) RNN model	R^2 (test) RNN model	R^2 (training) linear model	R^2 (test) linear model	
average	77.1493	57.1784	48.2789	43.8573	**jax-sysid**
			39.2822	32.0410	**n4sid**

- More parameters/smaller regularization leads to overfitting training data

- **Pure Adam** vs **LBFG-B+Adam** vs **OWL-QN** (Andrew, Gao, 2007): ($\tau = 0.008$)

solver	adam iters	fcn evals	$\overline{R^2}$ training	$\overline{R^2}$ test	# zeros (θ_x, θ_y)	CPU time (s)
L-BFGS-B	2000	2000	**77.1493**	**57.1784**	556/1590	**309.87**
OWL-QN	2000	2000	74.7816	54.0531	**736/1590**	449.17
Adam	6000	0	71.0687	54.3636	1/1590	389.39

- Adam is unable to sparsify the model

OTHER FEATURES OF JAX-SYSID LIBRARY

- **Parallel training**: train models from different initial guesses (x_0, θ)

```
from jax_sysid.models import find_best_model

models = model.parallel_fit(Ys, Us, init_fcn=init_fcn, seeds=range(10))

best_model, best_R2 = find_best_model(models, Ys_test, Us_test, fit='R2')
```

- **Multiple training datasets**: $(u_k^i, y_k^i), \quad k = 0, \ldots, N_i - 1, \quad i = 1, \ldots, M$

```
model.fit( [Ys1, ..., YsM],[Us1, ..., UsM] )
```

- **Static gain**: $\hat{y}_{ss} \approx y_{ss}$ when $x_{ss} = f_x(x_{ss}, u_{ss}, \theta)$

```
dcgain_loss = model.dcgain_loss(Uss = Uss, Yss = Yss)

model.loss(rho_x0=1.e-3, rho_th=1.e-2, custom_regularization = dcgain_loss)
```

(for linear models: can also specify the desired DC gain $\hat{y}_{ss} = Gu_{ss}$ directly)

OTHER FEATURES OF JAX-SYSID LIBRARY

- **Custom output loss** $\ell(\hat{y}, y)$

```
eps = 1.e-4
def cross_entropy_loss(Yhat,Y):
    loss=jnp.sum(-Y*jnp.log(eps+Yhat)-(1.-Y)*jnp.log(eps+1.-Yhat))
    return loss

model.loss(rho_x0=0.01, rho_th=0.001, output_loss=cross_entropy_loss)
```

- **Custom regularization** $r(\theta, x_0)$

```
def custom_reg_fcn(th,x0):
    return 1000.*jnp.maximum(jnp.sum(th**2)-1.,0.)**2

model.loss(rho_x0=0.01, rho_th=0.001, custom_regularization= custom_reg_fcn)
```

- **Upper and lower bounds** on parameters and states

```
model.optimization(params_min=lb, params_max=ub, x0_min=xl, x0_max=xu, ...)
```

OTHER FEATURES OF JAX-SYSID LIBRARY

- **RNN models** described in `flax.linen`

```
from flax import linen as nn
from jax_sysid.models import RNN

model = RNN(nx, ny, nu, FX=FX, FY=FY, x_scaling=0.1)
```

```
class FX(nn.Module):
    @nn.compact
        def __call__(self, x):
        x = nn.Dense(features=5)(x)
        x = nn.swish(x)
        x = nn.Dense(features=5)(x)
        x = nn.swish(x)
        x = nn.Dense(features=nx)(x)
        return x
```

- **Continuous-time models** $\dot{x} = f(x, u, t), y = g(x, u, t)$

```
from jax_sysid import CTModel

model = CTModel(nx, ny, nu, state_fcn=state_fcn, output_fcn=output_fcn)
```

- **Static models** $\hat{y} = f(x)$ (=standard nonlinear regression)

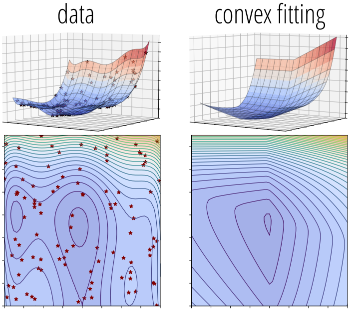
```
from jax_sysid import StaticModel

model = StaticModel(ny, nx, output_fcn)
```

Example: **NARX model** $\hat{y}_k = f(y_{k-1}, \ldots, y_{k-n_a}, u_{k-1}, \ldots, u_{k-n_b})$
(minimize 1-step-ahead prediction error $y_k - \hat{y}_k$)

LEARNING PARAMETRIC CONVEX FUNCTIONS

PARAMETRIC CONVEX FUNCTIONS

- **Goal**: learn a **parametric convex function** (PCF) from data (x_k, θ_k, y_k)



$$y = f(x, \theta)$$

$$f : \mathbb{R}^n \times \Theta \to \mathbb{R}^d$$

with $f(x,\theta)$ **convex** wrt the **variable** $x \in \mathbb{R}^n$ for each **parameter** $\theta \in \Theta$

- **Use**: **optimize** f wrt x for each given θ in production

- **Example**: $f(x,\theta) = \frac{1}{2}x'F'(\theta)F(\theta)x + c(\theta)x + h(\theta)$

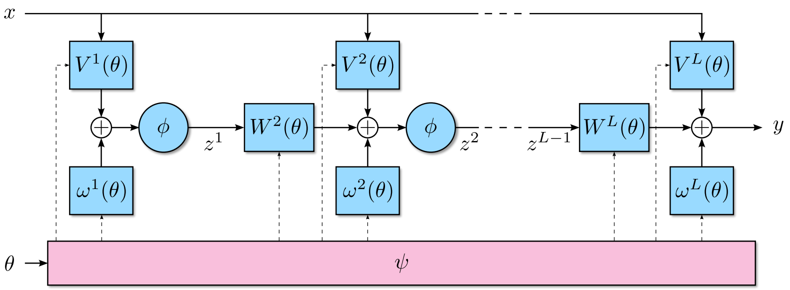
- Several **input-convex** NN architectures have been proposed in the literature

  (Amos, Xu, Kolter, 2017) (Calafiore, Gaubert, Possieri, 2020)

NEURAL PCF ARCHITECTURE

(Schaller, Bemporad, Boyd, 2025)

- f = neural network with weights $V^i(\theta)$, $W^i(\theta)$, biases $\omega^i(\theta)$, and activation ϕ



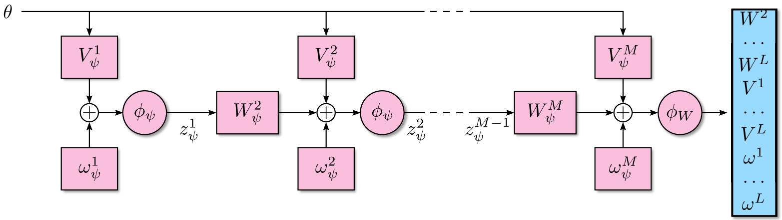
- V^i, W^i, ω^i generated by another network ψ (to be learned)
- Activation function ϕ is **nondecreasing** and **convex** (e.g., ReLU, softplus)
- The weights $W^i(\theta)$ are **elementwise nonnegative** for all θ

→ **$f(x,\theta)$ convex for all θ**

NEURAL PCF ARCHITECTURE

- The NN ψ is nonlinear re-parametrization from θ to the PCF weights

- ψ has weights $w = (W_\psi^2, \ldots, W_\psi^M, V_\psi^1, \ldots, V_\psi^M, \omega_\psi^1, \ldots, \omega_\psi^M)$



- The last layer of ψ makes $W^i(\theta)$ elementwise nonnegative $\forall\theta$

Examples:

$$W^i(\theta) = \max(V_\psi^M\theta + W_\psi^M z_\psi^{M-1} + \omega_\psi^M, 0)$$

$$W^i(\theta) = (V_\psi^M\theta + W_\psi^M z_\psi^{M-1} + \omega_\psi^M)^2$$

THE LPCF PACKAGE

- Open-source package for fitting a PCF to given data (Schaller, Bemporad, Boyd, 2025)

```
pip install lpcf
```

https://github.com/cvxgrp/lpcf

- Customizable neural network architecture

- Customizable loss and regularization

- Relies on **`jax_sysid`** (Adam + L-BFGS) for training

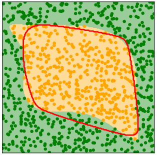
- Returns the PCF f as
  - a **JAX** function for fast evaluation (and differentiation)
  - a **CVXPY** expression for use in optimization models (Diamond, Boyd, 2016)

USING THE LPCF PACKAGE

```
from lpcf.pcf import PCF

# observed data
Y = ... # shape (N, d)
X = ... # shape (N, n)
Theta = ... # shape (N, p)

# fit PCF to data
pcf = PCF()
pcf.fit(Y, X, Theta)

# export PCF to CVXPY
x = cp.Variable((n, 1))
theta = cp.Parameter((p, 1))
y = pcf.tocvxpy(x, theta) # CVXPY expression
prob = cp.Problem(cp.Minimize(y + ...))
...

f = pcf.tojax() # JAX function f(x, θ)
```

Additional features:

- add (convex) **quadratic term** to the neural network
- require f to be **monotone** in x
- require f to be **nonnegative**
- require $\arg\min_x f(x,\theta) = g(\theta)$ for a given function g
- fit a parametrized convex set $C(\theta) = \{x \mid f(x,\theta) \leq 0\}$ (**convex classification** problem)



EXAMPLE: APPROXIMATE DYNAMIC PROGRAMMING (ADP)

- Consider the **input-affine** nonlinear system

$$x_{t+1} = F(x_t, \theta) + G(x_t, \theta)u_t, \quad t = 0, 1, \ldots$$

- θ are measured parameters (e.g., physical quantities)
- **Goal**: for each given initial state x_0, find $u_0, u_1, \ldots$ that minimize

$$J(x_0) = \sum_{t=0}^{\infty} H(x_t, u_t, \theta) = H(x_0, u_0, \theta) + \underbrace{\sum_{t=1}^{\infty} H(x_t, u_t, \theta)}_{\text{cost to go}}$$

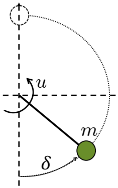
- **ADP controller** (i.e., MPC with horizon $N = 1$):

$$u_t = \arg\min_u \left(H(x_t, u, \theta) + f(F(x_t, \theta) + G(x_t, \theta)u, \theta) \right), \quad t = 0, 1, \ldots$$

- **Convex problem** if $f(x, \theta)$ = PCF approx of $y = J(x, \theta)$ and H convex in u

EXAMPLE: APPROXIMATE DYNAMIC PROGRAMMING (ADP)

- **Example**: swing up inverted pendulum
  $x = [\delta, \dot{\delta}]', \theta = m > 0$ (mass)

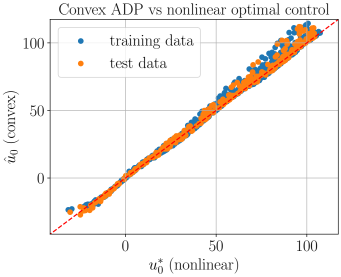


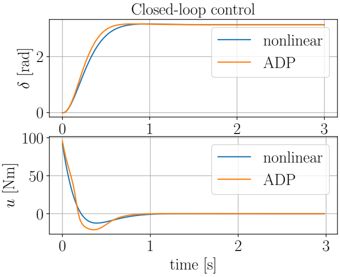
- Solve nonlinear optimal control problem

$$J(x_0) = \sum_{t=0}^{150} H(x_t, u_t, \theta)$$

  on 1000 data points $(x_k, \theta_k), \theta_k \in [0.5, 2], \delta_k \in [-\pi/6, 7\pi/6], \dot{\delta}_k \in [-1, 1]$

- Fit PCF $f(x, \theta)$ and use CVXPY to solve the ADP problem online





LEARNING COMBINED PROCESS & NOISE MODELS

PROCESS AND NOISE MODELS

(Bemporad, Tóth, 2025)

- Role of **stochastic noise models**, combined with process models, is well understood for **linear systems**

- **Goal**: extend the use of noise models to general **nonlinear state-space models** (RNNs, qLPVs, ...)

- **Model structure:** $y_k = y_{o,k} + v_k$ data generating system

$$G_o : \begin{cases} x_{k+1} = f_x(x_k, u_k) \\ y_{o,k} = g_x(x_k, u_k) \end{cases} \qquad H_o : \begin{cases} z_{k+1} = f_z(z_k, x_k, u_k, e_k) \\ v_k = g_z(z_k, x_k, u_k) + e_k \end{cases}$$

process model   noise model

- Since $e_k = v_k - g_z(z_k, x_k, u_k)$, we also get the **inverse noise model**

$$H_o^{-1} : \begin{cases} z_{k+1} = f_z(z_k, x_k, u_k, v_k - g_z(z_k, x_k, u_k)) \\ e_k = v_k - g_z(z_k, x_k, u_k) \end{cases}$$

MODEL PARAMETERIZATION AND TRAINING PROBLEM

- Training dataset: N samples $(u_0, y_0, \ldots, u_{N-1}, y_{N-1})$

- Parametric process + noise model and prediction error:

$$\begin{aligned} \hat{x}_{k+1} &= f_{\rm x}(\hat{x}_k, u_k, \theta_{\rm x}) \\ \hat{z}_{k+1} &= \tilde{f}_{\rm z}(\hat{z}_k, \hat{x}_k, u_k, y_k - g_{\rm x}(\hat{x}_k, u_k, \theta_{\rm y}), \theta_{\rm z}) \\ \hat{e}_k^{\rm pred} &= y_k - g_{\rm x}(\hat{x}_k, u_k, \theta_{\rm y}) - g_{\rm z}(\hat{z}_k, \hat{x}_k, u_k, \theta_{\rm e}) \end{aligned}$$

inverse noise-model update

one-step-ahead prediction error

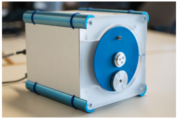
- (Regularized) prediction-error minimization (PEM) problem:

$$\min_{\theta, \hat{x}_0, \hat{z}_0} \frac{1}{N} \sum_{k=0}^{N-1} \|\hat{e}_k^{\rm pred}\|_2^2 \quad + \quad R(\theta, \hat{x}_0, \hat{z}_0) \qquad\qquad \theta = (\theta_{\rm x}, \theta_{\rm y}, \theta_{\rm z}, \theta_{\rm e})$$

- Special cases: **LTI**, **LPV** (ext.-scheduled, self-scheduled), **nonlinear** models

- Under suitable assumptions, **consistency** can be proved as $N \to \infty$

EXAMPLE: UNBALANCED DISK SYSTEM



- System dynamics: $\ddot{\alpha} = -\frac{1}{\tau}\dot{\alpha} + \frac{K_{\rm m}}{\tau}u - \frac{mgl}{J}\sin(\alpha)$

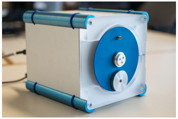
  (Kulcsár, Dong, van Wingerden, Verhaegen, 2009) (Koelewijn, Tóth, 2019)

- **LTI** model: 2000 training and test data generated by linearized system + noise

	$\hat{n}_{\rm x}$	$\hat{n}_{\rm z}$	BFR train.	BFR test	type	time
best achievable	2	1	72.12% 77.66%	68.13% 72.85%	sim pred	- -
plant only	2	0	72.03%	68.08%	sim	0.13 s
combined	2 2	1 1	72.02% 77.67%	68.08% 72.84%	sim pred	0.34 s
n4sid (s)	2	-	0.33%	0.56%	sim	0.15 s
n4sid (p)	3	-	61.21% 75.63%	54.49% 70.46%	sim pred	0.11 s
ssest (s)	2	-	1.33%	1.37%	sim	0.47 s
ssest (p)	3	-	64.23% 76.31%	58.02% 71.43%	sim pred	0.20 s

- Training: `jax-sysid`
- 10000 L-BFGS iters
- CPU: [Apple M4 Max]

EXAMPLE: UNBALANCED DISK SYSTEM



- **self-scheduled LPV** model: 2000 training + 2000 test data generated by NL system + noise

	$\hat{n}_x$	$\hat{n}_z$	BFR train.	BFR test	type	noise	time
best achievable	2	1	89.32% 91.24%	90.11% 91.86%	sim pred	LPV LPV	- -
plant only	2	0	87.73%	86.45%	sim	–	10.06 s
combined	2	1	85.19% 90.92%	86.19% 91.46%	sim pred	LTI LTI	12.91 s
combined	2	1	85.60% 90.96%	86.56% 91.51%	sim pred	LPV LPV	18.82 s

- Training: `jax-sysid`
- 1000 Adam + 10000 L-BFGS iters
- CPU: [Apple M4 Max]

- **nonlinear** model: (same dataset)

	$\hat{n}_x$	$\hat{n}_z$	BFR train.	BFR test	type	time
best achievable	2	1	89.32% 91.24%	90.11% 91.86%	sim pred	- -
plant only	2	0	89.33%	89.89%	sim	10.24 s
combined	2	1	89.23% 91.05%	89.83% 91.39%	sim pred	11.31 s

EXAMPLE: CONTROL MOMENT GYROSCOPE

- Data generated by high-fidelity CMG simulation model (Bloemers, Tóth, 2019), red gymbal locked, 1 input, 1 output
- 10,000 training data + 30,000 test data (SNR = 35dB in both datasets)
- **LTI** model:

	$\hat{n}_\mathrm{x}$	$\hat{n}_\mathrm{z}$	BFR train.	BFR test	type	time
best achievable	5	-	98.16%	98.19%	sim	-
plant only	8	0	35.99%	25.28%	sim	3.50 s
combined	8	2	34.46%	29.91%	sim	8.73 s
	8	2	97.17%	97.12%	pred	
n4sid	8	0	29.72%	20.98%	sim	0.85 s
n4sid	10	0	34.76%	22.45%	sim	1.05 s
ssest (s)	8	0	35.48%	24.70%	sim	2.05 s
ssest (p)	10	0	33.51%	26.88%	sim	3.06 s

- Training: `jax-sysid`
- 10000 L-BFGS iters
- CPU: [Apple M4 Max]

EXAMPLE: CONTROL MOMENT GYROSCOPE

- **self-scheduled LPV** model: 2000 training + 2000 test data generated by NL system + noise

	$\hat{n}_x$	$\hat{n}_z$	BFR train.	BFR test	type	time
best achievable	5	-	98.16%	98.19%	sim	-
plant only	5	0	97.61%	96.50%	sim	47.54 s
combined	5	2	81.96%	83.78%	sim	67.19 s
			97.56%	97.64%	pred	
SUBNET	5	0	97.28%	96.40%	sim	≈ 10 h

- Training: `jax-sysid`
- 1000 Adam + 10000 L-BFGS iters
- CPU: [Apple M4 Max]

- **nonlinear** model: (same dataset)

	$\hat{n}_x$	$\hat{n}_z$	BFR train.	BFR test	type	time
best achievable	5	-	98.16%	98.19%	sim	-
plant only	5	0	96.75%	96.12%	sim	42.10 s
combined	5	2	96.66%	96.12%	sim	47.78 s
	5	2	97.82%	97.84%	pred	

CONCLUSIONS

CONCLUSIONS

- **Quasi-Newton** methods for SYS-ID enabled by powerful **autodiff libs**
  - ☺ **Extremely flexible** (model structure, loss functions, regularization terms)
  - ☺ **Faster convergence/better models** than with classical GD methods (like Adam)
  - ☺ Numerically very **robust** (even to get **linear state-space models**!)
  - 😐 **Non-convex problem**: multiple runs often required from different initial guesses

- **Current research**:
  - How to get good-quality training data (**active learning**) (Xie, Bemporad, *CDC*, 2024)
  - Augmented Lagrangian methods for **non-smooth** nonlinear optimization with **constraints** (Adeoye, Latafat, Bemporad, 2025)

ERC Advanced Grant "COMPACT" (2024-2029)

Funded by the European Union

European Research Council
Established by the European Commission