MODEL PREDICTIVE CONTROL

LEARNING-BASED MPC

Alberto Bemporad

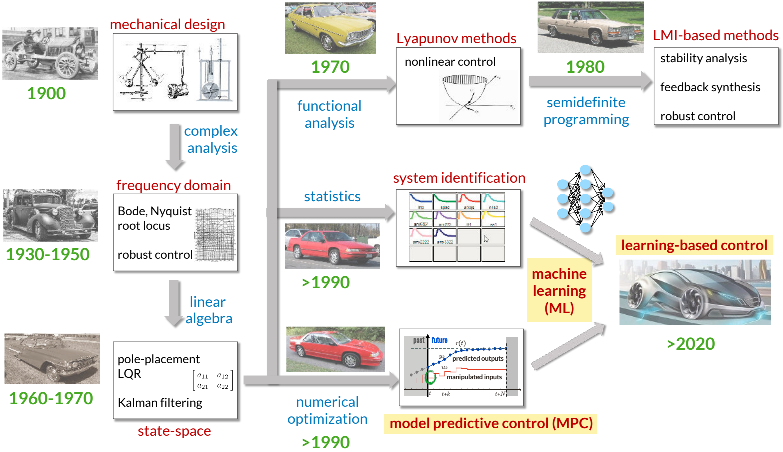
`http://cse.lab.imtlucca.it/~bemporad/mpc_course.html`

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

COURSE STRUCTURE

- ✓ **Basic concepts** of model predictive control (MPC) and **linear MPC**
- ✓ **Linear time-varying** and **nonlinear** MPC
- ✓ **Quadratic programming** (QP) and **explicit MPC**
- ✓ **Hybrid** MPC
- ✓ **Stochastic** MPC
- **Learning-based** MPC (or **data-driven** MPC)

MACHINE LEARNING AND CONTROL ENGINEERING



1900

mechanical design

complex analysis

frequency domain

1930-1950

Bode, Nyquist
root locus

robust control

linear algebra

1960-1970

pole-placement
LQR

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Kalman filtering

state-space

1970

functional analysis

Lyapunov methods

nonlinear control

1980

semidefinite programming

LMI-based methods

stability analysis

feedback synthesis

robust control

statistics

system identification

>1990

machine learning (ML)

learning-based control

>2020

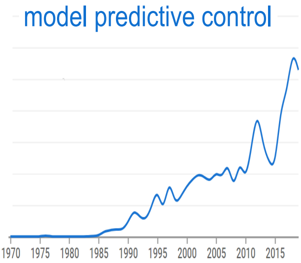
numerical optimization

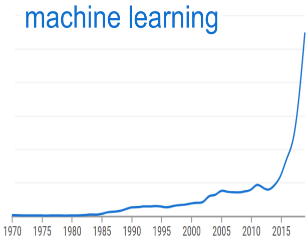
>1990

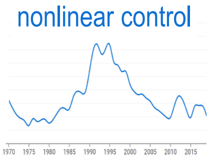
model predictive control (MPC)

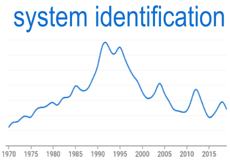
MPC AND ML

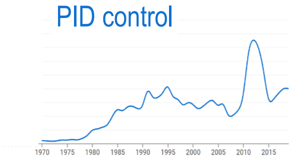
- **MPC** and **ML** = main R&D trends in industry for control!


model predictive control


machine learning


nonlinear control

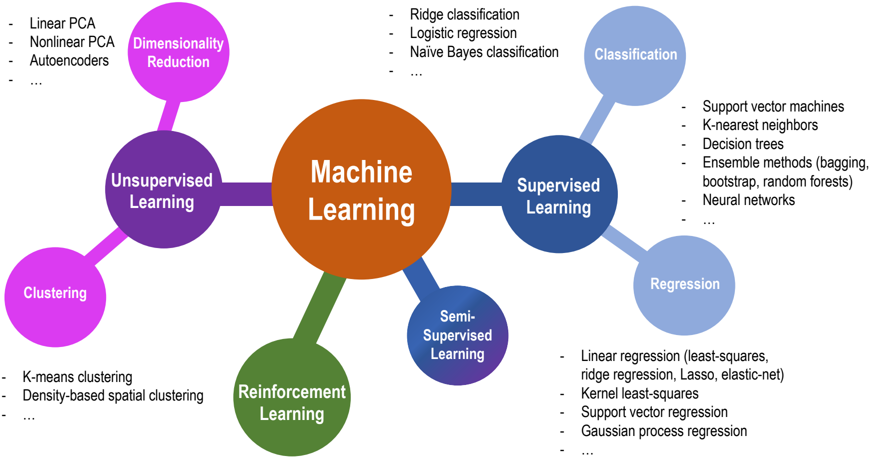

system identification


PID control

(source: https://books.google.com/ngrams)

MACHINE LEARNING (ML)

- Massive set of techniques to **extract mathematical models from data**



Machine Learning

Unsupervised Learning

Dimensionality Reduction

- Linear PCA
- Nonlinear PCA
- Autoencoders
- …

Clustering

- K-means clustering
- Density-based spatial clustering
- …

Reinforcement Learning

Semi-Supervised Learning

Supervised Learning

Classification

- Ridge classification
- Logistic regression
- Naïve Bayes classification
- …

- Support vector machines
- K-nearest neighbors
- Decision trees
- Ensemble methods (bagging, bootstrap, random forests)
- Neural networks
- …

Regression

- Linear regression (least-squares, ridge regression, Lasso, elastic-net)
- Kernel least-squares
- Support vector regression
- Gaussian process regression
- …

MACHINE LEARNING (ML)

- Good **mathematical foundations** from artificial intelligence, statistics, optimization

- **Works very well** in practice (despite training is most often a nonconvex optimization problem ...)

- Used in myriads of **very diverse application domains**

- Availability of excellent open-source **software tools** also explains success
  `scikit-learn`, **`TensorFlow/Keras`**, **`PyTorch`**, **`JAX`**, **`Flux.jl`**, ... python julia

MPC DESIGN FROM DATA

1. Use **machine learning** to get a **prediction model** from data (**system identification**)

   - **Autoencoders**, **recurrent neural networks** (nonlinear models)
   - **Online learning** of feedforward/recurrent neural networks by EKF
   - **Piecewise affine regression** to learn hybrid models

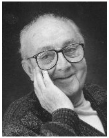
2. Use **reinforcement learning** to learn the **MPC law** from data

   - **Q-learning**: learn Q-function defining the MPC law from data
   - **Policy gradient methods**: learn optimal policy coefficients directly from data using stochastic gradient descent
   - **Global optimization methods**: learn MPC parameters (weights, models, horizon, solver tolerances, ...) by optimizing observed closed-loop performance

LEARNING PREDICTION MODELS FOR MPC

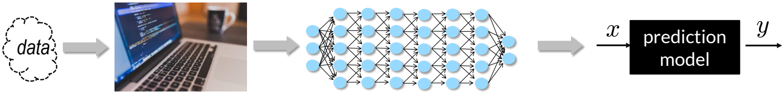
"All models are wrong, but some are useful."

(George E. P. Box)

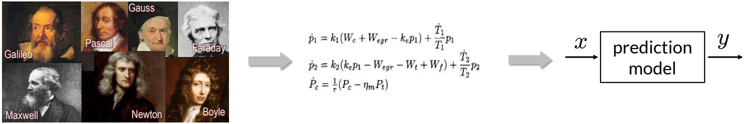


CONTROL-ORIENTED NONLINEAR MODELS

- **Black-box** models: purely data-driven. Use training data to fit a prediction model that can explain them (**need good data to get a good model**)



- **Physics-based** models: use physical principles to create a prediction model (**fewer parameters to learn, better generalizes on unseen data**)



$$\dot{p}_1 = k_1(W_c + W_{egr} - k_e p_1) + \frac{\dot{T}_1}{T_1} p_1$$
$$\dot{p}_2 = k_2(k_e p_1 - W_{egr} - W_t + W_f) + \frac{\dot{T}_2}{T_2} p_2$$
$$\dot{P}_c = \frac{1}{\tau}(P_c - \eta_m P_t)$$

- **Gray-box** (or **physics-informed**) models: mix of the two, can be quite effective

CONTROL-ORIENTED MODELS

- **Complex model = complex controller** (controller design and evaluation)
  Example: Model Predictive Control (MPC)

- Typically look for **small-scale models** (e.g., ≤ 10 states/inputs/outputs) with a **limited number of coefficients** (vs. Large Language Models: 2-300 B params)

- **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)

- **Computation constraints**: solve the learning problem using limited resources (=our laptop, no supercomputing infrastructures)

Solving system identification problems requires different algorithms compared to 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** to approximate the nonlinear difference equation $y_t \approx \mathcal{N}(y_{t-1}, \ldots, y_{t-n_a}, u_{t-1}, \ldots, u_{t-n_b})$

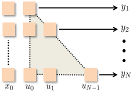
- **Neural state-space** models:

  - **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**: set x_t = value of an inner layer of the network (Prasad, Bequette, 2003) such as an **autoencoder** (Masti, Bemporad, 2021)

- Alternative for MPC: learn entire prediction (Masti, Smarra, D'Innocenzo, Bemporad, 2020)

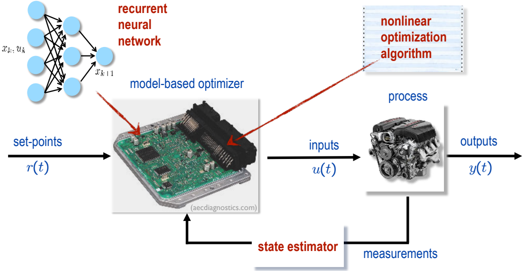
$$y_{t+k} = h_k(x_t, u_t, \ldots, u_{t+k-1}),\ k = 1, \ldots, N$$



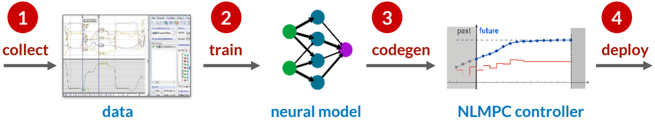
- **Recurrent neural networks** are more appropriate for accurate open-loop predictions, but more difficult to train (see later ...)

NLMPC BASED ON NEURAL NETWORKS

- **Approach**: use a neural network model for prediction



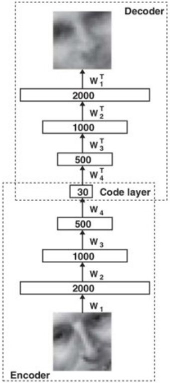
- MPC design workflow:



LEARNING NONLINEAR STATE-SPACE MODELS FOR MPC

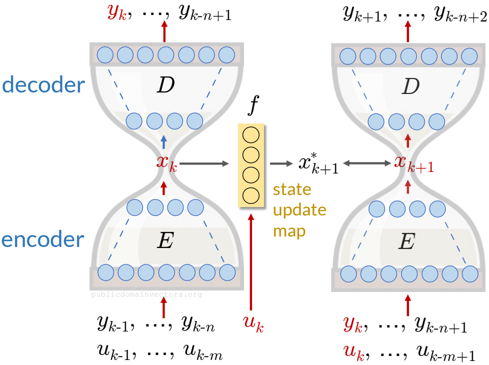
(Masti, Bemporad, 2021)

- **Idea**: use **autoencoders** and artificial neural networks to learn a **nonlinear state-space model** of **desired order** from input/output data



ANN with hourglass structure

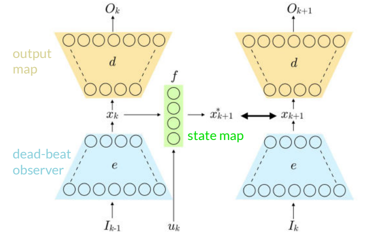
(Hinton, Salakhutdinov, 2006)



LEARNING NONLINEAR STATE-SPACE MODELS FOR MPC

- **Training problem**: choose n_a, n_b, n_x and solve

$$
\begin{aligned}
\min_{f,d,e} \quad & \sum_{k=k_0}^{N-1} \alpha\Big(\ell_1(\hat{O}_k, O_k) + \ell_1(\hat{O}_{k+1}, O_{k+1})\Big) \\
& +\beta\ell_2(x^\star_{k+1}, x_{k+1}) + \gamma\ell_3(O_{k+1}, O^\star_{k+1}) \\
\text{s.t.} \quad & x_k = e(I_{k-1}),\ k = k_0, \ldots, N \\
& x^\star_{k+1} = f(x_k, u_k),\ k = k_0, \ldots, N-1 \\
& \hat{O}_k = d(x_k),\ O^\star_k = d(x^\star_k),\ k = k_0, \ldots, N
\end{aligned}
$$



$$O_k = [y'_k \ \ldots \ y'_{k-m}]'$$

$$I_k = [y'_k \ \ldots \ y'_{k-n_a+1} \ u'_k \ \ldots \ u'_{k-n_b+1}]'$$

- Model complexity can be reduced by adding **group-LASSO** penalties

- **Quasi-LPV** structure for MPC: set

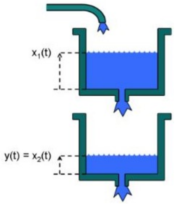
$$
\begin{aligned}
f(x_k, u_k) &= A(x_k, u_k)\left[\begin{smallmatrix} x_k \\ 1 \end{smallmatrix}\right] + B(x_k, u_k)u_k \\
y_k &= C(x_k, u_k)\left[\begin{smallmatrix} x_k \\ 1 \end{smallmatrix}\right]
\end{aligned}
$$

(A_{ij}, B_{ij}, C_{ij} = feedforward NNs)

- Different options for the **state-observer**:
  - use encoder e to map past I/O into x_k (deadbeat observer)
  - design extended Kalman filter based on obtained model f, d
  - **simultaneously fit state observer** $\hat{x}_{k+1} = s(x_k, u_k, y_k)$ with loss $\ell_4(\hat{x}_{k+1}, x_{k+1})$

LEARNING NONLINEAR NEURAL STATE-SPACE MODELS FOR MPC

- **Example**: nonlinear two-tank benchmark problem

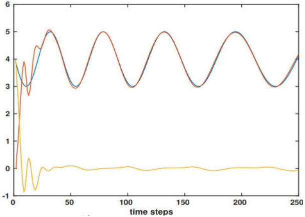


www.mathworks.com

$$\begin{cases} x_1(t+1) = x_1(t) - k_1\sqrt{x_1(t)} + k_2 u(t) \\ x_2(t+1) = x_2(t) + k_3\sqrt{x_1(t)} - k_4\sqrt{x_2(t)} \\ y(t) = x_2(t) + u(t) \end{cases}$$

Model is totally unknown to learning algorithm

- Artificial neural network (ANN): 3 hidden layers 60 exponential linear unit (ELU) neurons
- For given number of model parameters, **autoencoder approach is superior to NNARX**
- **Jacobians** directly obtained from ANN structure for Kalman filtering & MPC problem construction



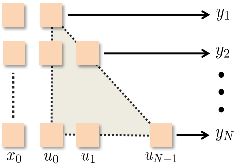
LTV-MPC results

LEARNING AFFINE NEURAL PREDICTORS FOR MPC

(Masti, Smarra, D'Innocenzo, Bemporad, 2020)

- Alternative: **learn the entire prediction**

$$y_k = h_k(x_0, u_0, \ldots, u_{k-1}),\ k = 1, \ldots, N$$



- **LTV-MPC formulation**: linearize h_k around nominal inputs $\bar u_j$

$$y_k = h_k(x_0, \bar u_0, \ldots, \bar u_{k-1}) + \sum_{j=0}^{k-1} \frac{\partial h_k}{\partial u_j}(x_0, \bar u_0, \ldots, \bar u_{k-1})(u_j - \bar u_j)$$

Example: $\bar u_k$ = MPC sequence optimized @$k-1$

- Avoid computing Jacobians by fitting h_k in the affine form

$$y_k = f_k(x_0, \bar u_0, \ldots, \bar u_{k-1}) + g_k(x_0, \bar u_0, \ldots, \bar u_{k-1}) \begin{bmatrix} u_0 - \bar u_0 \\ \vdots \\ u_{k-1} - \bar u_{k-1} \end{bmatrix}$$

cf. (Liu, Kadirkamanathan, 1998)

LEARNING AFFINE NEURAL PREDICTORS FOR MPC

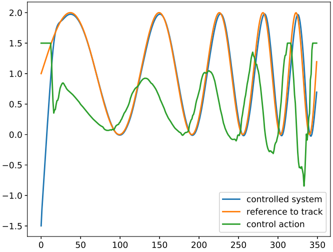
- **Example**: apply **affine neural predictor** to nonlinear two-tank benchmark problem

10000 training samples, ANN with **2** layers of **20 ReLU neurons**

$$\text{Best fit rate BFR} = \max\left\{0, 1 - \frac{\|\hat{y} - y\|_2}{\|y - \bar{y}\|_2}\right\}$$

Prediction step	BFR
1	0.959
2	0.958
4	0.948
7	0.915
10	0.858

- Closed-loop LTV-MPC results:



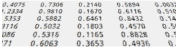
- Model complexity reduction: add **group-LASSO** term with penalty λ

λ	BFR (average on all prediction steps)	# nonzero weights
.01	0.853	328
0.005	0.868	363
0.001	0.901	556
0.0005	0.911	888
0	0.917	9000

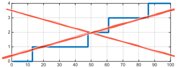
ON THE USE OF NEURAL NETWORKS FOR MPC

- Neural prediction models can **speed up** the MPC design a lot

- Experimental **data** need to well cover the operating range (as in linear system identification)

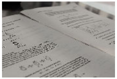


- No need to define linear operating ranges with NN's, it is a **one-shot model-learning** step



- Physical models may **better predict** unseen situations than black box models

- Physical modeling can help driving the choice of the **nonlinear model structure** to use (**gray-box** models)



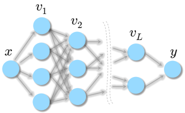
- NN model can be updated online for **adaptive nonlinear MPC**

LEARNING NEURAL NETWORK MODELS FOR CONTROL

TRAINING FEEDFORWARD NEURAL NETWORKS

- **Feedforward neural network** model:

$$y_k = f_y(x_k,\theta) = \begin{cases} v_{1k} & = & A_1 x_k + b_1 \\ v_{2k} & = & A_2 f_1(v_{1k}) + b_2 \\ \vdots & & \vdots \\ v_{Lk} & = & A_{L_y} f_{L-1}(v_{(L-1)k}) + b_L \\ \hat{y}_k & = & f_L(v_{Lk}) \end{cases}$$

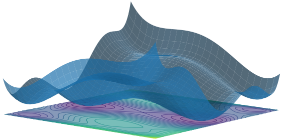


$$\theta = (A_1, b_1, \ldots, A_L, b_L)$$

E.g.: x_k = current state & input, or $x_k = (y_{k-1}, \ldots, y_{k-n_a}, u_{k-1}, \ldots, u_{k-n_b})$

- **Training problem**: given a dataset $\{x_0, y_0, \ldots, x_{N-1}, y_{N-1}\}$ solve

$$\min_{\theta} r(\theta) + \sum_{k=0}^{N-1} \ell(y_k, f(x_k, \theta))$$



- It is a nonconvex, unconstrained, nonlinear programming problem that can be solved by **stochastic gradient descent**, **quasi-Newton** methods, ... and **EKF** !

TRAINING FEEDFORWARD NEURAL NETWORKS VIA EKF

(Singhal, Wu, 1989) (Puskorius, Feldkamp, 1994)

- **Key idea**: treat parameter vector θ of the feedforward neural network as a **constant state**

$$\begin{cases} \theta_{k+1} & = & \theta_k + \eta_k \\ y_k & = & f(x_k, \theta_k) + \zeta_k \end{cases}$$

and use EKF to estimate θ_k **on line** from a streaming dataset $\{x_k, y_k\}$

- Ratio $\mathrm{Var}[\eta_k]/\,\mathrm{Var}[\zeta_k]$ is related to the **learning-rate**

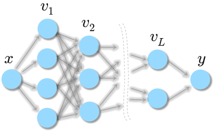
- Initial matrix $(P_{0|-1})^{-1}$ is related to **quadratic regularization** on θ

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, ...)



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

$$\theta = (A_1, b_1, \ldots, A_L, b_L)$$

- **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

GRADIENT DESCENT METHODS FOR TRAINING RNNS

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

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

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

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

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

(Kingma, Ba, 2015)

GRADIENT DESCENT METHODS FOR TRAINING RNNS

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

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

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

- Newton's method: very fast (2^{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 $\nabla V(\theta_x{}^t, \theta_y{}^t, x_0^t)$

TRAINING RNNS VIA EXTENDED KALMAN FILTERING

TRAINING RNNS BY EKF

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

- Iterating **Extended Kalman Filter** (**EKF**) based on the following model

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

$$Q = \mathrm{Var}[\left[\begin{smallmatrix}\xi_k \\ \eta_k\end{smallmatrix}\right]] \qquad R = \mathrm{Var}[\zeta_k] \qquad P_0 = \mathrm{Var}\left[\left[\begin{smallmatrix}\theta_x \\ \theta_y \\ x_0\end{smallmatrix}\right]\right]$$

= applying Newton's method incrementally to solve the relaxed problem

$$\min_{\substack{\theta_x, \theta_y \\ x_0, x_1, \ldots, x_{N-1}}} \left\| \left[\begin{smallmatrix}\theta_x \\ \theta_y \\ x_0\end{smallmatrix}\right] \right\|_{P_0^{-1}}^2 + \sum_{k=0}^{N-1} \|y_k - f_y(x_k, \theta_y)\|_{R^{-1}}^2 + \sum_{k=0}^{N-2} \left\| \left[\begin{smallmatrix} x_{k+1} - f_x(x_k, u_k, \theta_x) \\ \theta_{k+1} - \theta_k \end{smallmatrix}\right] \right\|_{Q^{-1}}^2$$

(Humpherys, Redd, West, 2012)

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

- The inverse of the initial matrix P_0 is related to **ℓ_2-penalty** on θ_x, θ_y, and x_0

- **Generalization**: train via **Moving Horizon Estimation** (MHE)

  (Løwenstein, Bernardini, Bemporad, Fagiano, 2023)

TRAINING RNNS BY EKF

(Bemporad, 2023)

- EKF can be generalized to handle **general strongly convex and smooth** losses $\ell(y_k, \hat{y}_k)$ by taking a local quadratic approximation of the loss around $\hat{y}_k$:

$$\begin{array}{rcl}\ell(y_k, \hat{y}) & \approx & \frac{1}{2}\Delta y' H(k)\Delta y + \phi_k'\Delta y + \text{const} \\ & = & \frac{1}{2}\left\|y_k - H^{-1}(k)\phi_k - \hat{y}\right\|_{H(k)}^2 + \text{const}\end{array}$$

$$\Delta y = \hat{y} - \hat{y}_k,\ \phi_k = \frac{\partial \ell(y_k, \hat{y}_k)}{\partial \hat{y}}$$

$$H(k) = \frac{\partial^2 \ell(y_k, \hat{y}_k)}{\partial \hat{y}_k^2}$$

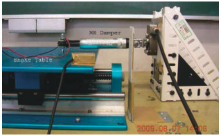
- Strongly convex smooth regularization $r(x_0, \theta_x, \theta_y)$ can be handled similarly

- 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

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

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)

TRAINING RNNS BY EKF - EXAMPLES

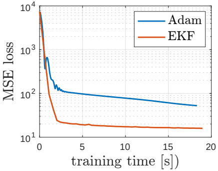


- **Dataset**: **magneto-rheological fluid damper**
  3499 I/O data (Wang, Sano, Chen, Huang, 2009)

- N=2000 data used for training, 1499 for testing the model

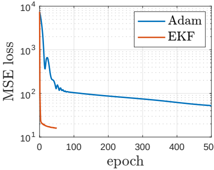
- Same data used in NNARX modeling demo of SYS-ID Toolbox for MATLAB

- **RNN model**: **4** hidden states, shallow state-update and output functions
  6 neurons, **atan** activation, I/O feedthrough



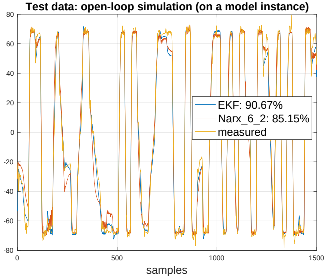
- Compare with **gradient descent** (Adam)

  MATLAB+CasADi implementation (Apple M1 Max CPU)



TRAINING RNNS BY EKF - EXAMPLES

- **RNN model**: **4** states, shallow NNs with **6 neurons** each, **atan** activation, I/O feedthrough

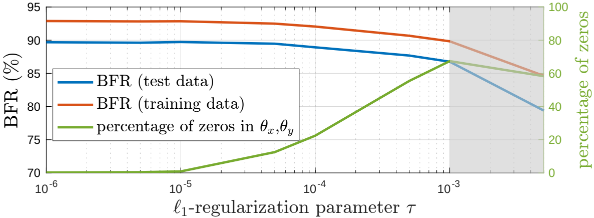


- Compare BFR wrt NNARX model (SYS-ID TBX):

EKF = **92.82**, Adam = **89.12**, NNARX(6,2) = **88.18** (**training**)

EKF = **89.78**, Adam = **85.51**, NNARX(6,2) = **85.15** (**test**)

- Repeat training with ℓ_1-penalty $\tau \left\| \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} \right\|_1$



TRAINING LSTMS BY EKF - EXAMPLES

- Use EKF to train **Long Short-Term Memory** (**LSTM**) model

(Hochreiter, Schmidhuber, 1997) (Bonassi et al., 2020)

$$\begin{aligned}
x_a(k+1) &= \sigma_G(W_F u(k) + U_f x_b(k) + b_f) \odot x_a(k) \\
&\quad + \sigma_G(W_I u(k) + U_I x_b(k) + b_I) \odot \sigma_C(W_C u(k) + U_C x_b(k) + b_C) \\
x_b(k+1) &= \sigma_G(W_O u(k) + U_O x_b(k) + b_O) \odot \sigma_C(x_a(k+1)) \\
y(k) &= f_y(x_b(k), u(k), \theta_y)
\end{aligned}$$

gate activation fcn $\sigma_G(\alpha) = \frac{1}{1+e^{-\alpha}}$, cell activation fcn $\sigma_C(\alpha) = \tanh(\alpha)$

- Training results (mean and std over 20 runs):

	BFR	Adam	EKF
RNN	training	89.12 (1.83)	**92.82** (0.33)
$n_\theta = 107$	test	85.51 (2.89)	**89.78** (0.58)
LSTM	training	89.60 (1.34)	92.63 (0.43)
$n_\theta = 139$	test	85.56 (2.68)	88.97 (1.31)

- EKF training **applicable to arbitrary classes** of black/gray box recurrent models!

TRAINING RNNS BY EKF - EXAMPLES

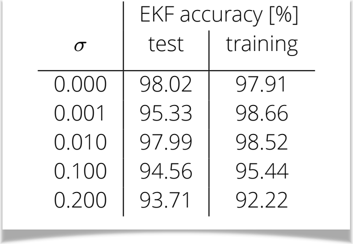
- Dataset: 2000 I/O data of linear system with **binary outputs**

$$
\begin{aligned}
x(k+1) &= \begin{bmatrix} .8 & .2 & -.1 \\ 0 & .9 & .1 \\ .1 & -.1 & .7 \end{bmatrix} x(k) + \begin{bmatrix} -1 \\ .5 \\ 1 \end{bmatrix} u(k) + \xi(k) & \mathrm{Var}[\xi_i(k)] = \sigma^2 \\
y(k) &= \begin{cases} 1 & \text{if } \left[\,-2\ 1.5\ 0.5\,\right] x(k) - 2 + \zeta(k) \geq 0 \\ 0 & \text{otherwise} \end{cases} & \mathrm{Var}[\zeta(k)] = \sigma^2
\end{aligned}
$$

- N=1000 data used for training, 1000 for testing the model

- Train **linear state-space model** with 3 states and **sigmoidal output** function

$$f_1^y(y) = 1/(1+e^{-A_1^y[x'(k)\ u(k)]'-b_1^y})$$



σ	EKF accuracy [%] test	EKF accuracy [%] training
0.000	98.02	97.91
0.001	95.33	98.66
0.010	97.99	98.52
0.100	94.56	95.44
0.200	93.71	92.22

- Training loss: (modified) **cross-entropy** loss

$$\ell_{\mathrm{CE}\epsilon}(y(k),\hat{y}) = \sum_{i=1}^{n_y} -y_i(k)\log(\epsilon+\hat{y}_i) - (1-y_i(k))\log(1+\epsilon-\hat{y}_i)$$

EXAMPLE: MPC OF ETHYLENE OXIDATION PLANT

- **Chemical process** = **oxidation of ethylene to ethylene oxide** in a nonisothermal continuously stirred tank reactor (CSTR)

$$
\begin{aligned}
&C_2H_4 + \tfrac{1}{2}O_2 \to C_2H_4O\\
&C_2H_4 + 3O_2 \to 2CO_2 + 2H_2O\\
&C_2H_4O + \tfrac{5}{2}O_2 \to 2CO_2 + 2H_2O
\end{aligned}
$$

- **Nonlinear model** (dimensionless variables): (Durand, Ellis, Christofides, 2016)

$$
\left\{
\begin{array}{rcl}
\dot{x}_1 & = & u_1(1 - x_1 x_4)\\
\dot{x}_2 & = & u_1(u_2 - x_2 x_4) - A_1 e^{\frac{\gamma_1}{x_4}}(x_2 x_4)^{\frac{1}{2}} - A_2 e^{\frac{\gamma_2}{x_4}}(x_2 x_4)^{\frac{1}{4}}\\
\dot{x}_3 & = & -u_1 x_3 x_4 + A_1 e^{\frac{\gamma_1}{x_4}}(x_2 x_4)^{\frac{1}{2}} - A_3 e^{\frac{\gamma_3}{x_4}}(x_3 x_4)^{\frac{1}{2}}\\
\dot{x}_4 & = & \frac{u_1(1-x_4)+B_1 e^{\frac{\gamma_1}{x_4}}(x_2 x_4)^{\frac{1}{2}}+B_2 e^{\frac{\gamma_2}{x_4}}(x_2 x_4)^{\frac{1}{4}}}{x_1}\\
 & & + \frac{B_3 e^{\frac{\gamma_3}{x_4}}(x_3 x_4)^{\frac{1}{2}}-B_4(x_4-T_c)}{x_1}\\
y & = & x_3
\end{array}
\right.
$$

x_1 = gas density
x_2 = ethylene concentration
x_3 = ethylene oxide concentration
x_4 = temperature in reactor

u_1 = feed volumetric flow rate
u_2 = ethylene concentration in feed

- u_1 = manipulated variables, x_3 = controlled output, u_2 = measured disturbance

RNN MODEL OF ETHYLENE OXIDATION PLANT

- Train a black-box **recurrent neural-network** model

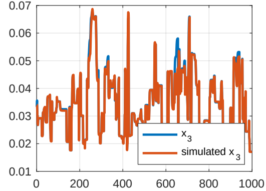
$$\begin{aligned} x_{k+1} &= \mathcal{N}_x(x_k, u_k) \\ y_k &= \mathcal{N}_y(x_k) \end{aligned}$$

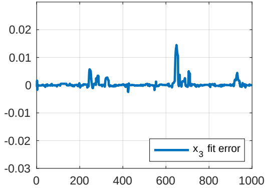
1,000 training samples $\{u_k, y_k\}$, sample time $T_s = 5$ s

2 layers (6 neurons, 4 neurons), sigmoid activation

$\rightarrow$ **95 coefficients**

x_3 (validation data)





sample

- NN model trained in MATLAB by EKF (Bemporad, 2023)

CPU time ≈ **12.58 s** [Apple M1 Max]

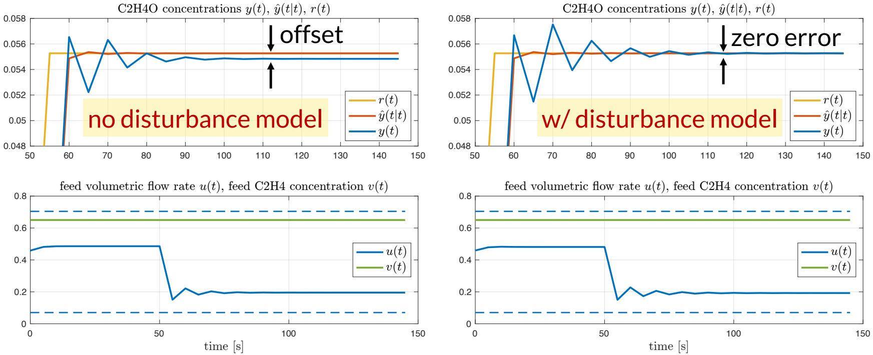
- Model validated on 1000 samples

BFR (Best Fit Rate)	training	test
	95.1611	84.3623

MPC OF ETHYLENE OXIDATION PLANT

- **Nonlinear MPC**: $\min \sum_{k=0}^{9} 10(y_{k+1} - r_{k+1})^2 + \frac{1}{10}(u_{1,k} - u_{1,k-1})^2$ subject to RNN model and input constraints $0.0704 \leq u_k \leq 0.7042$

- **EKF** used to estimate the hidden state x_k and, possibly, the disturbance d_k


offset

zero error

no disturbance model

w/ disturbance model

- Model mismatch compensated by **output integrator** in steady-state

ADAPTIVE NONLINEAR MPC BASED ON EKF

- By combining **online EKF-based learning** of the model parameters with **nonlinear MPC** we get an **adaptive nonlinear MPC** controller

- Do we really need to train the full model online?

- **Output integrators**: only update the **bias term** d_k on the output. Conceived to track **constant** set-points

- Can we train more general **nonlinear disturbance models** to better track **time-varying** references?

NONLINEAR DISTURBANCE MODELS FOR MPC

(Krupa, Zanon, Bemporad, *arXiv*, 2023)

- Consider the following general prediction model with unmeasured disturbance:

$$\begin{aligned} x(k+1) &= f(x(k), u(k), d(k)) \\ d(k) &= h(x(k), u(k), \theta(k)) \\ y(k) &= g(x(k), d(k)) \end{aligned}$$

- **Key idea**: only train the **disturbance model** online to refine the prediction model only where the system is operating

- The nominal model f, g is trained offline and frozen

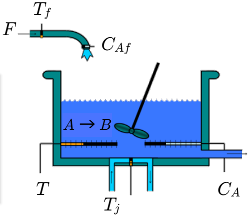
- **Motivation**: train the full model online may be difficult (lack of excitation, catastrophic forgetting, computational demand, etc.)

- Under certain assumptions, we can show that the tracking error $y(k) - r(k)$ **asymptotically converges to zero**, **even if $r(k)$ is not constant**

EXAMPLE: CSTR PROCESS

- MPC control of a diabatic **continuous stirred tank reactor (CSTR)**

- Process model is nonlinear (Seborg, Edgar, Mellichamp, 2004)



$$\begin{aligned}\frac{dC_A}{dt} &= \frac{F}{V}(C_{Af}-C_A) - C_A k_0 e^{-\frac{\Delta E}{RT}}\\ \frac{dT}{dt} &= \frac{F}{V}(T_f - T) + \frac{UA}{\rho C_p V}(T_j - T) - \frac{\Delta H}{\rho C_p} C_A k_0 e^{-\frac{\Delta E}{RT}}\end{aligned}$$

  - T : temperature inside the reactor $[K]$ (state)
  - C_A : concentration of the reactant in the reactor $[kgmol/m^3]$ (state)
  - T_j : jacket temperature $[K]$ (input)
  - T_f : feedstream temperature $[K]$ (measured disturbance)
  - C_{Af} : feedstream concentration $[kgmol/m^3]$ (measured disturbance)

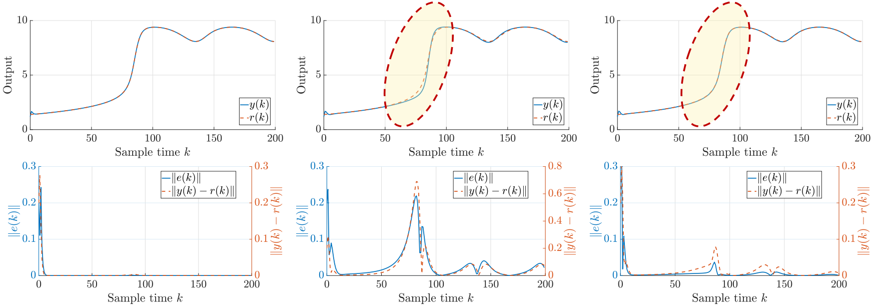
- Objective: **manipulate** T_j to **regulate** C_A on desired setpoint

EXAMPLE: CSTR PROCESS

- **Model**: nominal (white-box) NL model with slightly different model coefficients and three different disturbance models:
  - **Constant Disturbance Model**: $y = v + d, \theta = d$
  - **Polynomial Disturbance Model**: a polynomial function $d_x = h_x(x, u, \theta)$ entering the white-box model in a way that, for a suitable (unknown) θ, it matches the exact CSTR dynamics
  - **Feedforward Neural Network**: perturb both $y(k)$ and $x(k+1)$ by setting d_x = FNN with input (x, u), 2 layers with 6 neurons each, sigmoid activation, d_y = FNN with input x, single layer with 4 neurons. **Total # parameters = 97**
- **NLMPC**: $N = 5$, cost = $(x - x_r)'(x - x_r) + (u - u_r)'(u - u_r)$, terminal constraint $x(k+N) = x_r(k+N)$, no other constraints.

EXAMPLE: CSTR PROCESS

- Generic trackable reference signal $r(k)$ (**w/ preview**)



(a) White-box disturbance model (b) Constant disturbance model (c) FNN disturbance model

- Constant disturbance model is worse than FNN disturbance model, especially when $r(k)$ changes rapidly

TRAINING RNNS VIA SEQUENTIAL LEAST SQUARES

TRAINING RNNS BY SEQUENTIAL LEAST-SQUARES

(Bemporad, 2023)

- RNN training problem = **optimal control** problem:

$$\begin{array}{rl} \displaystyle\min_{\theta_x,\theta_y,x_0,x_1,\ldots,x_{N-1}} & r(x_0,\theta_x,\theta_y)+\displaystyle\sum_{k=0}^{N-1}\ell(y_k,\hat{y}_k) \\ \text{s.t.} & x_{k+1}=f_x(x_k,u_k,\theta_x) \\ & \hat{y}_k=f_y(x_k,u_k,\theta_y) \end{array}$$

inputs $= \theta_x, \theta_y, x_0$

output $= \hat{y}$

reference $= y_k$

meas. dist. $= u_k$

  - $r(x_0,\theta_x,\theta_y)$ = input penalty
  - $\ell(y_k,\hat{y}_k)$ = output penalty
  - prediction horizon = N steps, control horizon = 1 step

- **Linearized model**: given a current guess ${\theta_x}^h, {\theta_y}^h, x_0^h, \ldots, x_{N-1}^h$, approximate

$$\begin{array}{rcl} \Delta x_{k+1} & = & (\nabla_x f_x)'\Delta x_k + (\nabla_{\theta_x} f_x)'\Delta\theta_x \\ \Delta y_k & = & (\nabla_x f_y)'\Delta x_k + (\nabla_{\theta_y} f_y)'\Delta\theta_y \end{array}$$

TRAINING RNNS BY SEQUENTIAL LEAST-SQUARES

(Bemporad, 2023)

- Linearized dynamic response: $\Delta x_k = M_{kx}\Delta x_0 + M_{k\theta_x}\Delta\theta_x$

$$\begin{aligned} M_{0x} &= I, \quad M_{0\theta_x} = 0 \\ M_{(k+1)x} &= \nabla_x f_x(x_k^h, u_k, {\theta_x}^h) M_{kx} \\ M_{(k+1)\theta_x} &= \nabla_x f_x(x_k^h, u_k, {\theta_x}^h) M_{k\theta_x} + \nabla_{\theta_x} f_x(x_k^h, u_k, {\theta_x}^h) \end{aligned}$$

- Take 2^{nd}-order expansion of the loss ℓ and regularization term r

- Solve **least-squares** problem to get increments $\Delta x_0, \Delta\theta_x, \Delta\theta_y$

- Update $x_0^{h+1}, {\theta_x}^{h+1}, {\theta_y}^{h+1}$ by applying either a
  - **line-search** (LS) method based on Armijo rule
  - or a **trust-region** method (Levenberg-Marquardt) (LM)

- The resulting training method is a **Generalized Gauss-Newton** method

  → very good convergence properties (Messerer, Baumgärtner, Diehl, 2021)

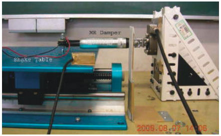
- No guarantee to converge to a global minimum (multiple runs may be required)

TRAINING RNNS BY SEQUENTIAL LS AND ADMM

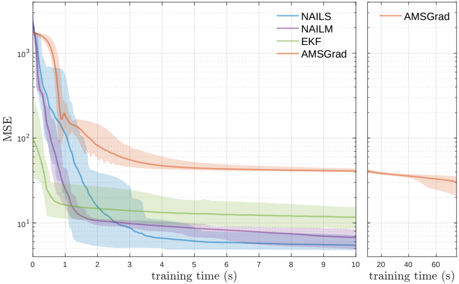
- **Example**: magneto-rheological fluid damper

  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 and range over 20 runs from different random initial weights

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

NAILS = GNN method with **line search**

NAILM = GNN method with **LM steps**

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)

TRAINING RNNS BY SEQUENTIAL LS AND ADMM

- We also want to handle **non-smooth**/**non-convex** regularization terms

$$\begin{array}{rl}\min_{\theta_x,\theta_y,x_0} & r(x_0,\theta_x,\theta_y)+\sum_{k=0}^{N-1}\ell(y_k,f_y(x_k,\theta_y))+g(\theta_x,\theta_y)\\ \text{s.t.} & x_{k+1}=f_x(x_k,u_k,\theta_x)\end{array}$$

E.g.: $g(\theta_x,\theta_y)=\tau(\|\theta_x\|_1+\|\theta_y\|_1)$ (Lasso regularization)

- **Idea**: use **alternating direction method of multipliers** (ADMM) by splitting

$$\begin{array}{rl}\min_{\theta_x,\theta_y,x_0,\nu_x,\nu_y} & r(x_0,\theta_x,\theta_y)+\sum_{k=0}^{N-1}\ell(y_k,f_y(x_k,\theta_y))+g(\nu_x,\nu_y)\\ \text{s.t.} & x_{k+1}=f_x(x_k,u_k,\theta_x)\\ & \begin{bmatrix}\nu_x\\ \nu_y\end{bmatrix}=\begin{bmatrix}\theta_x\\ \theta_y\end{bmatrix}\end{array}$$

TRAINING RNNS BY SEQUENTIAL LS AND ADMM

- ADMM + Seq. LS = **NAILS** algorithm (Nonconvex ADMM Iterations and Sequential LS)

$$\begin{bmatrix} x_0^{t+1} \\ {\theta_x}^{t+1} \\ {\theta_y}^{t+1} \end{bmatrix} = \arg\min_{x_0,\theta_x,\theta_y} V(x_0,\theta_x,\theta_y) + \frac{\rho}{2}\left\|\begin{bmatrix} \theta_x - \nu_x^t + w_x^t \\ \theta_y - \nu_y^t + w_y^t \end{bmatrix}\right\|_2^2$$

(sequential) LS

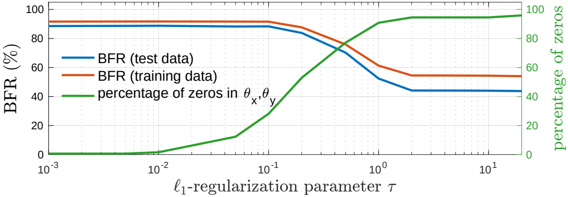
$$\begin{bmatrix} \nu_x^{t+1} \\ \nu_y^{t+1} \end{bmatrix} = \text{prox}_{\frac{1}{\rho}g}({\theta_x}^{t+1} + w_x^t, {\theta_y}^{t+1} + w_y^t)$$

proximal step

$$\begin{bmatrix} w_x^{t+1} \\ w_y^{t+1} \end{bmatrix} = \begin{bmatrix} w_x^h + {\theta_x}^{t+1} - \nu_x^{t+1} \\ w_y^h + {\theta_y}^{t+1} - \nu_y^{t+1} \end{bmatrix}$$

update dual vars

- ADMM + Levenberg-Marquardt steps = **NAILM** algorithm
- Fluid-damper example: **Lasso regularization** $g(\nu_x,\nu_y) = \tau(\|\nu_x\|_1 + \|\nu_y\|_1)$



(mean results over 20 runs from different initial weights)

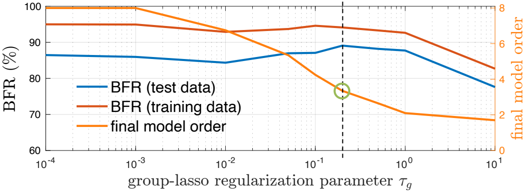
TRAINING RNNS BY SEQUENTIAL LS AND ADMM

- Fluid-damper example: **Lasso regularization** $g(\nu_x,\nu_y)=0.2(\|\nu_x\|_1+\|\nu_y\|_1)$

training algorithm	BFR training	BFR test	sparsity %	CPU time	# epochs
NAILS	91.00 (1.66)	87.71 (2.67)	65.1 (6.5)	11.4 s	250
NAILM	91.32 (1.19)	87.80 (1.86)	64.1 (7.4)	11.7 s	250
AMSGrad	91.04 (0.47)	88.32 (0.80)	16.8 (7.1)	64.0 s	2000
Adam	90.47 (0.34)	87.79 (0.44)	8.3 (3.5)	63.9 s	2000
DiffGrad	90.05 (0.64)	87.34 (1.14)	7.4 (4.5)	63.9 s	2000
EKF	89.27 (1.48)	86.67 (2.71)	47.9 (9.1)	13.2 s	50

≈ same fit than SGD/EKF but sparser models and faster (Apple M1 Pro)

- Fluid-damper example: **group-Lasso regularization** $g(\nu_i^g)=\tau_g\sum_{i=1}^{n_x}\|\nu_i^g\|_2$ to zero entire rows/columns and **reduce the state-dimension** automatically



good choice: $n_x=3$ (best fit on test data)

TRAINING RNNS BY SEQUENTIAL LS AND ADMM

(Bemporad, 2023)

- Fluid-damper example: **quantization** of θ_x, θ_y for simplifying model arithmetic +leaky-ReLU activation function

$$g(\theta_i) = \begin{cases} 0 & \text{if } \theta_i \in \mathcal{Q} \\ +\infty & \text{otherwise} \end{cases}$$

$\mathcal{Q}$ = multiples of 0.1 between -0.5 and 0.5

  - BFR = **84.36** (training), **78.43** (test) $\leftarrow$ **NAILS w/ quantization**
  - BFR = **17.64** (training), **12.79** (test) $\leftarrow$ **no ADMM, just quantize after training**
  - Training time: $\approx$ 12 s (w/ quantization), 7 s (no ADMM)

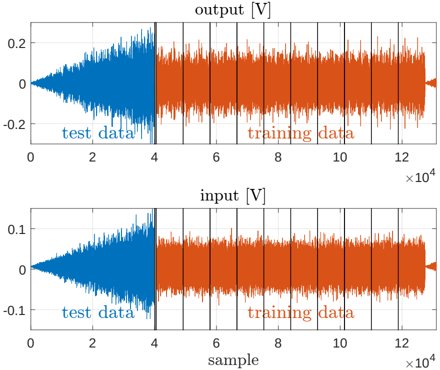
- **Note**: no convergence to a global minimum is guaranteed

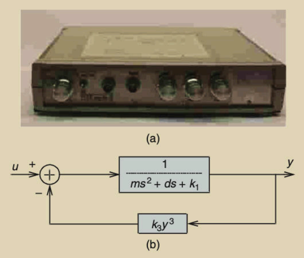
- **NAILS/LM** = flexible & efficient algorithm for training **control-oriented RNNs**

TRAINING RNNS - SILVERBOX BENCHMARK

(Wigren, Schoukens, 2013)

- **Silverbox benchmark** (Duffin oscillator): 10 traces (≈8600 samples each) used for training, 40000 for testing





(Schoukens, Ljung, 2019)

Data download: `http://www.nonlinearbenchmark.org`

TRAINING RNNS - SILVERBOX BENCHMARK

(Bemporad, 2023)

- **RNN model**: 8 states, 3 layers of 8 neurons, atan activation, no I/O feedthrough
- **Initial-state**: **encode** x_0 as the output of a NN with atan activation, 2 layers of 4 neurons, receiving 8 past inputs and 8 past outputs

$$\begin{array}{rl} \min_{\theta_{x_0},\theta_x,\theta_y} & r(\theta_{x_0},\theta_x,\theta_y)+\sum_{j=1}^{M}\sum_{k=0}^{N-1}\ell(y_k^j,\hat{y}_k^j) \\ \text{s.t.} & x_{k+1}^j=f_x(x_k^j,u_k^j,\theta_x),\ \hat{y}_k^j=f_y(x_k^j,u_k^j,\theta_y) \\ & x_0^j=f_{x_0}(v^j,\theta_{x_0}) \end{array} \qquad v=\begin{bmatrix} y_{-1}\\ \vdots\\ y_{-8}\\ u_{-1}\\ \vdots\\ u_{-8}\end{bmatrix}$$

[cf. (Beintema, Toth, Schoukens, 2021)]

- ℓ_2-regularization: $r(\theta_{x_0},\theta_x,\theta_y)=\frac{0.01}{2}(\|\theta_x\|_2^2+\|\theta_y\|_2^2)+\frac{0.1}{2}\|\theta_{x_0}\|_2^2$
- Total number of parameters $n_{\theta_x}+n_{\theta_y}+n_{\theta_{x_0}}=296+225+128=$ **649**
- Training: use NAILM over **150 epochs**

TRAINING RNNS - SILVERBOX BENCHMARK

- Identification results on test data [1]:

identification method	RMSE [mV]	BFR [%]
ARX (ml) [1]	16.29 [4.40]	69.22 [73.79]
NLARX (ms) [1]	8.42 [4.20]	83.67 [92.06]
NLARX (mlc) [1]	1.75 [1.70]	96.67 [96.79]
NLARX (ms8c50) [1]	1.05 [0.30]	98.01 [99.43]
Recurrent LSTM model [2]	2.20	95.83
SS encoder [3] ($n_x = 4$)	[1.40]	[97.35]
NAILM	0.35	99.33

[1] Ljung, Zhang, Lindskog, Juditski, 2004

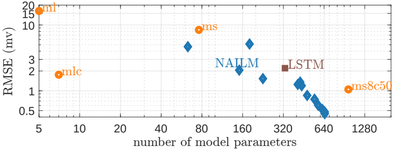
[2] Ljung, Andersson, Tiels, Schön, 2020

[3] Beintema, Toth, Schoukens, 2021

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(y_k - \hat{y}_k)^2}$$

- NAILM training time ≈ 400 s (MATLAB+CasADi on Apple M1 Max CPU)

- Repeat training with ℓ_1-regularization:



[1] Trained RNN: `http://cse.lab.imtlucca.it/~bemporad/shared/silverbox/rnn888.zip`

LINEAR AND NONLINEAR IDENTIFICATION VIA L-BFGS

SYSTEM IDENTIFICATION PROBLEM

- Class of dynamical models with n_x states, n_u inputs, n_y outputs:

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

Special cases:

linear model, RNN, ...

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

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

$$z = \left[\begin{smallmatrix} x_0 \\ \Theta \end{smallmatrix}\right]$$

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

- 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**

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

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

ℓ_1-regularization

group-Lasso penalty

- Examples of **group-Lasso penalties**:

  $m = n_x$ and I_i selected to reduce the **number of states**

  $m = n_u$ and I_i selected to reduce the **number of inputs**

HANDLING NON-SMOOTH REGULARIZATION TERMS

(Bemporad, 2024)

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

$$\min_x f(x)+\tau\|x\|_1+r(x)$$

→

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

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

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 $y_i,z_j>0$

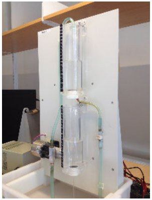
Example: $r(x)$ = group-Lasso penalty + constraint $y,z\geq\epsilon$ = machine precision

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

	R^2 (training)			R^2 (test)			
n_x	lbfgs	sippy[2]	MATLAB[3]	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]

NLP with bounds solved in **JAX/JAXOPT** using the **L-BFGS-B** solver (Byrd, Lu, Nocedal, Zhu, 1995)

```
pip install jax-sysid
```

```
github.com/bemporad/jax-sysid
```

[2] (Armenise, Vaccari, Bacci Di Capaci, Pannocchia, 2018)

[3] (Ljung, SYS-ID Toolbox)

JAX-SYSID LIBRARY

(Bemporad, 2024)

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

```python
from jax_sysid.models import LinearModel
from jax_sysid.utils import compute_scores

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()
```

- Python code for testing the model:

```python
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)
```

JAX-SYSID LIBRARY

(Bemporad, 2024)

- Sample Python code to identify a **nonlinear RNN** model:

jax-sysid

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

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

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

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

output function, $y(k)$

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

A  = 0.5*np.eye(nx)
     ...
b4 = np.zeros(ny) # Parameter initialization:
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)
model.optimization(adam_epochs=1000, lbfgs_epochs=1000)
model.fit(Y, U)

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

EXAMPLE: LINEAR SYSTEM IDENTIFICATION

- Synthetic data generated by the **cascaded 2x2 linear system**

$$x_{k+1} = \begin{bmatrix} 0.96 & 0.26 & 0.04 & 0 & 0 & 0 \\ -0.26 & 0.70 & 0.26 & 0 & 0 & 0 \\ 0 & 0 & 0.93 & 0.32 & 0.07 & 0 \\ 0 & 0 & -0.32 & 0.61 & 0.32 & 0 \\ 0 & 0 & 0 & 0 & 0.90 & 0.38 \\ 0 & 0 & 0 & 0 & -0.38 & 0.52 \end{bmatrix} x_k + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0.07 & 0 \\ 0.32 & 0 \\ 0 & 0.10 \\ 0 & 0.38 \end{bmatrix} u_k + \xi_k$$

$$y_k = \begin{bmatrix} x_1 \\ x_3 \end{bmatrix} + \eta_k$$

$\xi_{ki}, \eta_{kj} \in \mathcal{N}(0, 0.01)$

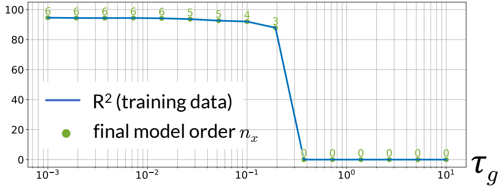
N=2000 training data

$\{(u_k, y_k)\}$

- Group-lasso penalty for **model-order reduction**:

```
model.group_lasso_x()
```

$$\min_{\theta_x,\theta_y,x_0} \frac{1}{1000}\|z\|_2^2 + 10^{-16}\|z\|_1 + \tau_g \sum_{i=1}^{n_x} \left\| \begin{bmatrix} A'_{i,:} \\ A_{:,i} \\ B'_{i,:} \\ C_{:,i} \end{bmatrix} \right\|_2 + \frac{1}{N}\sum_{k=0}^{N-1} \|y_k - Cx_k\|_2^2$$



best results out of 10 runs

CPU time ≈ 3.85 s per run

[Apple M1 Max]

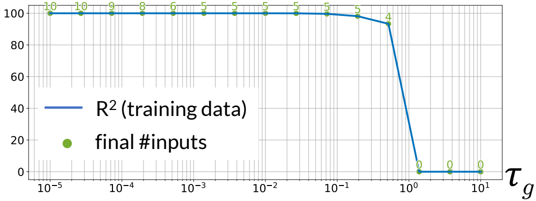
EXAMPLE: LINEAR SYSTEM IDENTIFICATION

- Synthetic data generated by a **random linear system** with $n_x = 3$ states, $n_u = 10$ inputs, $n_y = 1$ outputs, noise in $\mathcal{N}(0, 0.01)$, $N = 10000$ training data

- The last 5 columns of the B matrix are **1000x smaller** than the first 5

- Group-lasso penalty for **input selection**: `model.group_lasso_u()`

$$\min_{\theta_x,\theta_y,x_0} 10^{-8}\|z\|_2^2 + 10^{-16}\|z\|_1 + \tau_g \sum_{i=1}^{n_u} \|B_{:,i}\|_2 + \frac{1}{N}\sum_{k=0}^{N-1} \|y_k - Cx_k\|_2^2$$



best results out of 10 runs

CPU time ≈ 3.71 s per run

[Apple M1 Max]

- Can be useful to identify **Hammerstein models** using basis functions on u

LINEAR SYSTEM IDENTIFICATION W/ STABILITY CONSTRAINTS

- We try enforcing $\|A\|_2 \leq 1$ by adding the loss $\rho_A \max\{\|A\|_2^2 - 1 + \epsilon_A, 0\}^2$

- Example: 1000 training + 1000 test data generated by the **unstable LTI** system

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

$$y_k = Cx_k + z_k$$

where the entries of $B, C \in \mathcal{N}(0,1), \xi_{ki} \in \mathcal{N}(0, 0.01^2), \zeta_k \in \mathcal{N}(0, 0.05^2)$, and u_k uniformly generated in $[-\frac{1}{2}, \frac{1}{2}]$

- 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 ≈ **5.38 s** [Apple M1 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-LPV** (qLPV) models are defined by:

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

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

  where $p_k \in \mathbb{R}^{n_p}$ is the **scheduling parameter** vector, such as

$$p_{ki} = \frac{1}{1+e^{-f(x_k,u_k;\theta_i)}},\ i = 1,\ldots,n_p-1$$

  where $f(x_k,u_k;\theta_i)$ is a FNN with linear output layer and parameters θ_i

- qLPV models are a powerful class of control-oriented nonlinear models

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 setup:
  - warm start: identify LTI model (2000 L-BFGS iters)
  - 1000 Adam + 5000 L-BFGS iters for qLPV-SYSID
  - CPU time ≈ **20 s** [Apple M1 Max]

BFR (Best Fit Rate)	n_p	training	test
LTI	0	74.7374	74.9277
qLPV	1	94.5179	94.5059
qLPV	2	96.3040	94.3056
qLPV	3	96.5766	96.4442

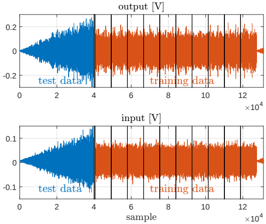
EXAMPLE: QUASI-LPV MODEL OF SILVERBOX BENCHMARK

(Bemporad, *NL-SYSID Workshop, 2024*)

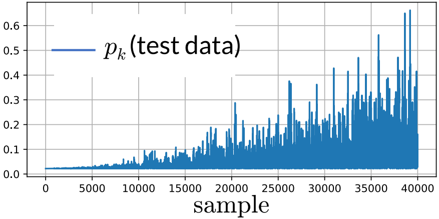
- **Quasi-LPV** model structure ($n_x = 8$ states):

$$\begin{aligned} x_{k+1} &= (A_0 + A_1 p_k)x_k + (B_0 + B_1 p_k)u_k \\ y_k &= Cx_k \\ p_k &= \mathrm{swish}(W_2\,\mathrm{swish}(W_1 x_k + b_1) + b_2) \end{aligned}$$

$$\mathrm{swish}(x) = \frac{x}{1+e^{-x}}$$



- Training setup:
  - ℓ_2-regularization ($\rho = 10^{-4}$)
  - warm start on first experiment (8,600 samples) 500 Adam + 500 L-BFGS iterations
  - 5000 L-BFGS iterations on full dataset (86,114 samples)
  - CPU time ≈ **265 s** [Apple M1 Max]



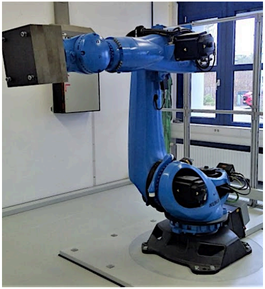
- RMSE on test data: **0.397 mV** (LTI model: **14.090 mV**)

  ($\|A_0\|_2 = 1.96$, $\|A_1\|_2 = 0.35$, $\|B_0\|_2 = 0.79$, $\|B_1\|_2 = 0.09$)

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 over-sampled** ($\|y_k - y_{k-1}\|$ 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`**



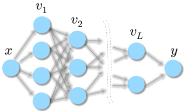
`nonlinearbenchmark.org`

RECURRENT NEURAL NETWORKS IN RESIDUAL FORM

(Bemporad, 2024)

- **Recurrent Neural Network** (RNN) model in **residual form**:

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



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

$$\theta = (A_1, b_1, \ldots, A_L, b_L)$$

- **Goal**: minimize **open-loop simulation error** under **elastic net** regularization

$$\min_{x_0, A, B, C, \theta_x, \theta_y} \frac{1}{N} \sum_{k=1}^{N} \|y_k - \hat{y}_k\|_2^2 + \frac{1}{2}\rho(\|\theta_x\|_2^2 + \|\theta_y\|_2^2) + \tau(\|\theta_x\|_1 + \|\theta_y\|_1)$$

s.t. model equations

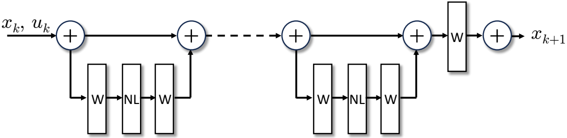
- ℓ_1**-regularization** introduced to reduce # model coefficients (=simpler model)

TRAINING RNN W/ ℓ_1-PENALTIES - INDUSTRIAL ROBOT

- Main **issues** with industrial robot benchmark:
  - **many parameters** to train, **large dataset** $\Rightarrow$ complex NLP
  - **high sensitivity** wrt weights (dynamics gets easily unstable)
  - **local minima** (solution depends on initial guess)
  - cannot easily use **mini-batches**: open-loop simulation cost is not separable, long-term memory effects present due to small sample time

- More general **residual networks** + ℓ_1/group-Lasso regularization possible

(Frascati, Bemporad, 2023)



SOLUTION APPROACH

(Bemporad, 2024)

1. Standard-scale I/O data for numerical reasons $u_i \leftarrow \frac{u_i-\mu_u^i}{\sigma_u^i}$, $y_i \leftarrow \frac{y_i-\mu_y^i}{\sigma_y^i}$ $i=1,\ldots,6$

2. Train (A,B,C,x_0) by **jax-sysid** (1000 L-BFGS iters) w/o ℓ_1-regularization ($x \in \mathbb{R}^{12}$) (CPU time: 9.12 s) [Apple M1 Max]

   For comparison: **n4sid** takes 36.21 s and gives lower R^2-scores on training & test data in MATLAB
   sippy fails

3. Fix (A,B,C) and train simple RESNET model with shallow NNs:

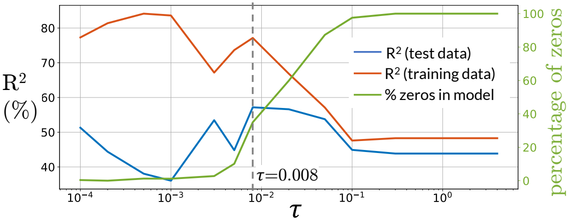
$$x_{k+1} = Ax_k + Bu_k + f_x(x_k,u_k,\theta_x), \qquad y_k = Cx_k + f_y(x_k,\theta_y)$$

- **Optimization**: to handle $\tau\|\theta\|_1$, use **jax-sysid** running 2000 **Adam** iters first (for warm-start) and then 2000 **L-BGFS-B** iters

INDUSTRIAL ROBOT BENCHMARK: RESULTS

- 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 on a single core per run [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

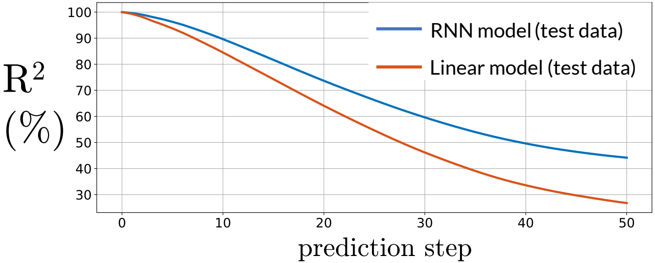
- **Pure Adam** vs **LBFG-B+Adam** vs **OWL-QN** (Andrew, Gao, 2007): ($\tau = 0.008$)

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

INDUSTRIAL ROBOT BENCHMARK: RESULTS

- Compute p-step ahead prediction $\hat{y}_{k+p|k}$, with hidden state $x_{k|k}$ estimated by an Extended Kalman Filter based on identified RNN model

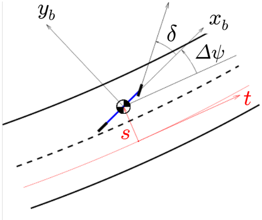


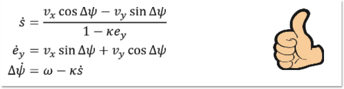
- This is a more relevant indicator of model quality for MPC purposes than open-loop simulation error $\hat{y}_{k|0} - y_k$

DEEP NONLINEAR MPC: AN EXAMPLE

DEEP NONLINEAR MPC FOR AUTONOMOUS DRIVING

- **Goal**: track desired longitudinal speed (v_y), lateral displacement (e_y) and orientation ($\Delta\Psi$)
- **Inputs**: wheel torque T_w and steering angle δ
- **Constraints**: on e_y and lateral displacement s (for obstacle avoidance) and manipulated inputs T_w, δ
- **Sampling time**: 100 ms
- **Model**: gray-box bicycle model





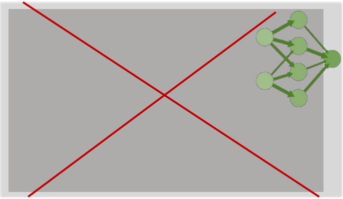
$$\dot{s} = \frac{v_x \cos\Delta\psi - v_y \sin\Delta\psi}{1 - \kappa e_y}$$

$$\dot{e}_y = v_x \sin\Delta\psi + v_y \cos\Delta\psi$$

$$\Delta\dot{\psi} = \omega - \kappa \dot{s}$$

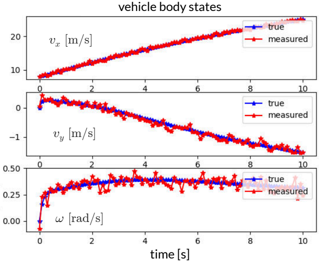
- **kinematics** is simple to model (white box)
- **tire forces** harder to model + **stiff** wheel slip ratio dynamics (k_f, k_r) $\Rightarrow$ small integration step required
- learn a **black-box neural-network model** !

(Boni, Capelli, Frascati @ODYS, 2021)



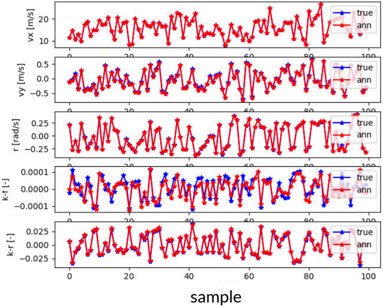
DEEP NONLINEAR MPC FOR AUTONOMOUS DRIVING

- ODYS Deep Learning Toolset used to learn a neural-network with input $(v_x, v_y, \omega, k_f, k_r, T_w, \delta)$ @k and output $(v_x, v_y, \omega, k_f, k_r)$ @$k+1$
- Data generated from high-fidelity simulation model with noisy measurements, sampled @10Hz
- Neural network model: **2 hidden layers, 55 neurons each**
- Advantages of black-box (neural network) model:
  - No physical model required describing tire-road interaction
  - directly learn the model in discrete-time ($T_s = 100$ ms)

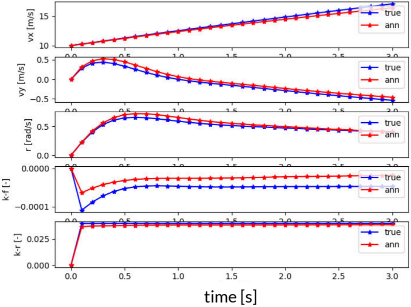


DEEP NONLINEAR MPC FOR AUTONOMOUS DRIVING

- Model validation on test data:



one-step ahead prediction on test data



open-loop predictions

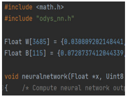
- C-code (network+Jacobians) automatically generated for ODYS MPC

Tensorflow
Keras
PyTorch
scikit-learn

ODYS-NN training

automatic
C-code gen

ODYS



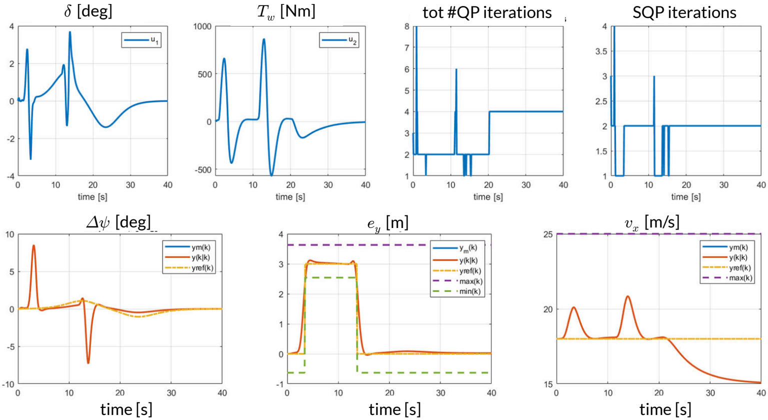
```
#include <math.h>
#include "odys_nn.h"

Float W[3685] = {0.030809202148441,
Float B[115] = {0.0728737412044339,

void neuralnetwork(Float *x, Uint8
{    /* Compute neural network out
```

Embedded MPC

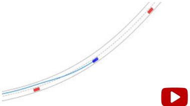
ODYS

DEEP NONLINEAR MPC FOR AUTONOMOUS DRIVING

- **Closed-loop MPC**: overtake vehicle #1, keep safety distance from vehicle #2

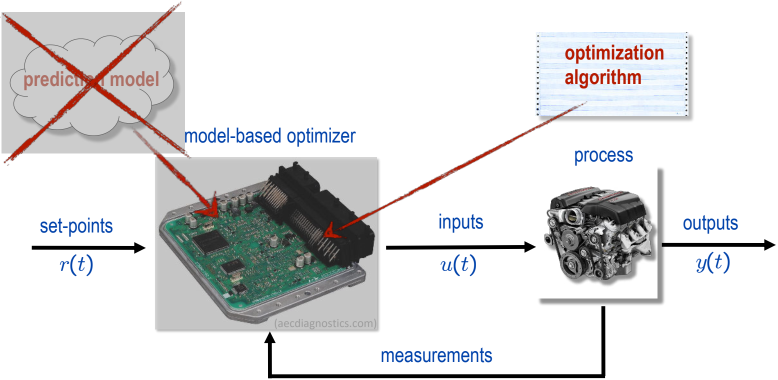


- Good reference tracking, constraints on e_y, v_x satisfied, smooth command action



DIRECT DATA-DRIVEN MPC

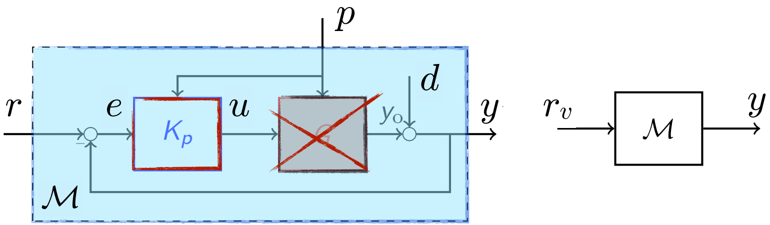
DIRECT DATA-DRIVEN MPC



- Can we design an MPC controller **without** first identifying a model of the **open-loop process** ?

DATA-DRIVEN DIRECT CONTROLLER SYNTHESIS

(Campi, Lecchini, Savaresi, 2002) (Formentin et al., 2015)

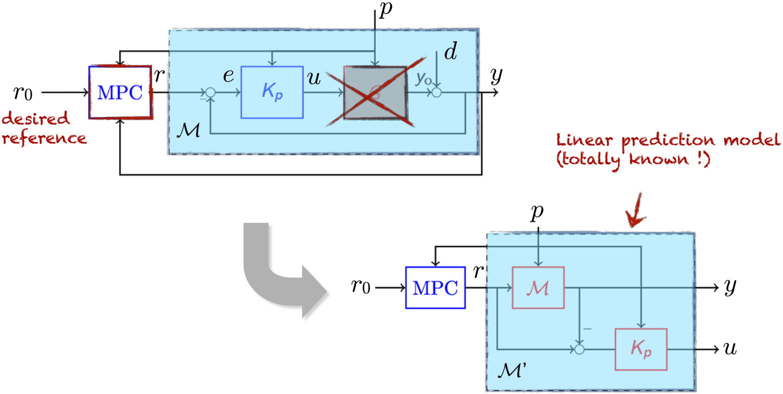


- Collect a set of **data** $\{u(t), y(t), p(t)\}, t = 1, \ldots, N$
- Specify a **desired closed-loop linear model** $\mathcal{M}$ from r to y
- Compute $r_v(t) = \mathcal{M}^{\#} y(t)$ from **pseudo-inverse model** $\mathcal{M}^{\#}$ of $\mathcal{M}$
- **Identify** linear (LPV) model K_p from $e_v = r_v - y$ (virtual tracking error) to u

DIRECT DATA-DRIVEN MPC

- Design a linear MPC (**reference governor**) to generate the reference r

  (Bemporad, Mosca, 1994) (Gilbert, Kolmanovsky, Tan, 1994)



desired reference

Linear prediction model (totally known !)

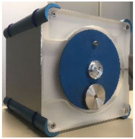
- MPC designed to handle input/output **constraints** and improve **performance**

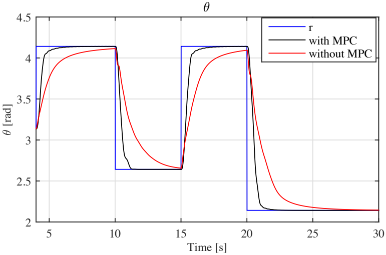
  (Piga, Formentin, Bemporad, 2017)

DIRECT DATA-DRIVEN MPC - AN EXAMPLE

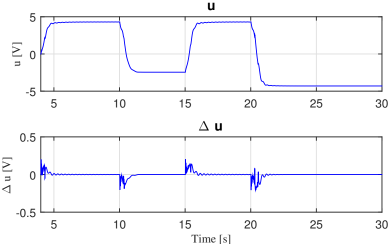
- Experimental results: MPC handles soft constraints on u, Δu and y

(motor equipment by courtesy of TU Delft)





desired tracking performance achieved

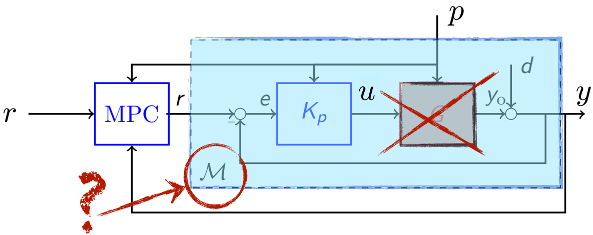


constraints on input increments satisfied

No open-loop process model was identified to design the MPC controller!

OPTIMAL DIRECT DATA-DRIVEN MPC

- **Question**: How to choose the reference model $\mathcal{M}$?



- Can we choose $\mathcal{M}$ from data so that K_p is an **optimal controller** ?

OPTIMAL DIRECT DATA-DRIVEN MPC

(Selvi, Piga, Bemporad, 2018)

- **Idea**: parameterize desired closed-loop model $\mathcal{M}(\theta)$ and optimize

$$\min_{\theta} J(\theta) = \frac{1}{N}\sum_{t=0}^{N-1} \underbrace{W_y(r(t) - y_p(\theta,t))^2 + W_{\Delta u}\Delta u_p^2(\theta,t)}_{\text{performance index}} + \underbrace{W_{\rm fit}(u(t) - u_v(\theta,t))^2}_{\text{identification error}}$$

- Evaluating $J(\theta)$ requires synthesizing $K_p(\theta)$ from data and simulating the nominal model and control law

$$y_p(\theta,t) = \mathcal{M}(\theta)r(t) \qquad u_p(\theta,t) = K_p(\theta)(r(t) - y_p(\theta,t))$$

$$\Delta u_p(\theta,t) = u_p(\theta,t) - u_p(\theta,t-1)$$

- Optimal θ obtained by solving a **(non-convex) nonlinear programming** problem

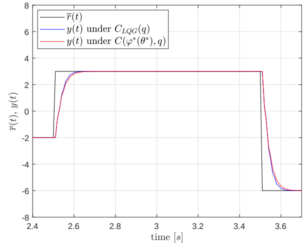
OPTIMAL DIRECT DATA-DRIVEN MPC

(Selvi, Piga, Bemporad, 2018)

- **Results**: linear process

$$G(z)=\frac{z-0.4}{z^2+0.15z-0.325}$$

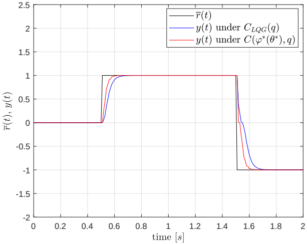
Data-driven controller **only 1.3% worse** than model-based LQR (=SYS-ID on same data + LQR design)



- **Results**: nonlinear (Wiener) process

$$\begin{aligned} y_L(t) &= G(z)u(t) \\ y(t) &= |y_L(t)|\arctan(y_L(t)) \end{aligned}$$

The data-driven controller is **24% better** than LQR based on identified open-loop model !



DATA-DRIVEN OPTIMAL POLICY SEARCH

DATA-DRIVEN OPTIMAL POLICY SEARCH

(Ferrarotti, Bemporad, 2019)

- Plant + environment dynamics (**unknown**):

$$s_{t+1} = h(s_t, p_t, u_t, d_t)$$

- s_t states of plant & environment
- p_t exogenous signal (e.g., reference)
- u_t control input
- d_t unmeasured disturbances

- **Control policy**: $\pi : \mathbb{R}^{n_s+n_p} \longrightarrow \mathbb{R}^{n_u}$ deterministic control policy

$$u_t = \pi(s_t, p_t)$$

- Closed-loop **performance** of an execution is defined as

$$\mathcal{J}_\infty(\pi, s_0, \{p_\ell, d_\ell\}_{\ell=0}^{\infty}) = \sum_{\ell=0}^{\infty} \rho(s_\ell, p_\ell, \pi(s_\ell, p_\ell))$$

$$\rho(s_\ell, p_\ell, \pi(s_\ell, p_\ell)) \ = \ \text{stage cost}$$

OPTIMAL POLICY SEARCH PROBLEM

- **Optimal policy:**

$$\begin{aligned}\pi^* &= \arg\min_\pi \mathcal{J}(\pi)\\ \mathcal{J}(\pi) &= \mathbb{E}_{s_0,\{p_\ell,d_\ell\}}\left[\mathcal{J}_\infty(\pi,s_0,\{p_\ell,d_\ell\})\right]\end{aligned}$$

expected performance

- **Simplifications**:
  - Finite parameterization: $\pi=\pi_K(s_t,p_t)$ with K = parameters to optimize
  - Finite horizon: $\mathcal{J}_L(\pi,s_0,\{p_\ell,d_\ell\}_{\ell=0}^{L-1})=\sum_{\ell=0}^{L-1}\rho(s_\ell,p_\ell,\pi(s_\ell,p_\ell))$

- Optimal policy search: use **stochastic gradient descent (SGD)**

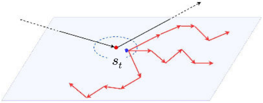
$$K_t \leftarrow K_{t-1}-\alpha_t\mathcal{D}(K_{t-1})$$

with $\mathcal{D}(K_{t-1})$ = descent direction

DESCENT DIRECTION

- The descent direction $\mathcal{D}(K_{t-1})$ is computed by generating:
  - N_s perturbations $s_0^{(i)}$ around the current state s_t
  - N_r random reference signals $r_\ell^{(j)}$ of length L,
  - N_d random disturbance signals $d_\ell^{(h)}$ of length L,

$$\boxed{\mathcal{D}(K_{t-1}) = \sum_{i=1}^{N_s} \sum_{j=1}^{N_p} \sum_{k=1}^{N_q} \nabla_K \mathcal{J}_L(\pi_{K_{t-1}}, s_0^{(i)}, \{r_\ell^{(j)}, d_\ell^{(k)}\})}$$



SGD step = mini-batch of size $M = N_s \cdot N_r \cdot N_d$

- Computing $\nabla_K \mathcal{J}_L$ requires predicting the effect of π over L future steps

- We use a **local linear model** just for computing $\nabla_K \mathcal{J}_L$, obtained by running **recursive linear system identification**

OPTIMAL POLICY SEARCH ALGORITHM

- At each step t:

  1. Acquire current s_t
  2. Recursively update the local linear model
  3. Estimate the direction of descent $\mathcal{D}(K_{t-1})$
  4. Update policy: $K_t \leftarrow K_{t-1} - \alpha_t \mathcal{D}(K_{t-1})$

- If policy is **learned online** and needs to be applied to the process:
  - Compute the nearest policy $K_t^\star$ to K_t that stabilizes the local model

$$K_t^\star = \quad \arg\min_K \|K - K_t^s\|_2^2$$

s.t. K stabilizes local linear model

linear matrix inequality

- When policy is learned online, **exploration** is guaranteed by the reference r_t

SPECIAL CASE: OUTPUT TRACKING

- $x_t = [\, y_t,\ y_{t-1},\ \ldots,\ y_{t-n_o},\ u_{t-1},\ u_{t-2},\ \ldots,\ u_{t-n_i}\,]$

  $\Delta u_t = u_t - u_{t-1}$ control input increment

- Stage cost: $\| \, y_{t+1} - r_t \, \|^2_{Qy} + \| \, \Delta u_t \, \|^2_R + \| \, q_{t+1} \, \|^2_{Q_q}$

- Integral action dynamics $q_{t+1} = q_t + (y_{t+1} - r_t)$

➡ $$s_t = \begin{bmatrix} x_t \\ q_t \end{bmatrix}, \quad p_t = r_t.$$

- **Linear policy parametrization**:

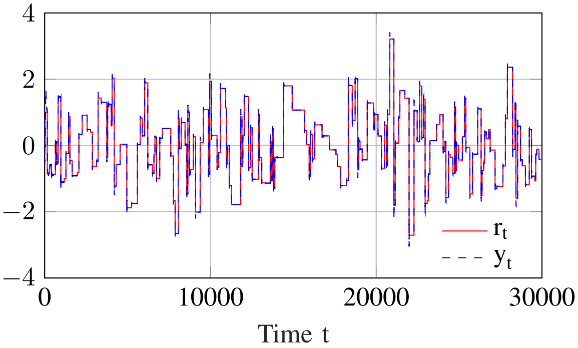
$$\pi_K(s_t,\, r_t) = -K^s \cdot s_t - K^r \cdot r_t, \qquad K = \begin{bmatrix} K^s \\ K^r \end{bmatrix}$$

EXAMPLE: RETRIEVE LQR FROM DATA

$$\begin{cases} x_{t+1} = \begin{bmatrix} -0.669 & 0.378 & 0.233 \\ -0.288 & -0.147 & -0.638 \\ -0.337 & 0.589 & 0.043 \end{bmatrix} x_t + \begin{bmatrix} -0.295 \\ -0.325 \\ -0.258 \end{bmatrix} u_t \\ y_t = \begin{bmatrix} -1.139 & 0.319 & -0.571 \end{bmatrix} x_t \end{cases}$$

model is unknown

Online tracking performance (no disturbance, $d_t = 0$):



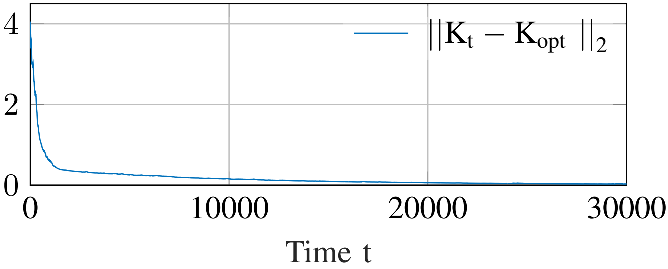
$Q_y = 1$
$R = 0.1$
$Q_q = 1$

n_i	n_o	L
3	3	20

N_0	N_r	N_q
50	1	10

EXAMPLE: RETRIEVE LQR FROM DATA

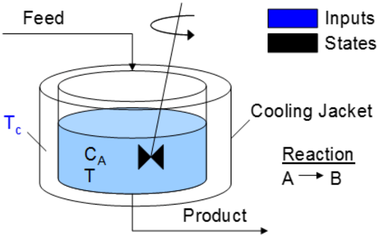
Evolution of the error $\|K_t - K_{opt}\|_2$:



$$K_{\rm SGD} = [-1.255, 0.218, 0.652, 0.895, 0.050, 1.115, -2.186]$$

$$K_{\rm opt} = [-1.257, 0.219, 0.653, 0.898, 0.050, 1.141, -2.196]$$

NONLINEAR EXAMPLE



Continuously Stirred Tank Reactor (CSTR)

apmonitor.com

model is unknown

Feed:

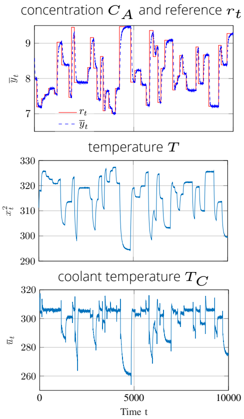
- concentration: 10kg mol/m^3
- temperature: 298.15K

$$T = \hat{T} + \eta_T,\ C_A = \hat{C_A} + \eta_C, \quad \eta_T, \eta_C \sim \mathcal{N}(0, \sigma^2), \quad \sigma = 0.01$$

$$\boxed{Q_y = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \qquad R = 0.1 \qquad Q_q = \begin{bmatrix} 0.01 & 0 \\ 0 & 0 \end{bmatrix}}$$

NONLINEAR EXAMPLE

Online learning

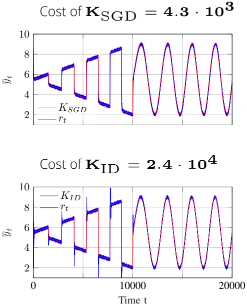

concentration C_A and reference r_t

temperature T

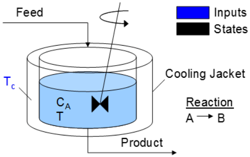
coolant temperature T_C

n_i	n_o	L
2	3	10
N_0	N_r	N_q
50	20	20

Validation phase


Cost of $\mathbf{K}_{\mathrm{SGD}} = \mathbf{4.3 \cdot 10^3}$

Cost of $\mathbf{K}_{\mathrm{ID}} = \mathbf{2.4 \cdot 10^4}$



Continuously Stirred Tank Reactor (CSTR)

(courtesy: apmonitor.com)

SGD beats SYS-ID + LQR

- Extended to **switching-linear** and **nonlinear** policy, and to **collaborative learning**

(Ferrarotti, Bemporad, 2020a) (Ferrarotti, Bemporad, 2020b) (Ferrarotti, Breschi, Bemporad, 2021)

LEARNING OPTIMAL MPC CALIBRATION

MPC CALIBRATION PROBLEM

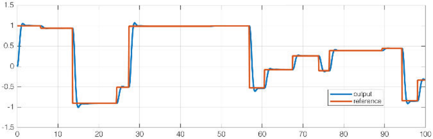
- The design depends on a vector x of **MPC parameters**

- Parameters can be many things:
  - MPC weights, prediction model coefficients, horizons
  - Covariance matrices used in Kalman filters
  - Tolerances used in numerical solvers
  - ...

- Define a **performance index** f over a closed-loop simulation or real experiment. For example:

$$f(x) = \sum_{t=0}^{T} \|y(t) - r(t)\|^2$$

(tracking quality)



- **Automatic calibration** = find the **best** combination of parameters by solving the **global optimization problem**

$$\min_x f(x)$$

GLOBAL OPTIMIZATION ALGORITHMS FOR AUTO-TUNING

What is a good optimization algorithm to solve $\min f(x)$ **?**

- The algorithm should not require the gradient $\nabla f(x)$ of $f(x)$, in particular if experiments are involved (**derivative-free** or **black-box optimization**)

- The algorithm should not get stuck on local minima (**global optimization**)

- The algorithm should make the **fewest evaluations** of the cost function f (which is expensive to evaluate)

AUTO-TUNING - GLOBAL OPTIMIZATION ALGORITHMS

- Several derivative-free global optimization algorithms exist: (Rios, Sahidinis, 2013)
  - Lipschitzian-based partitioning techniques:
    - **DIRECT** (DIvide in RECTangles) (Jones, 2001)
    - **SHGO** (Simplicial Homology Global Optimisation) (Endres, Sandrock, Focke, 2018)
    - Multilevel Coordinate Search (**MCS**) (Huyer, Neumaier, 1999)
  - Response surface methods
    - **Kriging** (Matheron, 1967), **DACE** (Sacks et al., 1989)
    - Efficient Global Optimization (**EGO**) (Jones, Schonlau, Welch, 1998)
    - Bayesian Optimization (**BO**) (Brochu, Cora, De Freitas, 2010)
  - Genetic Algorithms (**GA**) (Holland, 1975)
  - Particle Swarm Optimization (**PSO**) (Kennedy, 2010)
  - ...
- **GLIS** method - radial basis function surrogates + inverse distance weighting

(Bemporad, 2020)

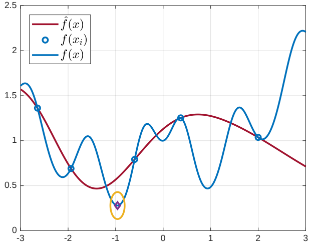
```
cse.lab.imtlucca.it/~bemporad/glis
```

```
pip install glis
```

AUTO-TUNING - GLIS

- **Goal**: solve the **global optimization** problem

$$\begin{array}{rl} \min_x & f(x) \\ \text{s.t.} & \ell \leq x \leq u \\ & g(x) \leq 0 \end{array}$$



- **Step #0**: Get random initial samples $x_1, \ldots, x_{N_{\rm init}}$ **(Latin Hypercube Sampling)**

- **Step #1**: given N samples of f at $x_1, \ldots, x_N$, build the **surrogate function**

$$\hat{f}(x) = \sum_{i=1}^{N} \beta_i \phi(\epsilon \|x - x_i\|_2)$$

$\phi =$ radial basis function

Example: $\phi(\epsilon d) = \frac{1}{1+(\epsilon d)^2}$

(inverse quadratic)

Vector β solves $\hat{f}(x_i) = f(x_i)$ for all $i = 1, \ldots, N$ (=linear system)

- **Note**: build and minimize $\hat{f}(x_i)$ iteratively may easily miss global optimum!

AUTO-TUNING - GLIS

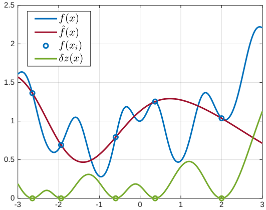
- **Step #2**: construct the **IDW exploration function**

$$z(x) = \frac{2}{\pi}\Delta F \tan^{-1}\left(\frac{1}{\sum_{i=1}^{N} w_i(x)}\right)$$

or 0 if $x \in \{x_1, \ldots, x_N\}$

where $w_i(x) = \dfrac{e^{-\|x-x_i\|^2}}{\|x-x_i\|^2}$

ΔF = observed range of $f(x_i)$



- **Step #3**: optimize the **acquisition function**

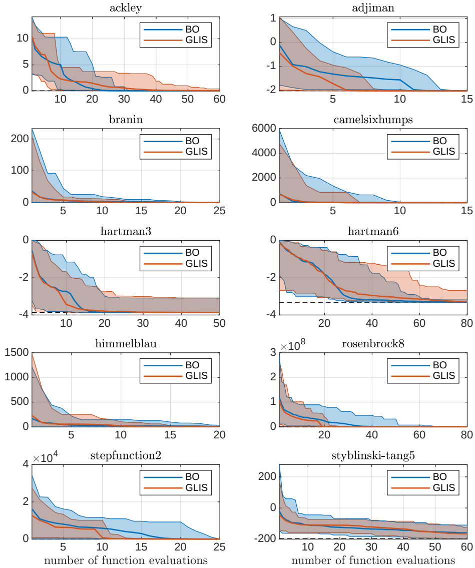
$$\begin{array}{rll} x_{N+1} = & \arg\min & \hat{f}(x) - \delta z(x) \\ & \text{s.t.} & \ell \le x \le u,\ g(x) \le 0 \end{array}$$

δ= *exploitation* vs *exploration* tradeoff

to get new sample x_{N+1}

- Iterate the procedure to get new samples $x_{N+2}, \ldots, x_{N_{\max}}$

GLIS VS BAYESIAN OPTIMIZATION



problem	n	BO [s]	GLIS [s]
ackley	2	29.39	3.13
adjiman	2	3.29	0.68
branin	2	9.66	1.17
camelsixhumps	2	4.82	0.62
hartman3	3	26.27	3.35
hartman6	6	54.37	8.80
himmelblau	2	7.40	0.90
rosenbrock8	8	63.09	13.73
stepfunction2	4	11.72	1.81
styblinski-tang5	5	37.02	6.10

Results computed on 20 runs per test

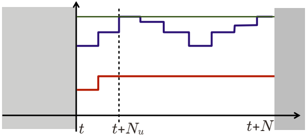
BO = MATLAB's **bayesopt** fcn

- Comparable performance
- GLIS is computationally lighter
- GLIS is more flexible

AUTO-TUNING: MPC EXAMPLE

- We want to auto-tune the linear MPC controller

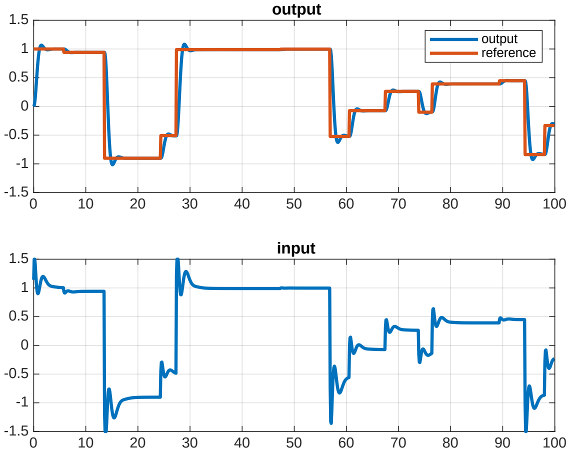
$$
\begin{aligned}
\min \quad & \sum_{k=0}^{50-1} (y_{k+1} - r(t))^2 + ({\color{red}W^{\Delta u}}(u_k - u_{k-1}))^2 \\
\text{s.t.} \quad & x_{k+1} = Ax_k + Bu_k \\
& y_c = Cx_k \\
& -1.5 \leq u_k \leq 1.5 \\
& u_k \equiv u_{N_u},\ \forall k = {\color{red}N_u}, \ldots, N-1
\end{aligned}
$$

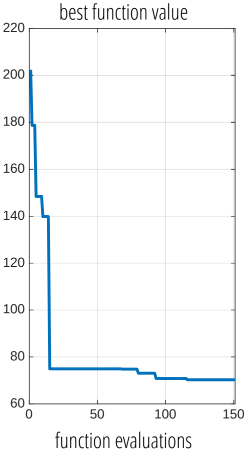


- Calibration parameters: ${\color{red}x = [\log_{10} W^{\Delta u}, N_u]}$
- Range: $-5 \leq x_1 \leq 3$ and $1 \leq x_2 \leq 50$
- Closed-loop performance objective:

$$
f(x) = \sum_{t=0}^{T} \underbrace{(y(t) - r(t))^2}_{\text{track well}} + \underbrace{\frac{1}{2}(u(t) - u(t-1))^2}_{\text{smooth control action}} + \underbrace{2N_u}_{\text{small QP}}
$$

AUTO-TUNING: EXAMPLE



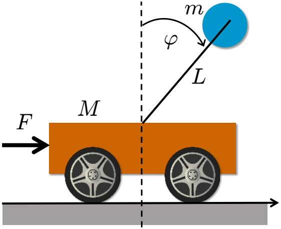


- Result: $x^\star = [-0.2341, 2.3007]$ → $W^{\Delta u} = 0.5833, N_u = 2$

MPC AUTOTUNING EXAMPLE

(Forgione, Piga, Bemporad, 2020)

- Linear MPC applied to cart-pole system: **14 parameters** to tune



– **sample time**

– **weights** on outputs and input increments

– prediction and control **horizons**

– **covariance** matrices of Kalman filter

– absolute and relative **tolerances** of QP solver

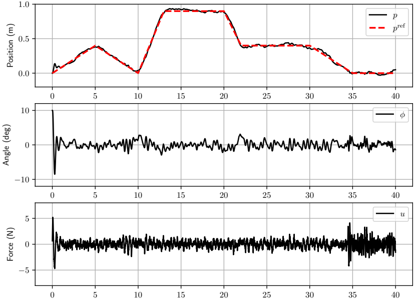
- Closed-loop performance score: $J = \displaystyle\int_0^T |p(t) - p_{\rm ref}(t)| + 30|\phi(t)|dt$

- MPC parameters tuned using 500 iterations of GLIS

- Performance tested with simulated cart on two hardware platforms (PC, Raspberry PI)

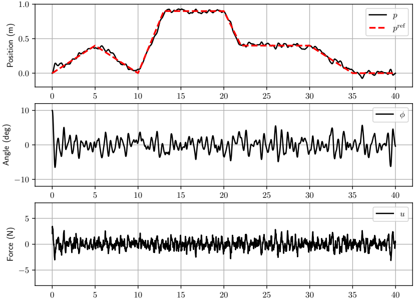
MPC AUTOTUNING EXAMPLE

MPC optimized for **desktop PC**



optimal sample time = **6 ms**

MPC optimized for **Raspberry PI**



optimal sample time = **22 ms**

- MPC parameters tuned by **GLIS** global optimizer (500 fcn evals)
- Auto-calibration can squeeze max performance out of the available hardware
- Bayesian optimization gives similar results, but with larger computation effort

AUTO-TUNING: PROS AND CONS

- Pros:
  - 👍 Selection of calibration parameters x to test is fully automatic
  - 👍 Applicable to any calibration parameter (weights, horizons, solver tolerances, ...)
  - 👍 Rather arbitrary performance index $f(x)$ (tracking performance, response time, worst-case number of flops, ...)
- Cons:
  - 👎 Need to **quantify** an objective function $f(x)$
  - 👎 No room for **qualitative** assessments of closed-loop performance
  - 👎 Often have **multiple objectives**, not clear how to blend them in a single one

ACTIVE PREFERENCE LEARNING

(Bemporad, Piga, *Machine Learning*, 2021)

- Objective function $f(x)$ is not available (**latent function**)

- We can only express a **preference** between two choices:

$$\pi(x_1,x_2) = \begin{cases} -1 & \text{if } x_1 \text{ “better” than } x_2 \qquad [f(x_1) < f(x_2)] \\ 0 & \text{if } x_1 \text{ “as good as” } x_2 \qquad [f(x_1) = f(x_2)] \\ 1 & \text{if } x_2 \text{ “better” than } x_1 \qquad [f(x_1) > f(x_2)] \end{cases}$$

- We want to find a global optimum $x^\star$ (=“better” than any other x)

$$\text{find } x^\star \text{ such that } \pi(x^\star,x) \leq 0,\ \forall x \in \mathcal{X},\ \ell \leq x \leq u$$

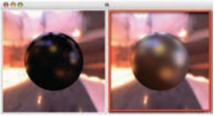
- **Active preference learning**: iteratively propose a new sample to compare

- **Key idea**: learn a **surrogate** of the (latent) objective function from preferences

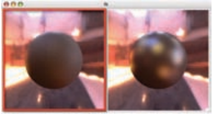
PREFERENCE-LEARNING EXAMPLE

(Brochu, de Freitas, Ghosh, 2007)

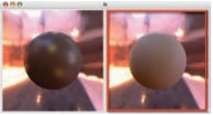
$Target$



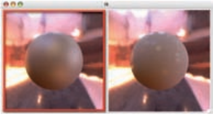
1



2



3



4

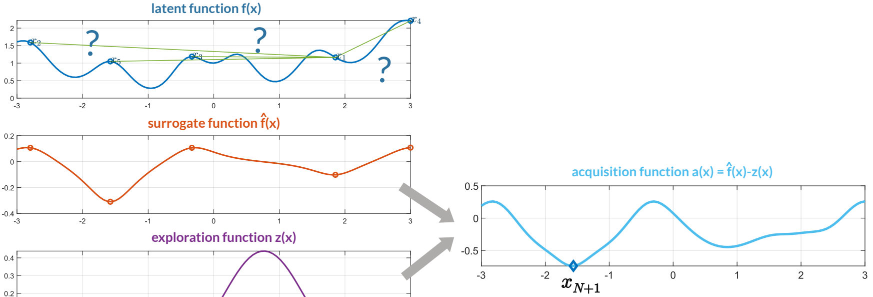
- Realistic image synthesis of material appearance are based on models with many parameters $x_1, \ldots, x_n$

- Defining an objective function $f(x)$ is hard, while a human can easily assess whether an image resembles the target one or not

- **Preference gallery** tool: at each iteration, the user compares two images generated with two different parameter instances

ACTIVE PREFERENCE LEARNING ALGORITHM

(Bemporad, Piga, *Machine Learning*, 2021)



latent function f(x)

surrogate function $\hat{f}(x)$

exploration function z(x)

acquisition function a(x) = $\hat{f}$(x)-z(x)

- **Fit a surrogate** $\hat{f}(x)$ that respects the **preferences** expressed by the decision maker at sampled points (by solving a QP)

- **Minimize an acquisition function** $\hat{f}(x) - \delta z(x)$ to get a **new sample** x_{N+1}

- **Compare** x_{N+1} to the current “best” point (👍, 👎, ≈) and **iterate**

SEMI-AUTOMATIC CALIBRATION BY PREF.-BASED LEARNING

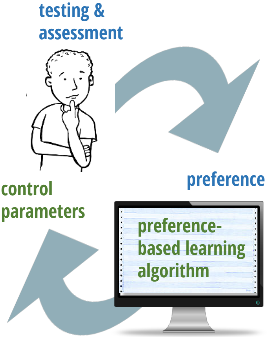
- Use **preference-based optimization** (**GLISp**) algorithm for **semi-automatic tuning** of MPC (Zhu, Bemporad, Piga, 2021)

- Latent function = calibrator’s (unconscious) score of closed-loop MPC performance

- GLISp **proposes a new combination** x_{N+1} of MPC parameters to test

- By observing test results, the calibrator expresses a **preference**, telling if x_{N+1} is “**better**”, “**similar**”, or “**worse**” than current best combination

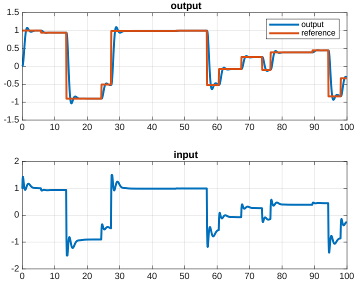
- Preference learning algorithm: **update the surrogate** $\hat{f}(x)$ of the latent function, optimize the acquisition function, **ask preference**, and **iterate**



PREFERENCE-BASED TUNING: MPC EXAMPLE

- Semi-automatic tuning of $x = [\log_{10} W^{\Delta u}, N_u]$ in linear MPC

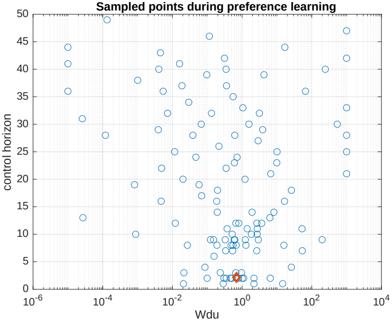
$$\begin{array}{ll}\min & \sum_{k=0}^{50-1} (y_{k+1} - r(t))^2 + (W^{\Delta u}(u_k - u_{k-1}))^2 \\ \text{s.t.} & x_{k+1} = Ax_k + Bu_k \\ & y_c = Cx_k \\ & -1.5 \leq u_k \leq 1.5 \\ & u_k \equiv u_{N_u}, \ \forall k = N_u, \ldots, N-1\end{array}$$



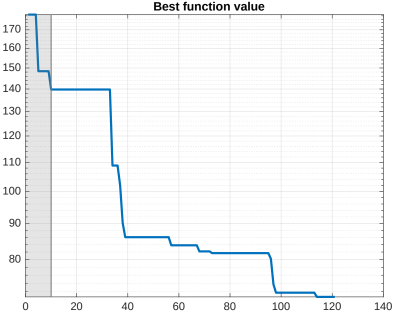
- Same performance index to assess closed-loop quality, but unknown: **only preferences** are available

- Result: $W^{\Delta u} = 0.6888$, $N_u = 2$

PREFERENCE-BASED TUNING: MPC EXAMPLE



tested combinations of MPC params



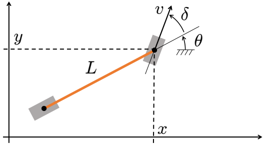
(latent) performance index

PREFERENCE-BASED TUNING: MPC EXAMPLE

(Zhu, Bemporad, Piga, 2021)

- Example: calibration of a simple MPC for lane-keeping (2 inputs, 3 outputs)

$$\begin{cases} \dot{x} &= v\cos(\theta+\delta) \\ \dot{y} &= v\sin(\theta+\delta) \\ \dot{\theta} &= \frac{1}{L}v\sin(\delta) \end{cases}$$



- Multiple control objectives:

"optimal obstacle avoidance", *"pleasant drive"*, *"CPU time small enough"*, …

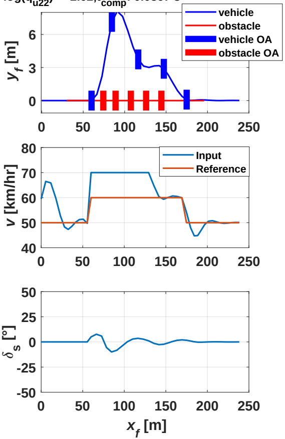
➡ **not easy to quantify in a single function**

- 5 MPC parameters to tune:
  - **sampling time**
  - prediction and control **horizons**
  - **weights** on input increments Δv, $\Delta\delta$

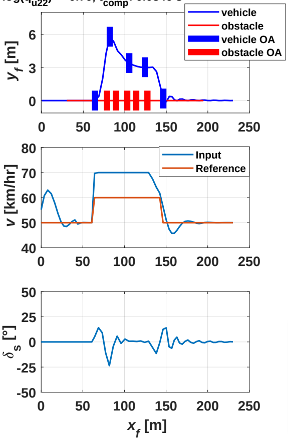
PREFERENCE-BASED TUNING: MPC EXAMPLE

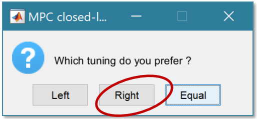
- Preference query window:

T_s = 0.332 s, N_u = 16, N_p = 17, log(q_{u11}) = 0.06, log(q_{u22}) = 2.02,t_{comp}: 0.0867 s



T_s = 0.243 s, N_u = 12, N_p = 17, log(q_{u11}) = 0.19, log(q_{u22}) = 0.70, t_{comp}: 0.0846 s



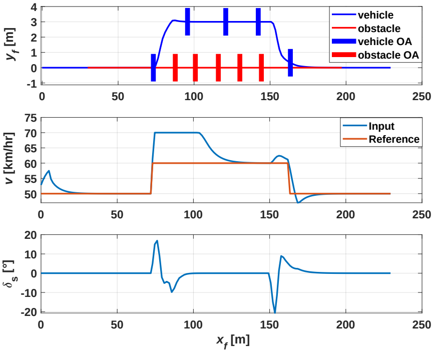


Which tuning do you prefer ?

Left [✓ Right] Equal

PREFERENCE-BASED TUNING: MPC EXAMPLE

- Convergence after 50 GLISp iterations (=49 queries):



Optimal MPC parameters:

- sample time = 85 ms (CPU time = 80.8 ms)
- prediction horizon = 16
- control horizon = 5
- weight on Δv = 1.82
- weight on $\Delta\delta$ = 8.28

- **Note**: no need to define a closed-loop performance index explicitly!

- Extended to handle also **unknown constraints** (Zhu, Piga, Bemporad, 2021)

WORST-CASE SCENARIO DETECTION

CORNER-CASE SCENARIO DETECTION PROBLEM

(Zhu, Bemporad, Kneissl, Esen, 2022)

- **Goal**: detect undesired simulation scenarios (=corner-cases)

- Let x = parameters defining the scenario, $\mathcal{X}_{\rm ODD}$ = operational design domain
  $x \in \mathcal{X}_{\rm ODD} \subseteq \mathbb{R}^n$

- critical scenario = vector x^* for which the closed-loop behavior is critical

- Example:
  - x = (initial distance between ego car and obstacle, obstacle acceleration, ...)
  - Critical scenario: time-to-collision is too short, excessive jerk of ego car, ...

- **Key idea**: use global optimizer GLIS to generate critical corner-cases

$$\begin{array}{rl} x^* \in \displaystyle\arg\min_{x\in\mathcal{X}_{\rm ODD}} & f(x) \\ \text{s.t.} & \ell \leq x \leq u \end{array}$$

$f(x)$ = criticality of closed-loop simulation (or experiment) determined by scenario x
(the smaller $f(x)$, the more critical x is)

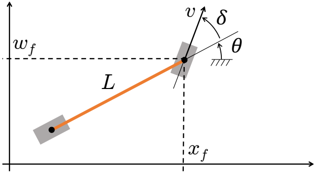
CORNER-CASE DETECTION: CASE STUDY

- **Problem**: find critical scenarios in automated driving w/ obstacles

- MPC controller for lane-keeping and obstacle-avoidance based on simple kinematic bicycle model (Zhu, Piga, Bemporad, 2021)

$$\begin{aligned}\dot{x}_f &= v\cos(\theta+\delta)\\ \dot{w}_f &= v\sin(\theta+\delta)\\ \dot{\theta} &= \frac{v\sin(\delta)}{L}\end{aligned}$$

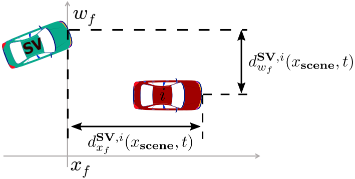
(x_f, w_f) = front-wheel position



- Black-box optimization problem: given k obstacles, solve

$$\min_{\ell\leq x\leq u} \sum_{i=1}^{k} d^{\mathrm{SV},i}_{x_f,\text{critical}}(x) + d^{\mathrm{SV},i}_{w_f,\text{critical}}(x)$$

s.t. other constraints



w_f
SV
i
$d^{\mathbf{SV},i}_{w_f}(x_{\mathbf{scene}}, t)$
$d^{\mathbf{SV},i}_{x_f}(x_{\mathbf{scene}}, t)$
x_f

CORNER-CASE DETECTION: CASE STUDY

- **Cost function terms** to minimize: for each obstacle $\#i$ define

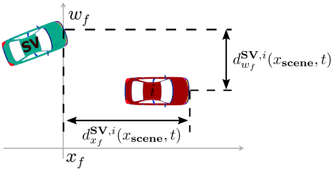
$$d^{\mathsf{SV},i}_{x_f,\text{critical}}(x) = \begin{cases} \min\limits_{t\in T_{\text{collision}}} d^{\mathsf{SV},i}_{x_f}(x,t) & \mathcal{I}^i_{\text{collision}} & \text{min dist. @collision with } \#i \\ L & \sim\mathcal{I}^i_{\text{collision}} \,\&\, \mathcal{I}_{\text{collision}} & \text{collision with other } \#j \neq \#i \\ \sum\limits_{t\in T_{\text{sim}}} d^{\mathsf{SV},i}_{x_f}(x,t) & \sim \mathcal{I}_{\text{collision}} & \text{no collision} \end{cases}$$

$$d^{\mathsf{SV},i}_{w_f,\text{critical}}(x) = \begin{cases} \min\limits_{t\in T_{\text{collision}}} d^{\mathsf{SV},i}_{w_f}(x,t) & \mathcal{I}^i_{\text{collision}} \\ w_{f,\text{safe}} & \sim\mathcal{I}^i_{\text{collision}} \,\&\, \mathcal{I}_{\text{collision}} \\ \sum\limits_{t\in T_{\text{sim}}} d^{\mathsf{SV},i}_{w_f}(x,t) & \sim \mathcal{I}_{\text{collision}} \end{cases}$$

$$\mathcal{I}^i_{\text{collision}} = \texttt{true} \quad \text{if} \quad \exists t \in T_{\text{sim}} \text{ s.t.}$$

$$(d^{\mathsf{SV},i}_{x_f}(x,t) \leq L) \,\&\, (d^{\mathsf{SV},i}_{w_f}(x,t) \leq W)$$

$$\mathcal{I}_{\text{collision}} = \texttt{true} \quad \text{if} \quad \exists h \quad \text{s.t.} \quad \mathcal{I}^h_{\text{collision}} = \texttt{true}$$

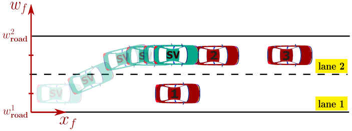

w_f
SV
i
$d^{\mathbf{SV},i}_{w_f}(x_{\mathbf{scene}},t)$
$d^{\mathbf{SV},i}_{x_f}(x_{\mathbf{scene}},t)$
x_f

CORNER-CASE DETECTION: CASE STUDY

- **Logical scenario 1**: GLIS identifies 64 collision cases within 100 simulations

iter	x					
	x_{f1}^0	v_1^0	x_{f2}^0	v_2^0	x_{f3}^0	v_3^0
51	15.00	30.00	44.14	10.00	49.10	47.39
79	28.09	30.00	70.29	10.00	74.79	31.74
40	34.30	30.00	60.59	10.00	77.80	35.97

red = optimal solution found by GLIS solver

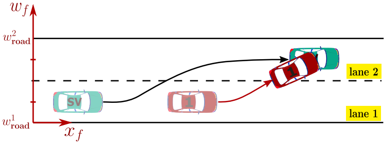


Ego car changes lane to avoid #1, but cannot brake fast enough to avoid #2

- **Logical scenario 2**: GLIS identifies 9 collision cases within 100 simulations

iter	x		
	x_{f1}^0	v_1^0	t_c
28	12.57	46.94	16.75
16	17.53	47.48	23.65
88	44.54	41.26	16.02

red = optimal solution found by GLIS solver

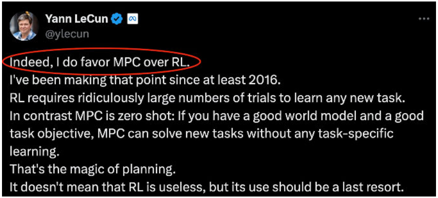


Ego car changes lane to avoid #1, but cannot decelerate in time for the sudden lane-change of #1

LEARNING-BASED MPC: FINAL REMARKS

LEARNING-BASED MPC: FINAL REMARKS

- ML very useful to get **control-oriented models** (and **control laws**) from **data**
- ML cannot replace control engineering:
  - Blindly applying **deep NNs** can lead to useless models for embedded control
  - Approximating MPC laws by NN's can fail, often still need **online optimization**
  - **Model-free reinforcement learning** can fail wrt **model-based** control design, which is more sample-efficient and better performs tasks it wasn't trained for



Yann LeCun
@ylecun

Indeed, I do favor MPC over RL.
I've been making that point since at least 2016.
RL requires ridiculously large numbers of trials to learn any new task.
In contrast MPC is zero shot: If you have a good world model and a good task objective, MPC can solve new tasks without any task-specific learning.
That's the magic of planning.
It doesn't mean that RL is useless, but its use should be a last resort.

(Yann LeCun, Twitter/X, August 25, 2024)