

MODEL PREDICTIVE CONTROL

LEARNING-BASED MPC

Alberto Bemporad

`imt.lu/ab`

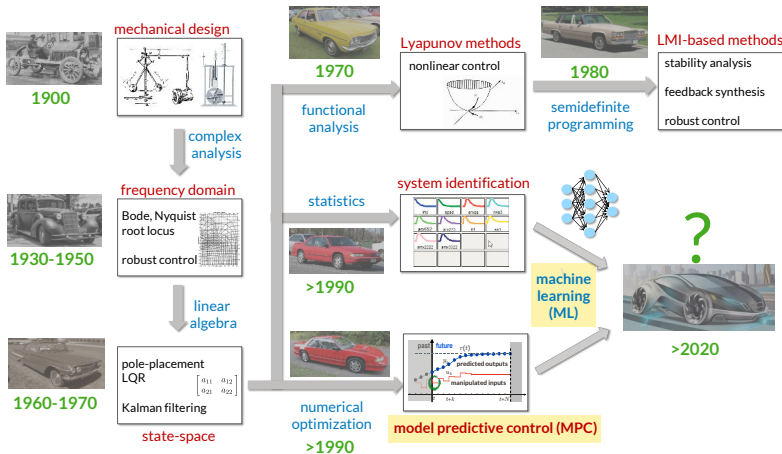
COURSE STRUCTURE

- ✓ 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)

Course page:

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

MACHINE LEARNING AND CONTROL ENGINEERING



MPC AND ML

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

model predictive control



machine learning



nonlinear control



system identification



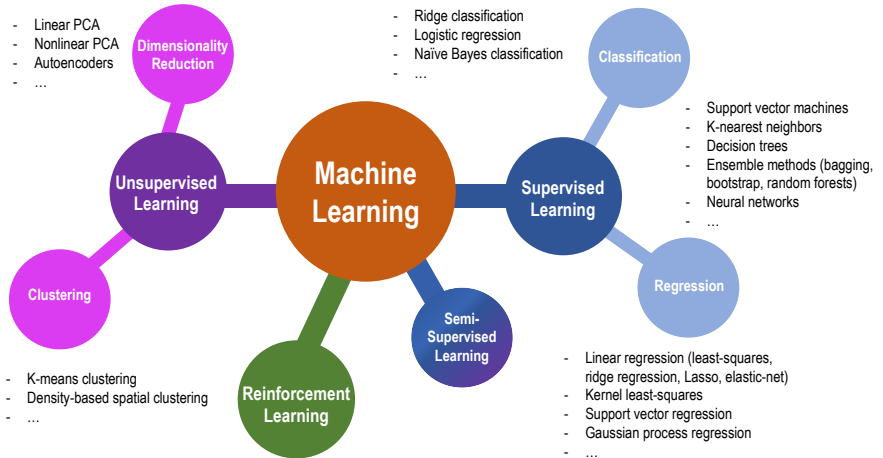
PID control





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

MACHINE LEARNING (ML)

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



MACHINE LEARNING (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

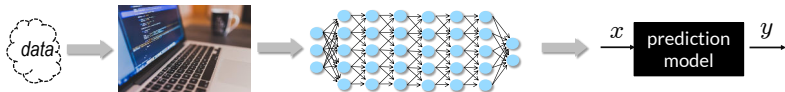
MPC DESIGN FROM DATA

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

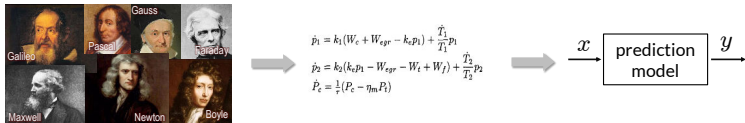
LEARNING PREDICTION MODELS FOR MPC

CONTROL-ORIENTED NONLINEAR MODELS

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



- **Physics-based** models: use physical principles to create a prediction model (e.g.: weather forecast, chemical reaction, mechanical laws, ...)



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

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

(George E. P. Box)

MODELS FOR CONTROL SYSTEMS DESIGN

- Prediction models for **model predictive control**:

- Complex model = complex controller
→ model must be as **simple** as possible
- Easy to **linearize** (to get Jacobian matrices for nonlinear optimization)

- Prediction models for **state estimation**:

- Complex model = complex Kalman filter
- Easy to linearize

- Models for **virtual sensing**:

- No need to use simple models
(except for computational reasons)

- Models for **diagnostics**:

- Usually a **classification problem** to solve
- Complexity is also less of an issue

Linear models

- linear I/O models (ARX, ARMAX,...)
- subspace linear SYS-ID
- linear regression
(ridge, elastic-net, Lasso)

Piecewise linear models

- decision-trees
- neural nets + (leaky)ReLU
- K-means + linear models

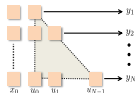
Nonlinear linear models

- basis functions + linear regression
- neural networks
- ~~K-nearest neighbors~~
- ~~support vector machines~~
- ~~kernel methods~~
- ~~random forests~~

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 $y_t \approx \mathcal{N}(y_{t-1}, \dots, y_{t-n_a}, u_{t-1}, \dots, u_{t-n_b})$
- Neural state-space** models:
 - w/ state data**: fit a neural network model $x_{t+1} \approx \mathcal{N}_x(x_t, u_t)$, $y_t \approx \mathcal{N}_y(x_t)$
 - 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)

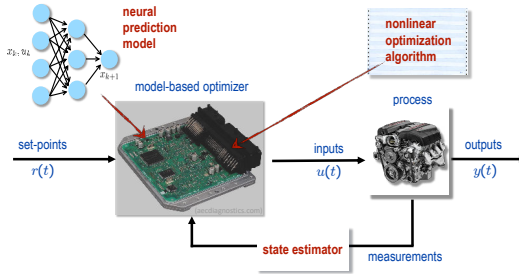
$$y_{t+k} = h_k(x_t, u_t, \dots, u_{t+k-1}), k = 1, \dots, N$$



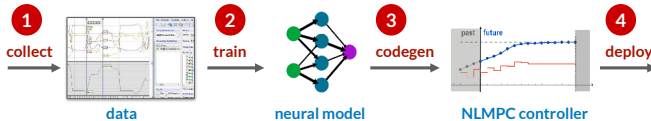
- Recurrent neural networks** are more appropriate for accurate open-loop predictions, but more difficult to train (see later ...)

NLMPC BASED ON NEURAL NETWORKS

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

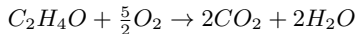
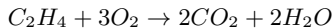
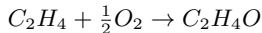


- MPC design workflow:



MPC OF ETHYLENE OXIDATION PLANT

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



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

$$\begin{cases} \dot{x}_1 &= u_1(1 - x_1x_4) \\ \dot{x}_2 &= u_1(u_2 - x_2x_4) - A_1e^{\frac{\gamma_1}{x_4}}(x_2x_4)^{\frac{1}{2}} - A_2e^{\frac{\gamma_2}{x_4}}(x_2x_4)^{\frac{1}{4}} \\ \dot{x}_3 &= -u_1x_3x_4 + A_1e^{\frac{\gamma_1}{x_4}}(x_2x_4)^{\frac{1}{2}} - A_3e^{\frac{\gamma_3}{x_4}}(x_3x_4)^{\frac{1}{2}} \\ \dot{x}_4 &= \frac{u_1(1-x_4) + B_1e^{\frac{\gamma_1}{x_4}}(x_2x_4)^{\frac{1}{2}} + B_2e^{\frac{\gamma_2}{x_4}}(x_2x_4)^{\frac{1}{4}}}{x_1} \\ &\quad + \frac{B_3e^{\frac{\gamma_3}{x_4}}(x_3x_4)^{\frac{1}{2}} - B_4(x_4 - T_C)}{x_1} \\ y &= x_3 \end{cases}$$

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

NEURAL NETWORK MODEL OF ETHYLENE OXIDATION PLANT

- Train **state-space neural-network** model

$$x_{k+1} = \mathcal{N}(x_k, u_k)$$

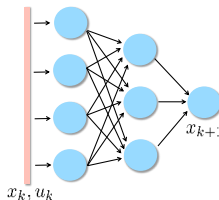
1,000 training samples $\{u_k, x_k\}$

2 layers (6 neurons, 6 neurons)

6 inputs, 4 outputs

sigmoidal activation function

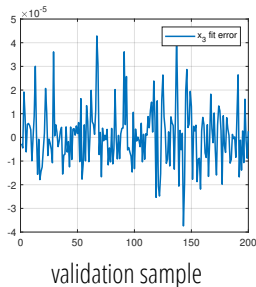
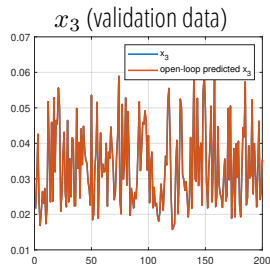
→ **112 coefficients**



- NN model trained by **ODYS Deep Learning** toolset
(model fitting + Jacobians → neural model in C)

- Model validated on 200 samples.

$x_{3,k+1}$ reproduced from x_k, u_k with max 0.4% error



MPC OF ETHYLENE OXIDATION PLANT

- **MPC** settings:

sampling time $T_s = 5 \text{ s}$ measured disturbance @t=200

prediction horizon $N = 10$

control horizon $N_u = 3$

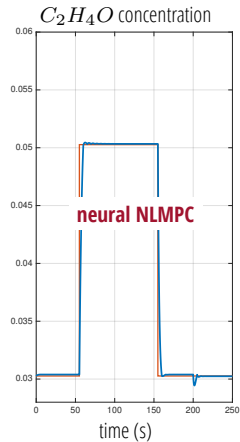
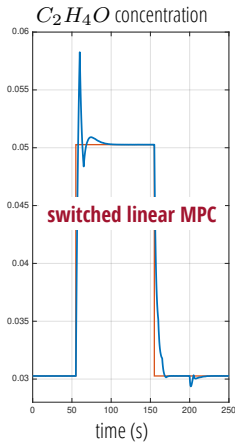
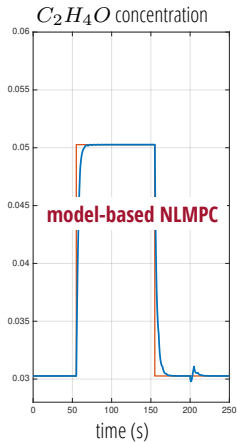
constraints $0.0704 \leq u_1 \leq 0.7042$

cost function $\sum_{k=0}^{N-1} (y_{k+1} - r_{k+1})^2 + \frac{1}{100} (u_{1,k} - u_{1,k-1})^2$

- We compare 3 different configurations:

- NLMPC based on **physical model**
- Switched linear MPC based on **3 linear models** obtained by linearizing the nonlinear model at $C_2H_4O = \{0.03, 0.04, 0.05\}$
- NLMPC based on black-box **neural network** model

MPC OF ETHYLENE OXIDATION PLANT - CLOSED-LOOP RESULTS

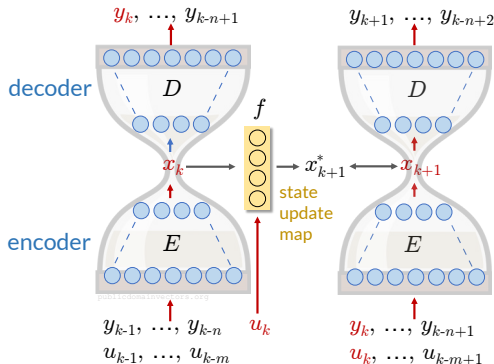
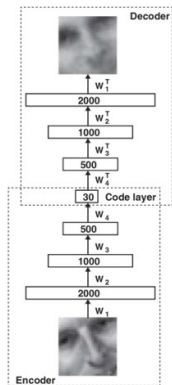


- Neural and model-based NLMPC have **similar** closed-loop performance
- Neural NLMPC requires **no physical model**

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



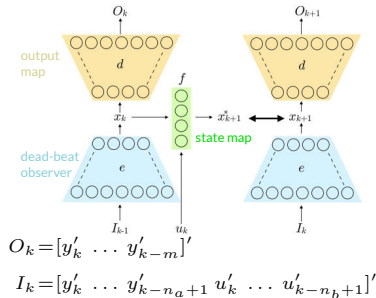
ANN with hourglass structure

(Hinton, Salakhutdinov, 2006)

LEARNING NONLINEAR STATE-SPACE MODELS FOR MPC

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

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



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

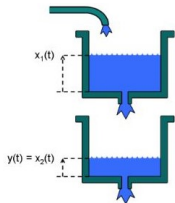
- Quasi-LPV** structure for MPC: set $f(x_k, u_k) = A(x_k, u_k) \begin{bmatrix} x_k \\ 1 \end{bmatrix} + B(x_k, u_k) u_k$
 $(A_{ij}, B_{ij}, C_{ij} = \text{feedforward NNs})$ $y_k = C(x_k, u_k) \begin{bmatrix} x_k \\ 1 \end{bmatrix}$

- 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

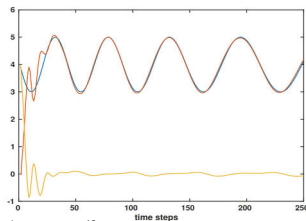


www.mathworks.com

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

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



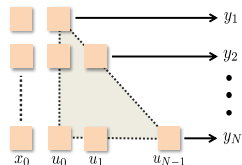
LTV-MPC results

LEARNING AFFINE NEURAL PREDICTORS FOR MPC

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

- Alternative: **learn the entire prediction**

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



- LTV-MPC formulation:** linearize h_k around nominal inputs \bar{u}_j

$$y_k = h_k(x_0, \bar{u}_0, \dots, \bar{u}_{k-1}) + \sum_{j=0}^{k-1} \frac{\partial h_k}{\partial u_j}(x_0, \bar{u}_0, \dots, \bar{u}_{k-1})(u_j - \bar{u}_j)$$

Example: \bar{u}_k = MPC sequence optimized @ $k - 1$

- Avoid computing Jacobians by fitting h_k in the affine form

$$y_k = f_k(x_0, \bar{u}_0, \dots, \bar{u}_{k-1}) + g_k(x_0, \bar{u}_0, \dots, \bar{u}_{k-1}) \begin{bmatrix} u_0 - \bar{u}_0 \\ \vdots \\ u_{k-1} - \bar{u}_{k-1} \end{bmatrix}$$

cf. (Liu, Kadiramanathan, 1998)

LEARNING AFFINE NEURAL PREDICTORS FOR MPC

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

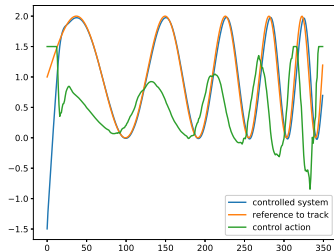
10000 training samples, ANN with **2** layers of **20 ReLU neurons**

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

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

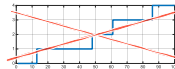


ON THE USE OF NEURAL NETWORKS FOR MPC

- 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**



0.4075	0.7306	0.2140	0.5894	0.0601
1.2234	0.5810	0.1670	0.6116	0.5191
0.9559	0.5983	0.6461	0.5833	0.14
0.116	0.5032	0.1803	0.4570	0.59
0.06	0.5316	0.1165	0.8826	0.1
0.1	0.6063	0.3653	0.4936	0.

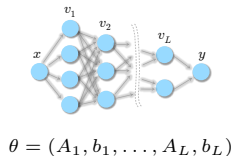


LEARNING NEURAL NETWORK MODELS FOR CONTROL

TRAINING FEEDFORWARD NEURAL NETWORKS

- **Feedforward neural network** model:

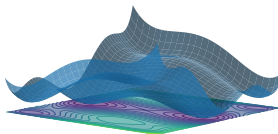
$$y_k = f_y(x_k, \theta) = \begin{cases} v_{1k} &= A_1 x_k + b_1 \\ v_{2k} &= A_2 f_1(v_{1k}) + b_2 \\ \vdots & \vdots \\ v_{Lk} &= A_{L_y} f_{L-1}(v_{(L-1)k}) + b_L \\ \hat{y}_k &= f_L(v_{Lk}) \end{cases}$$



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

- **Training problem:** given a dataset $\{x_0, y_0, \dots, x_{N-1}, y_{N-1}\}$ 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 RECURRENT NN'S VIA EKF

TRAINING FEEDFORWARD NEURAL NETWORKS BY 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 $\{x_k, y_k\}$

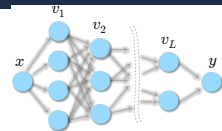
- Ratio $\text{Var}[\eta_k] / \text{Var}[\zeta_k]$ is related to the **learning-rate**
- Initial matrix $(P_{0|-1})^{-1}$ is related to **quadratic regularization** on θ

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, ...)



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

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

- **Training problem:** given a dataset $\{u_0, y_0, \dots, u_{N-1}, y_{N-1}\}$ solve

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

- **Main issue:** x_k are **hidden states**, i.e., are **unknowns** of the problem

TRAINING RNNS VIA EXTENDED KALMAN FILTERING

TRAINING RNNs BY EKF

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

- Estimate both hidden states x_k and parameters θ_x, θ_y by **EKF** based on model

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

Ratio $\text{Var}[\eta_k] / \text{Var}[\zeta_k]$ related to **learning-rate** of training algorithm

Inverse of initial matrix P_0 related to **ℓ_2 -penalty** on θ_x, θ_y

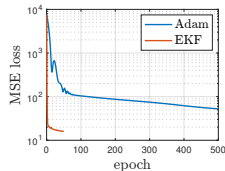
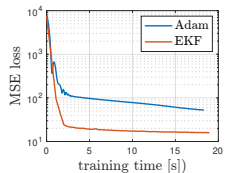
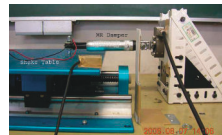
- RNN and its hidden state x_k can be estimated **on line** from a streaming dataset $\{u_k, y_k\}$, and/or **offline** by processing multiple epochs of a given dataset
- Can handle **general smooth strongly convex** loss fncs/regularization terms
- Can add **ℓ_1 -penalty** $\lambda \left\| \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} \right\|_1$ to **sparsify** θ_x, θ_y by changing EKF update into

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

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 (Macbook Pro 14" M1 Max)

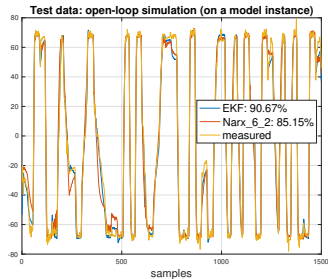


TRAINING RNNs BY EKF - EXAMPLES

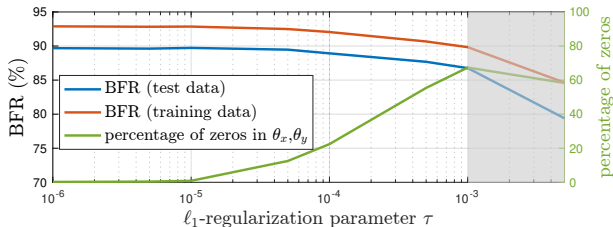
- Compare BFR¹ wrt NNARX model (SYS-ID TBX):

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 $\tau \left\| \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} \right\|_1$



¹Best fit rate $BFR = 100(1 - \frac{\|Y - \hat{Y}\|_2}{\|Y - \bar{y}\|_2})$, averaged over 20 runs from different initial weights

TRAINING LSTMS BY EKF - EXAMPLES

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

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

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

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

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

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

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

- $N=1000$ data used for training, 1000 for testing the model
- Train **linear state-space model** with 3 states and **sigmoidal output** function

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

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

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

σ	EKF accuracy [%]	
	test	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 RNNS VIA SEQUENTIAL LEAST SQUARES

TRAINING RNNs BY SEQUENTIAL LEAST-SQUARES

(Bemporad, 2021 - <http://arxiv.org/abs/2112.15348>)

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

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

- θ_x, θ_y, x_0 = manipulated variables, \hat{y}_k = output, y_k = reference, u_k = meas. dist.
- $r(x_0, \theta_x, \theta_y)$ = input penalty, $\ell(y_k, \hat{y}_k)$ = output penalty
- N = prediction horizon, control horizon = 1

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

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


TRAINING RNNs BY SEQUENTIAL LEAST-SQUARES

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

$$M_{0x} = I, \quad M_{0\theta_x} = 0$$

$$M_{(k+1)x} = \nabla_x f_x(x_k^h, u_k, \theta_x^h) M_{kx}$$

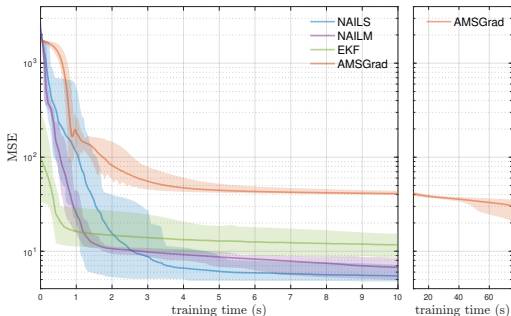
$$M_{(k+1)\theta_x} = \nabla_x f_x(x_k^h, u_k, \theta_x^h) M_{k\theta_x} + \nabla_{\theta_x} f_x(x_k^h, u_k, \theta_x^h)$$

- Take 2nd-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)

TRAINING RNNs BY SEQUENTIAL LS AND ADMM

(Bemporad, 2021 - <http://arxiv.org/abs/2112.15348>)

- Fluid-damper example: (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

NAILS = GNN method with line search

NAILM = GNN method with LM steps

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

TRAINING RNNs BY SEQUENTIAL LS AND ADMM

(Bemporad, 2021 - <http://arxiv.org/abs/2112.15348>)

- We also want to handle **non-smooth** (and **non-convex**) regularization terms

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

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

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

TRAINING RNNs BY SEQUENTIAL LS AND ADMM

(Bemporad, 2021 - <http://arxiv.org/abs/2112.15348>)

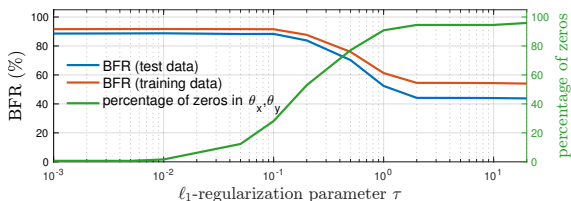
- ADMM + Seq. LS = **NAILS** algorithm (Nonconvex ADMM Iterations and Sequential LS)

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

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

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

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



$$\tau_x = \tau_y = \tau$$

(mean results over 20 runs
from different initial weights)

TRAINING RNNs BY SEQUENTIAL LS AND ADMM

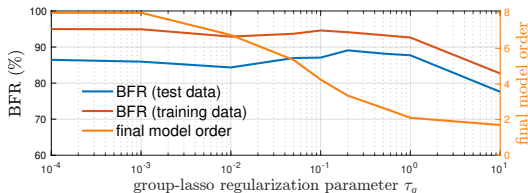
(Bemporad, 2021 - <http://arxiv.org/abs/2112.15348>)

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

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
EKF	89.27 (1.48)	86.67 (2.71)	47.9 (9.1)	13.2 s	50
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

\approx same fit than
SGD/EKF but sparser
models and faster
(CPU: 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 and columns and **reduce state-dimension** automatically



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

TRAINING RNNs BY SEQUENTIAL LS AND ADMM

(Bemporad, 2021 - <http://arxiv.org/abs/2112.15348>)

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

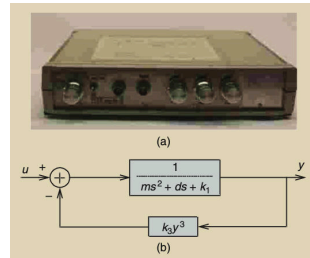
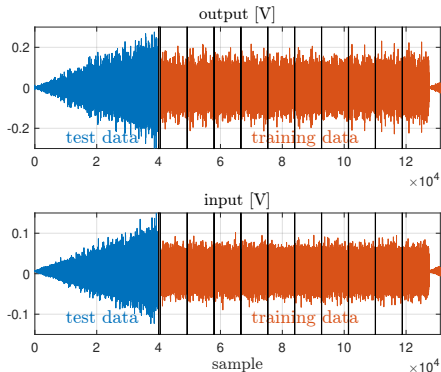
$$g(\theta_i) = \begin{cases} 0 & \text{if } \theta_i \in \mathcal{Q} \\ +\infty & \text{otherwise} \end{cases} \quad \mathcal{Q} = \text{multiples of 0.1 between -0.5 and 0.5}$$

- 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: ≈ 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 of ≈ 8600 data used for training, 40000 for testing



(Schoukens, Ljung, 2019)

Data download: <http://www.nonlinearbenchmark.org>

TRAINING RNNs - SILVERBOX BENCHMARK

(Bemporad, 2021 - <http://arxiv.org/abs/2112.15348>)

- **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

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

- 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 (1 epoch = 77505 training samples)

TRAINING RNNs - SILVERBOX BENCHMARK

(Bemporad, 2021 - <http://arxiv.org/abs/2112.15348>)

- Identification results on test data ²:

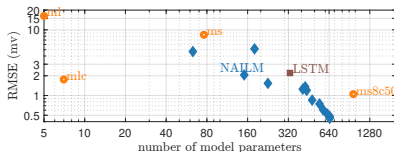
identification method	RMSE [mV]	BFR [%]
ARX (ml) [1]	16.29 [4.40]	69.22 [73.79]
NLARX (ms) [1]	8.42 [4.20]	83.67 [92.06]
NLARX (mlc) [1]	1.75 [1.70]	96.67 [96.79]
NLARX (ms8c50) [1]	1.05 [0.30]	98.01 [99.43]
Recurrent LSTM model [2]	2.20	95.83
SS encoder [3] ($n_x = 4$)	[1.40]	[97.35]
NAILM	0.35	99.33

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

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

[3] Beintema, Toth, Schoukens, 2021

- NAILM training time ≈ 400 s (MATLAB+CasADi on Apple M1 Max CPU)
- Repeat training with ℓ_1 -regularization:



²Trained RNN: <http://cse.lab.imtlucca.it/~bemporad/shared/silverbox/rnn888.zip>

- Computation time (Intel Core i9-10885H CPU @2.40GHz):

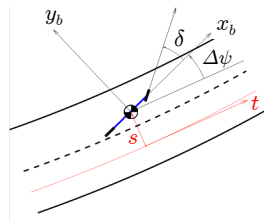
language	autodiff	EKF /time step CPU time	seq. LS /epoch CPU time
Python 3.8.1	PyTorch	≈ 30 ms	(N/A)
Python 3.8.1	JAX	≈ 9 ms	≈ 1.0 s
Julia 1.7.1	Flux.jl	≈ 2 ms	≈ 0.8 s
MATLAB R2021a	CasADi	≈ 0.5 ms	≈ 0.1 s

- Several **sparsity patterns** can be exploited in EKF updates (supported by **ODYS EKF** and **ODYS Deep Learning** libraries)
- **Note:** Extension to **gray-box** identification + state-estimation is immediate
- **Note:** RNN training by EKF can be used to generalize **output disturbance models** for offset-free set-point tracking to nonlinear I/O disturbance models

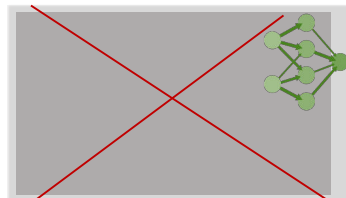
DEEP NONLINEAR MPC FOR AUTONOMOUS DRIVING

- **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
 - **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)

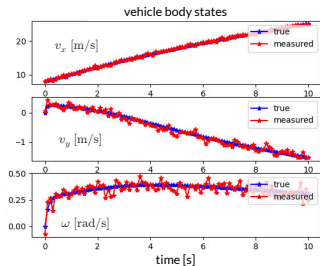


$$\begin{aligned}\dot{s} &= \frac{v_x \cos \Delta\psi - v_y \sin \Delta\psi}{1 - \kappa e_y} \\ \dot{e}_y &= v_x \sin \Delta\psi + v_y \cos \Delta\psi \\ \dot{\Delta\psi} &= \omega - \kappa \dot{s}\end{aligned}$$



DEEP NONLINEAR MPC FOR AUTONOMOUS DRIVING

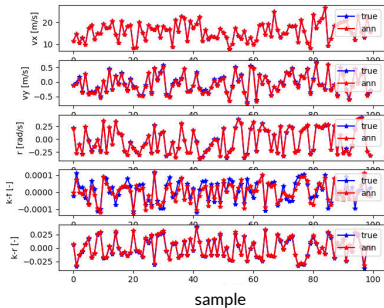
- **ODYS Deep Learning Toolset** used to learn a neural-network with input $(v_x, v_y, \omega, k_f, k_r, T_w, \delta) @k$ and output $(v_x, v_y, \omega, k_f, k_r) @k + 1$
- Data generated from high-fidelity simulation model with noisy measurements, sampled @10Hz
- Neural network model: **2 hidden layers, 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)



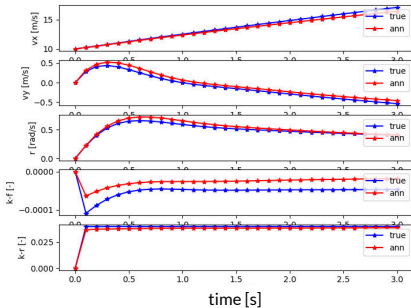
DEEP NONLINEAR MPC FOR AUTONOMOUS DRIVING

- 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



automatic
C-code gen

ODYS-NN training

ODYS



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

Float W[3685] = {0.838089282148443}
Float B[115] = {0.8728737412844339}

void neuralnetwork(Float xx, UInt8
{ /* Compute neural network out
```



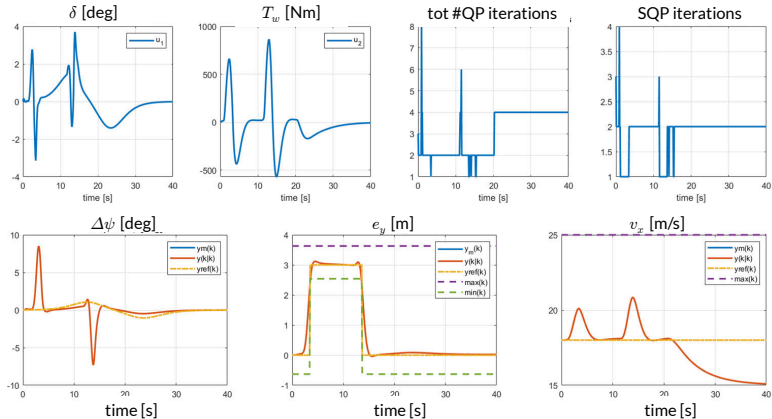
Embedded MPC



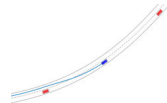
ODYS

DEEP NONLINEAR MPC FOR AUTONOMOUS DRIVING

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

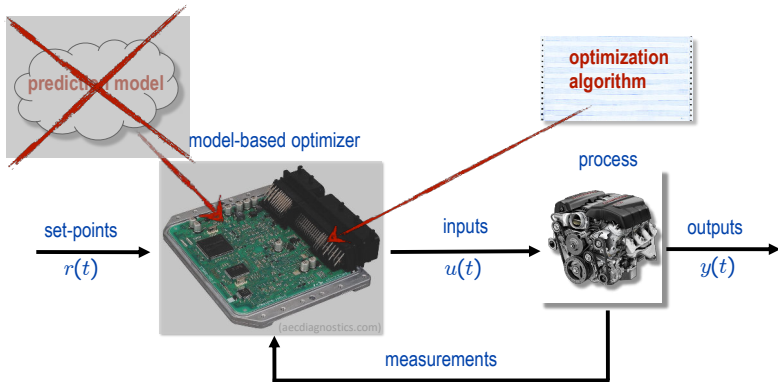


- Good reference tracking, constraints on e_y, v_x satisfied, smooth command action



DIRECT DATA-DRIVEN MPC

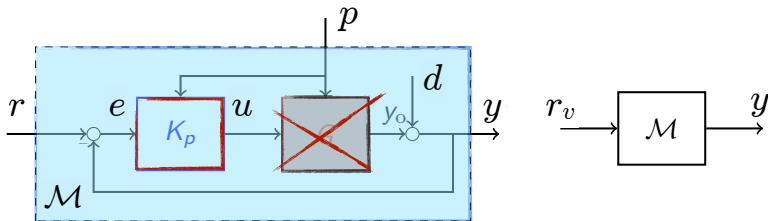
DIRECT DATA-DRIVEN MPC



- 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)

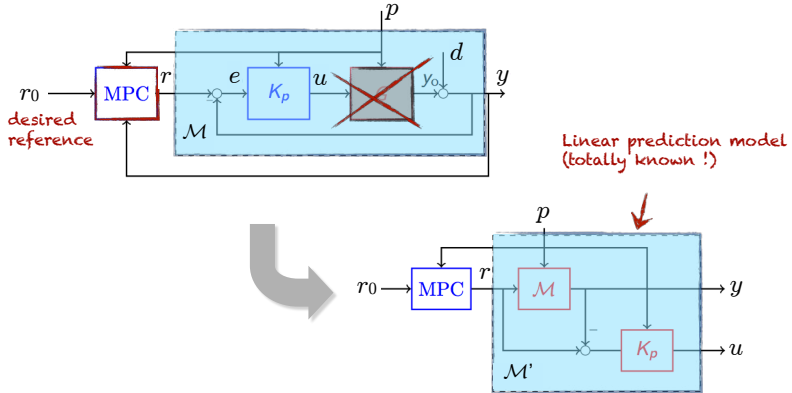


- Collect a set of **data** $\{u(t), y(t), p(t)\}, t = 1, \dots, N$
- Specify a **desired closed-loop linear model** \mathcal{M} from r to y
- Compute $r_v(t) = \mathcal{M}^\# y(t)$ from **pseudo-inverse model** $\mathcal{M}^\#$ of \mathcal{M}
- **Identify** linear (LPV) model K_p from $e_v = r_v - y$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

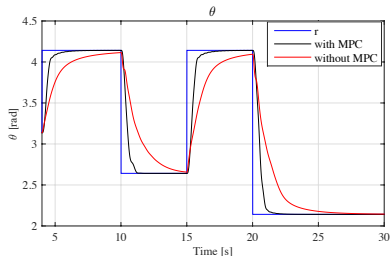
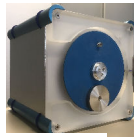


- MPC designed to handle input/output **constraints** and improve **performance**

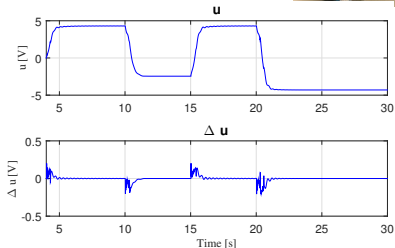
(Piga, Formentin, Bemporad, 2017)

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

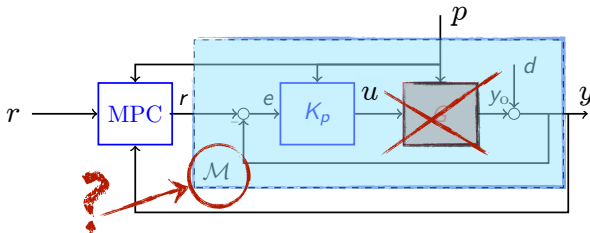


constraints on input
increments satisfied

No open-loop process model is 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**?

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

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

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

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

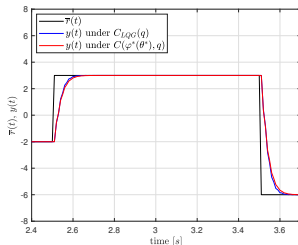
$$\Delta u_p(\theta, t) = u_p(\theta, t) - u_p(\theta, t-1)$$

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

- Results: **linear** process

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

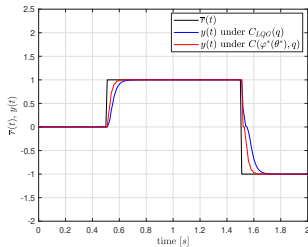
Data-driven controller **only 1.3% worse** than model-based LQR (=SYS-ID on same data + LQR design)



- 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

- 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:**

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

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)**

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

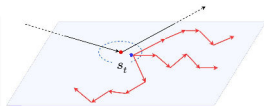
with $\mathcal{D}(K_{t-1})$ = descent direction

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 ,

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



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^* to K_t that stabilizes the local model

$$K_t^* = \underset{K}{\operatorname{argmin}} \|K - K_t^s\|_2^2$$

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

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

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

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

- 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:**

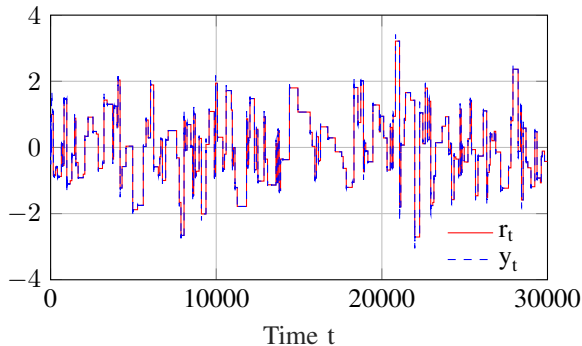
$$\pi_K(s_t, r_t) = -K^s \cdot s_t - K^r \cdot r_t, \quad K = \begin{bmatrix} K^s \\ K^r \end{bmatrix}$$

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, $d_t = 0$):

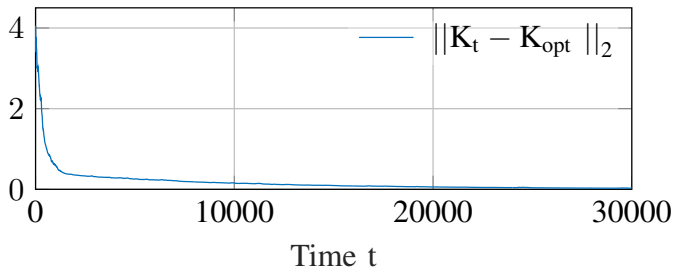


$$\begin{aligned} Q_y &= 1 \\ R &= 0.1 \\ Q_q &= 1 \end{aligned}$$

n_i	n_o	L
3	3	20
N_0	N_r	N_q
50	1	10

EXAMPLE: RETRIEVE LQR FROM DATA

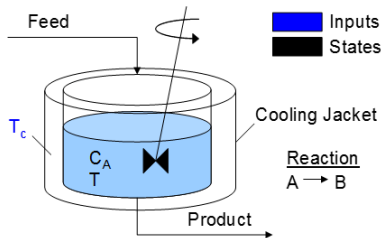
Evolution of the error $\|K_t - K_{opt}\|_2$:



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

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

NONLINEAR EXAMPLE



model is unknown

Feed:

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

Continuously Stirred Tank Reactor (CSTR)

apmonitor.com

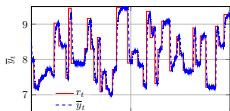
$$T = \hat{T} + \eta_T, \quad C_A = \hat{C}_A + \eta_C, \quad \eta_T, \eta_C \sim \mathcal{N}(0, \sigma^2), \quad \sigma = 0.01$$

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

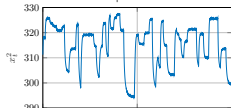
NONLINEAR EXAMPLE

Online learning

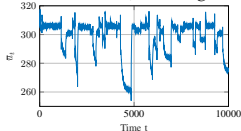
concentration C_A and reference r_t



temperature T



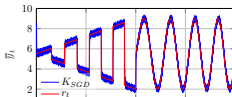
coolant temperature T_C



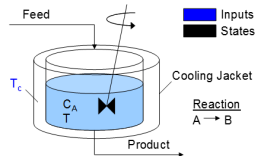
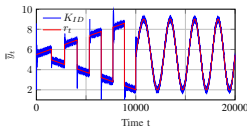
n_i	n_o	L
2	3	10
N_0	N_r	N_q
50	20	20

Validation phase

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



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



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)

"Model Predictive Control" - © 2023 A. Bemporad. All rights reserved.

LEARNING OPTIMAL MPC CALIBRATION

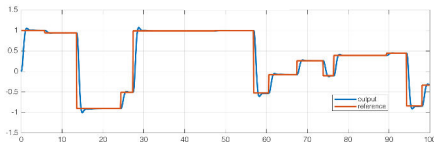
MPC CALIBRATION PROBLEM

- 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)



- **Auto-tuning** = find the best combination of parameters by solving the **global optimization problem**

$$\min_x f(x)$$

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)
 - 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** (Brochu, Cora, De Freitas, 2010)
 - Genetic algorithms (**GA**) (Holland, 1975)
 - Particle swarm optimization (**PSO**) (Kennedy, 2010)
 - ...
- New method: **radial basis function** surrogates + **inverse distance weighting** (**GLIS**) (Bemporad, 2020)

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

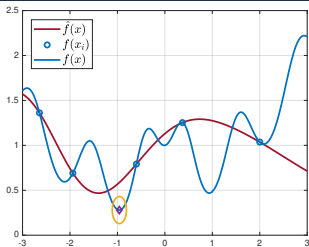


```
pip install glis
```

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

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

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



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

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

ϕ = 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 $i = 1, \dots, N$ (=linear system)

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

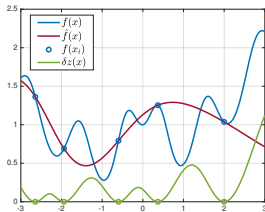
- **Step #2:** construct the **IDW exploration function**

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

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

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

ΔF = observed range of $f(x_i)$



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

$$x_{N+1} = \arg \min \quad \hat{f}(x) - \delta z(x)$$

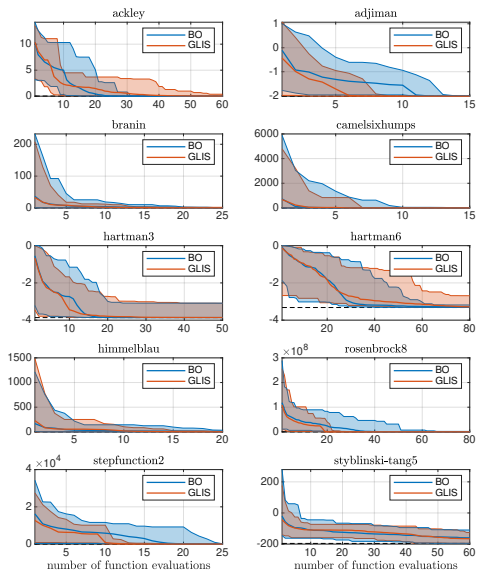
s.t. $\ell \leq x \leq u, g(x) \leq 0$

δ = exploitation vs
exploration tradeoff

to get new sample x_{N+1}

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

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

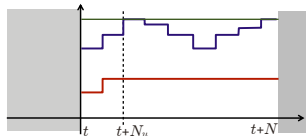
Results computed on 20 runs per test

BO = MATLAB's **bayesopt** fcn

AUTO-TUNING: MPC EXAMPLE

- We want to auto-tune the linear MPC controller

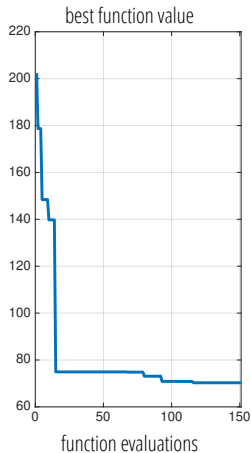
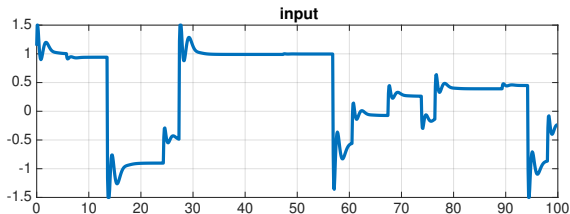
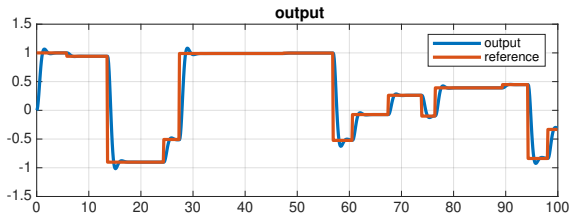
$$\begin{aligned}
 \min \quad & \sum_{k=0}^{50-1} (y_{k+1} - r(t))^2 + (W^{\Delta u} (u_k - u_{k-1}))^2 \\
 \text{s.t.} \quad & x_{k+1} = Ax_k + Bu_k \\
 & y_c = Cx_k \\
 & -1.5 \leq u_k \leq 1.5 \\
 & u_k \equiv u_{N_u}, \forall k = N_u, \dots, N-1
 \end{aligned}$$



- Calibration parameters: $x = [\log_{10} W^{\Delta u}, N_u]$
- 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: $x^* = [-0.2341, 2.3007]$

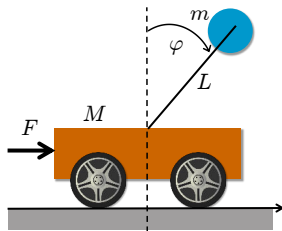


$$W^{\Delta u} = 0.5833, N_u = 2$$

MPC AUTOTUNING EXAMPLE

(Forgione, Piga, Bemporad, 2020)

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

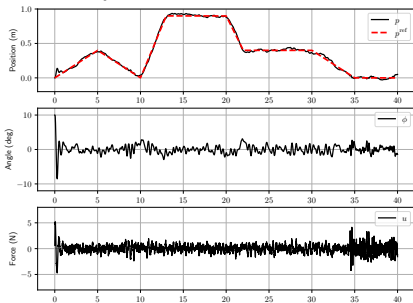


- **sample time**
- **weights** on outputs and input increments
- prediction and control **horizons**
- **covariance** matrices of Kalman filter
- absolute and relative **tolerances** of QP solver

- Closed-loop performance score: $J = \int_0^T |p(t) - p_{\text{ref}}(t)| + 30|\phi(t)| dt$
- MPC parameters tuned using 500 iterations of GLIS
- Performance tested with simulated cart on two hardware platforms (PC, Raspberry PI)

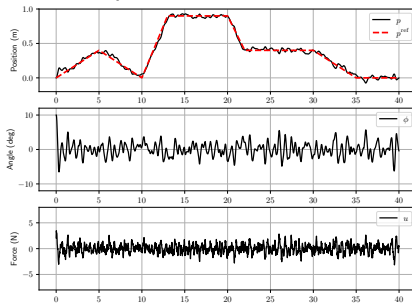
MPC AUTOTUNING EXAMPLE

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

- Objective function $f(x)$ is not available (**latent function**)
- We can only express a **preference** between two choices:

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

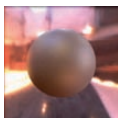
- We want to find a global optimum x^* (=“better” than any other x)

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

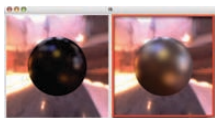
- **Active preference learning**: iteratively propose a new sample to compare
- **Key idea**: learn a **surrogate** of the (latent) objective function from preferences

PREFERENCE-LEARNING EXAMPLE

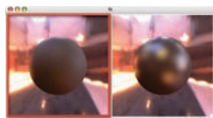
(Brochu, de Freitas, Ghosh, 2007)



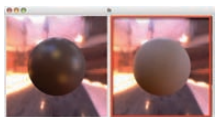
Target



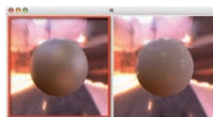
1



2



3

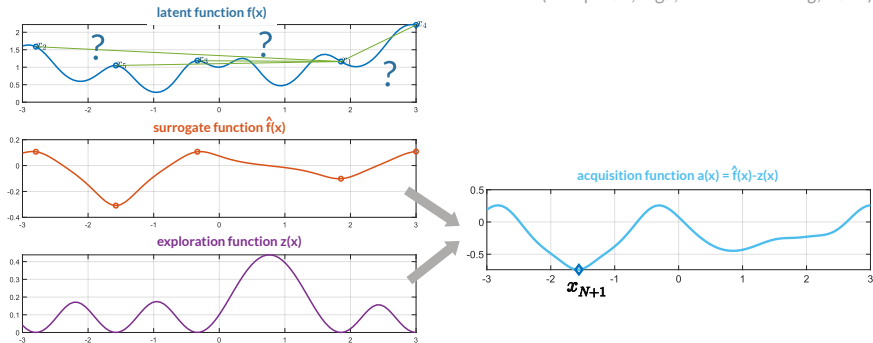


4

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

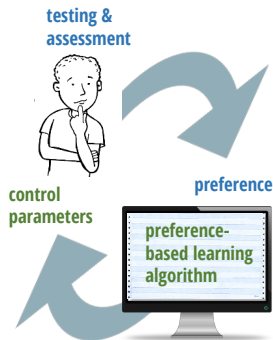
(Bemporad, Piga, *Machine Learning*, 2021)



- **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 PREFERENCE-BASED LEARNING

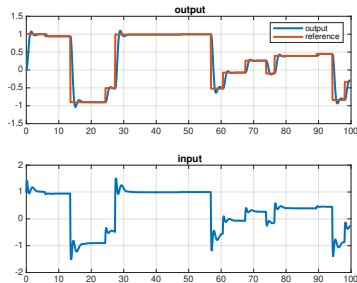
- Use **preference-based optimization (GLISp)** algorithm for **semi-automatic tuning** of MPC (Zhu, Bemporad, Piga, 2021)
- Latent function = calibrator's (unconscious) score of closed-loop MPC performance
- GLISp **proposes a new combination** x_{N+1} of MPC parameters to test
- By observing test results, 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

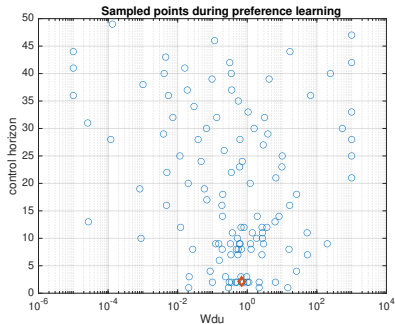
- Semi-automatic tuning of $x = [\log_{10} W^{\Delta u}, N_u]$ in linear MPC

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

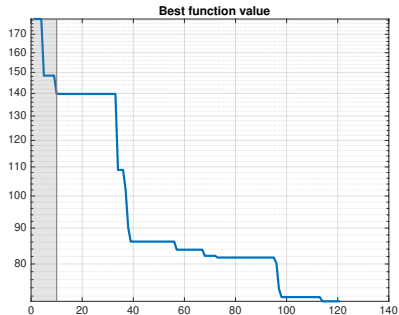


- Same performance index to assess closed-loop quality, but unknown: **only preferences** are available
- Result: $W^{\Delta u} = 0.6888, N_u = 2$

PREFERENCE-BASED TUNING: MPC EXAMPLE



tested combinations of MPC params



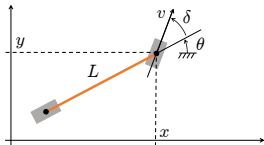
(latent) performance index

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”, ...



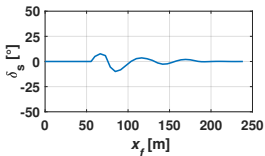
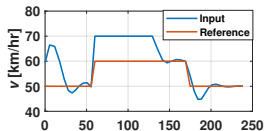
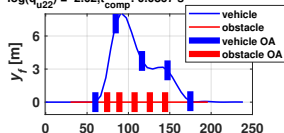
not easy to quantify in a single function

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

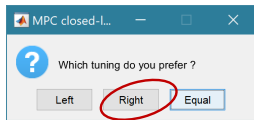
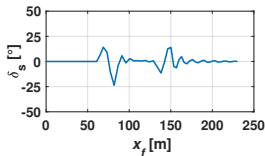
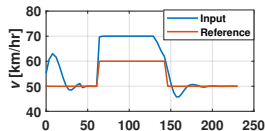
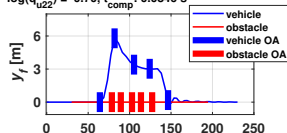
PREFERENCE-BASED TUNING: MPC EXAMPLE

- Preference query window:

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

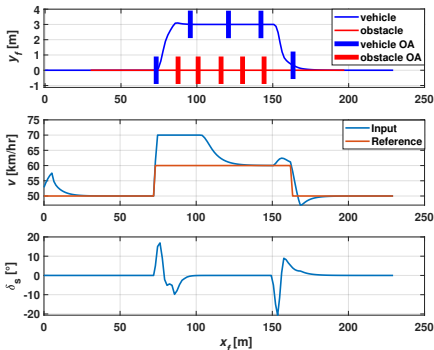


$T_s = 0.243$ s, $N_u = 12$, $N_p = 17$, $\log(q_{u11}) = 0.19$,
 $\log(q_{u22}) = 0.70$, $t_{\text{comp}} = 0.0846$ s



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 $\Delta v = 1.82$
- 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)

CORNER-CASE DETECTION

CORNER-CASE DETECTION PROBLEM

(Zhu, Bemporad, Kneissl, Esen, 2022)

- **Goal:** detect **undesired simulation scenarios** (= **corner-cases**)
- Let x = parameters defining the scenario, \mathcal{X}_{ODD} = **operational design domain**
 $x \in \mathcal{X}_{\text{ODD}} \subseteq \mathbb{R}^n$
- **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**

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

$f(x)$ = 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

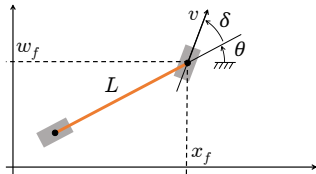
- **Problem:** find critical scenarios in automated driving w/ obstacles
- **MPC controller** for lane-keeping and obstacle-avoidance based on simple kinematic bicycle model (Zhu, Piga, Bemporad, 2021)

$$\dot{x}_f = v \cos(\theta + \delta)$$

$$\dot{w}_f = v \sin(\theta + \delta)$$

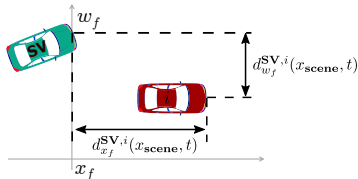
$$\dot{\theta} = \frac{v \sin(\delta)}{L}$$

$(x_f, w_f) = \text{front-wheel position}$



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

$$\begin{aligned} \min_{\ell \leq x \leq u} \quad & \sum_{i=1}^k d_{x_f, \text{critical}}^{\text{SV}, i}(x) + d_{w_f, \text{critical}}^{\text{SV}, i}(x) \\ \text{s.t.} \quad & \text{other constraints} \end{aligned}$$



CORNER-CASE DETECTION: CASE STUDY

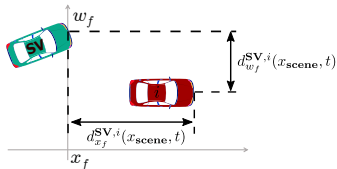
- Cost function terms** to minimize: for each obstacle $\#i$ define

$$d_{x_f, \text{critical}}^{\text{SV}, i}(x) = \begin{cases} \min_{t \in T_{\text{collision}}} d_{x_f}^{\text{SV}, i}(x, t) & \mathcal{I}_{\text{collision}}^i & \text{min time of collision with } \#i \\ L & \sim \mathcal{I}_{\text{collision}}^i \& \mathcal{I}_{\text{collision}} & \text{collision with other } \#j \neq \#i \\ \sum_{t \in T_{\text{sim}}} d_{x_f}^{\text{SV}, i}(x, t) & \sim \mathcal{I}_{\text{collision}} & \text{no collision} \end{cases}$$

$$d_{w_f, \text{critical}}^{\text{SV}, i}(x) = \begin{cases} \min_{t \in T_{\text{collision}}} d_{w_f}^{\text{SV}, i}(x, t) & \mathcal{I}_{\text{collision}}^i \\ w_{f, \text{safe}} & \sim \mathcal{I}_{\text{collision}}^i \& \mathcal{I}_{\text{collision}} \\ \sum_{t \in T_{\text{sim}}} d_{w_f}^{\text{SV}, i}(x, t) & \sim \mathcal{I}_{\text{collision}} \end{cases}$$

$$\mathcal{I}_{\text{collision}}^i = \text{true} \quad \text{if } \exists t \in T_{\text{sim}} \text{ s.t.} \\ (d_{x_f}^{\text{SV}, i}(x, t) \leq L) \& (d_{w_f}^{\text{SV}, i}(x, t) \leq W)$$

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

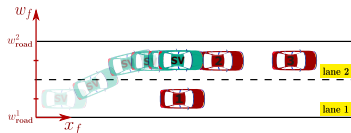


CORNER-CASE DETECTION: CASE STUDY

- 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

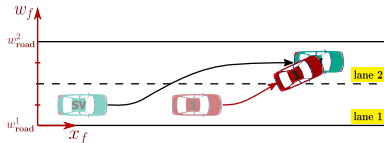


Ego car changes lane to avoid #1, but cannot brake fast enough to avoid #2

- 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** is a formidable combination for advanced control:
 - **MPC** / online optimization is an extremely powerful control methodology
 - **ML** extremely useful to get **control-oriented models** and **control laws** from **data**
- Ignoring **ML** tools would be a mistake (a lot to “learn” from machine learning)
- **ML** cannot replace control engineering:
 - **Black-box** modeling can be a failure. Better use **gray-box** models when possible
 - Approximating the control law can be a failure. Don't abandon online optimization
 - Pure AI-based **reinforcement learning** methods can be also a failure
- A wide spectrum of research opportunities and new practices is open !

