MODEL PREDICTIVE CONTROL

LEARNING-BASED MPC

Alberto Bemporad

imt.lu/ab
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
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
• **MPC** and **ML** = main trends in control R&D in industry!

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

"Model Predictive Control" - © A. Bemporad. All rights reserved.
• Massive set of techniques to extract mathematical models from data

- Linear PCA
- Nonlinear PCA
- Autoencoders
- ...

- Ridge classification
- Logistic regression
- Naïve Bayes classification
- ...

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

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

- K-means clustering
- Density-based spatial clustering
- ...

- Dimensionality Reduction
- Supervised Learning
- Unsupervised Learning
- Classification
- Regression
- Reinforcement Learning
• **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
**CONTROL-ORIENTED NONLINEAR MODELS**

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

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

- **Gray-box modeling** is a mix of the two. It can be quite effective.

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

*(George E. P. Box)*
• 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
  (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 $x_{t+1} \approx \mathcal{N}_x(x_t, u_t), \ y_t \approx \mathcal{N}_y(x_t)$
  - I/O data only: set $x_t =$ value of an inner layer of the network (Prasad, Bequette, 2003)

- 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 ...)

"Model Predictive Control" - © A. Bemporad. All rights reserved.
• **Approach:** use a neural network model for prediction

![Diagram of NLMPC based on Neural Networks](image)

- Model Predictive Control (MPC) design workflow:
  - 1. Collect data
  - 2. Train neural model
  - 3. Codegen
  - 4. Deploy NLMPC controller
MPC OF ETHYLENE OXIDATION PLANT

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

\[
\begin{align*}
C_2H_4 + \frac{1}{2}O_2 & \rightarrow C_2H_4O \\
C_2H_4 + 3O_2 & \rightarrow 2CO_2 + 2H_2O \\
C_2H_4O + \frac{5}{2}O_2 & \rightarrow 2CO_2 + 2H_2O
\end{align*}
\]

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

\[
\begin{align*}
\dot{x}_1 &= u_1(1 - x_1 x_4) \\
\dot{x}_2 &= u_1(u_2 - x_2 x_4) - A_1 e^{\frac{x_4}{x_1}} (x_2 x_4)^{\frac{1}{2}} - A_2 e^{\frac{x_4}{x_1}} (x_2 x_4)^{\frac{1}{4}} \\
\dot{x}_3 &= -u_1 x_3 x_4 + A_1 e^{\frac{x_4}{x_1}} (x_2 x_4)^{\frac{1}{2}} - A_3 e^{\frac{x_4}{x_1}} (x_3 x_4)^{\frac{1}{2}} \\
\dot{x}_4 &= \frac{u_1(1-x_4) + B_1 e^{\frac{x_4}{x_1}} (x_2 x_4)^{\frac{1}{2}} + B_2 e^{\frac{x_4}{x_1}} (x_3 x_4)^{\frac{1}{4}}}{x_1} \\
&\quad \quad + \frac{B_3 e^{\frac{x_4}{x_1}} (x_3 x_4)^{\frac{1}{2}} - B_4 (x_4 - T_C)}{x_1} \\
y &= x_3
\end{align*}
\]

- \(u_1\) = manipulated variables, \(x_3\) = controlled output, \(u_2\) = measured disturbance
• 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
\[ \rightarrow \text{112 coefficients} \]

• NN model trained by **ODYS Deep Learning** toolset
(model fitting + Jacobians \( \rightarrow \) neural model in C)

• Model validated on 200 samples.
\( x_{3,k+1} \) reproduced from \( x_k, u_k \) with max 0.4% error

"Model Predictive Control" - © A. Bemporad. All rights reserved.
MPC OF ETHYLENE OXIDATION PLANT

- **MPC settings:**
  - sampling time $T_s = 5$ s
  - 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
• 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)

\[
O_k = [y'_k \ldots y'_{k-m}]'
\]
\[
I_k = [y'_k \ldots y'_{k-n_a+1} \ u'_k \ldots u'_{k-n_b+1}]'
\]
Learning nonlinear state-space models for MPC

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

\[
\min_{f, d, e} \sum_{k=k_0}^{N-1} \alpha \left( \ell_1(\hat{O}_k, O_k) + \ell_1(\hat{O}_{k+1}, O_{k+1}) \right) + \beta \ell_2(x^*_{k+1}, x_{k+1}) + \gamma \ell_3(O_{k+1}, O^*_{k+1})
\]

\[\text{s.t. } x_k = e(I_{k-1}), k = k_0, \ldots, N\]
\[x^*_{k+1} = f(x_k, u_k), k = k_0, \ldots, N - 1\]
\[\hat{O}_k = d(x_k), O^*_k = d(x^*_k), k = k_0, \ldots, N\]

- Model complexity reduction: add group-LASSO penalties on subsets of weights

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

\[f(x_k, u_k) = A(x_k, u_k) \begin{bmatrix} x_k \\ 1 \end{bmatrix} + B(x_k, u_k)u_k \]
\[y_k = C(x_k, u_k) \begin{bmatrix} x_k \\ 1 \end{bmatrix} \]

- 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 $\ell_4(\hat{x}_{k+1}, x_{k+1})$
The performance achieved with the derivative-based controller suggests that an LTV-MPC formulation might also work well. We also assess its robustness using a model achieving 61% BFR in open loop. Computation time per step: ~40ms.

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

\[
\begin{align*}
    x_1(t+1) &= x_1(t) - k_1 \sqrt{x_1(t)} + k_2 u(t) \\
    x_2(t+1) &= x_2(t) + k_3 \sqrt{x_1(t)} - k_4 \sqrt{x_2(t)} \\
    y(t) &= x_2(t) + u(t)
\end{align*}
\]

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

www.mathworks.com

LTV-MPC results
• Alternative: **learn the entire prediction**

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

• **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: \( \bar{u}_k = \text{MPC sequence optimized @} k - 1 \)

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

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

\[ e_{\text{FIT}} = \max \left\{ 0, 1 - \frac{\| \hat{y} - y \|_2}{\| y - \bar{y} \|_2} \right\} \]

• Closed-loop LTV-MPC results:

• Model complexity reduction: add **group-LASSO** term with penalty \( \lambda \)

<table>
<thead>
<tr>
<th>( \lambda )</th>
<th>( e_{\text{FIT}} ) (average on all prediction steps)</th>
<th># nonzero weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>.01</td>
<td>0.853</td>
<td>328</td>
</tr>
<tr>
<td>0.005</td>
<td>0.868</td>
<td>363</td>
</tr>
<tr>
<td>0.001</td>
<td>0.901</td>
<td>556</td>
</tr>
<tr>
<td>0.0005</td>
<td>0.911</td>
<td>888</td>
</tr>
<tr>
<td>0</td>
<td>0.917</td>
<td>9000</td>
</tr>
</tbody>
</table>
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
**Feedforward neural network model:**

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

Examples: \( x_k = \) measured state, 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!**
• **Key idea**: treat parameter vector $\theta$ of the feedforward neural network as a constant state

\[
\begin{align*}
\theta_{k+1} &= \theta_k + \eta_k \\
y_k &= f(x_k, \theta_k) + \zeta_k
\end{align*}
\]

and use EKF to estimate $\theta_k$ 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 $\theta$

• Implemented in **ODYS Deep Learning** library

• Extended to rather arbitrary convex loss functions/regularization terms

(Bemporad, 2021 - https://arxiv.org/abs/2111.02673)
• **Recurrent Neural Network (RNN) model:**

\[
\begin{align*}
x_{k+1} &= f_x(x_k, u_k, \theta_x) \\
y_k &= f_y(x_k, \theta_y)
\end{align*}
\]

\(f_x, f_y = \text{feedforward neural network}\)

• **Training problem:** given a dataset \(\{u_0, y_0, \ldots, u_{N-1}, y_{N-1}\}\) solve

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

• **Main issue:** \(x_k\) are **hidden states**, i.e., are **unknowns** of the problem
• Estimate both hidden states $x_k$ and parameters $\theta_x, \theta_y$ by EKF based on

\[
\begin{align*}
    x_{k+1} &= f_x(x_k, u_k, \theta_x) + \xi_k \\
    \theta_{x(k+1)} &= \theta_{xk} + \eta_k \\
    \theta_{y(k+1)} &= \theta_{yk} + \eta_k \\
    y_k &= f_y(x_k, \theta_{yk}) + \zeta_k
\end{align*}
\]

• 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 strictly convex loss functions/regularization terms

• Can add $\ell_1$-penalty $\lambda \left\| \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} \right\|_1$ to sparsify $\theta_x, \theta_y$ by changing EKF update into

\[
\begin{align*}
    \begin{bmatrix}
        \hat{x}(k|k) \\
        \theta_x(k|k) \\
        \theta_y(k|k)
    \end{bmatrix}
    &= 
    \begin{bmatrix}
        \hat{x}(k|k-1) \\
        \theta_x(k|k-1) \\
        \theta_y(k|k-1)
    \end{bmatrix}
    + M(k)e(k) - \lambda P(k|k-1)
    \begin{bmatrix}
        \text{sign}(\theta_x(k|k-1)) \\
        \text{sign}(\theta_y(k|k-1))
    \end{bmatrix}
\end{align*}
\]
TRAINING RNNs BY EKF - EXAMPLES

- Dataset: 3499 I/O data of **magneto-rheological fluid damper** (Wang et al., 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** each, **leaky-ReLU** activation

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

- Training time measured on MATLAB+CasADi implementation of EKF/AMSGrad
• Compare NRMSE\(^1\) wrt NNARX model (SYS-ID TBX):

EKF = 91.97, AMSGrad = 85.58, NNARX(6,2) = 88.18 (training)
EKF = 90.54, AMSGrad = 80.95, NNARX(6,2) = 85.15 (test)

• Repeat training with $\ell_1$-penalty $\lambda \left\| \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} \right\|_1$

\(^1\)normalized root-mean-square error
Training RNNs by EKF - Examples

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

\[
x(k + 1) = \begin{bmatrix} 0.8 & 0.2 & -1 \\ 0.1 & -1 & 0.7 \end{bmatrix} x(k) + \begin{bmatrix} -1 \\ 1 \end{bmatrix} u(k) + \xi(k) \quad \text{Var}[\xi_i(k)] = \sigma^2
\]

\[
y(k) = \begin{cases} 1 & \text{if } (-2 \ 1.5 \ 0.5) x(k) - 2 + \zeta(k) \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad \text{Var}[\zeta(k)] = \sigma^2
\]

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

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

\[
f^y_1(y) = 1/(1 + e^{-A^y_1[x'(k) u(k)]' - b^y_1})
\]

- Training loss: (modified) cross-entropy loss

\[
\ell_{CE\epsilon}(y(k), \hat{y}) = \sum_{i=1}^{n_y} -y_i(k) \log(\epsilon + \hat{y}_i) - (1 - y_i(k)) \log(1 + \epsilon - \hat{y}_i)
\]
• RNN training problem = **optimal control** problem:

$$\min_{\theta_x, \theta_y, x_0, x_1, \ldots, x_{N-1}} \ r(x_0, \theta_x, \theta_y) + \sum_{k=0}^{N-1} \ell(y_k, \hat{y}_k)$$

subject to

$$x_{k+1} = f_x(x_k, u_k, \theta_x)$$

$$\hat{y}_k = f_y(x_k, \theta_y)$$

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

• Linearized model:

$$\Delta x_{k+1} = (\nabla_x f_x)' \Delta x_k + (\nabla_{\theta_x} f_x)' \Delta \theta_x$$

$$\Delta y_k = (\nabla_x f_y)' \Delta x_k + (\nabla_{\theta_y} f_y)' \Delta \theta_y$$

• **Idea**: take 2\textsuperscript{nd}-order expansions of the loss $\ell$ and regularization term $r$

and use **sequential least-squares** + line search to minimize wrt $x_0, \theta_x, \theta_y$
• Fluid-damper example:

![Graph 1](image1.png)

![Graph 2](image2.png)

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

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

s.t. \[ x_{k+1} = f_x(x_k, u_k, \theta_x) \]

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

\[
\min_{\theta_x, \theta_y, x_0, \nu_x, \nu_y} \ r(x_0, \theta_x, \theta_y) + \sum_{k=0}^{N-1} \ell(y_k, f_y(x_k, \theta_y)) + g(\nu_x, \nu_y)
\]

s.t. \[ x_{k+1} = f_x(x_k, u_k, \theta_x) \]

\[ \begin{bmatrix} \nu_x \\ \nu_y \end{bmatrix} = \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} \]
• ADMM + Seq. LS = **NAILS** algorithm (Nonconvex ADMM Iterations and Sequential LS)

\[
\begin{bmatrix}
    x_{0}^{t+1} \\
    \theta_{x}^{t+1} \\
    \theta_{y}^{t+1} \\
    \nu_{x}^{t+1} \\
    \nu_{y}^{t+1} \\
    w_{x}^{t+1} \\
    w_{y}^{t+1}
\end{bmatrix}
= \arg \min_{x_{0}, \theta_{x}, \theta_{y}} V(x_{0}, \theta_{x}, \theta_{y}) + \frac{\rho}{2} \left\| \begin{bmatrix}
    \theta_{x} - \nu_{x}^{t} + w_{x}^{t} \\
    \theta_{y} - \nu_{y}^{t} + w_{y}^{t}
\end{bmatrix} \right\|_{2}^{2}
\]

(sequential) LS

\[
= \text{prox} \frac{1}{\rho} g(\theta_{x}^{t+1} + w_{x}^{t}, \theta_{y}^{t+1} + w_{y}^{t})
\]

proximal step

\[
= \begin{bmatrix}
    w_{x}^{h} + \theta_{x}^{t+1} - \nu_{x}^{t+1} \\
    w_{y}^{h} + \theta_{y}^{t+1} - \nu_{y}^{t+1}
\end{bmatrix}
\]

update dual vars

• Fluid-damper example: **group-Lasso regularization**

\[ g(\nu_{i}^{g}) = \tau \sum_{i=1}^{n_{x}} \| \nu_{i}^{g} \|_{2} \]

to zero entire rows and columns and **reduce state-dimension** automatically
• Fluid-damper example: quantization of $\theta_x$, $\theta_y$ for simplifying model arithmetic
+ReLU activation function

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

$Q =$ multiples of 0.1 between -0.5 and 0.5

- NRMSE = 83.10 (training), 80.51 (test)
- NRMSE = 8.83 (training), 2.69 (test) ← no ADMM, just quantize after training
- Training time: $\approx 5$ s

• Note: no convergence to a global minimum is guaranteed

• NAILS = very flexible & efficient learning algorithm for control-oriented RNNs
• Computation time (Intel Core i9-10885H CPU @2.40GHz):

<table>
<thead>
<tr>
<th>language</th>
<th>autodiff</th>
<th>EKF /time step CPU time</th>
<th>seq. LS /epoch CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python 3.8.1</td>
<td>PyTorch</td>
<td>≈ 30 ms</td>
<td>(N/A)</td>
</tr>
<tr>
<td>Python 3.8.1</td>
<td>JAX</td>
<td>≈ 9 ms</td>
<td>≈ 1.0 s</td>
</tr>
<tr>
<td>Julia 1.7.1</td>
<td>Flux.jl</td>
<td>≈ 2 ms</td>
<td>≈ 0.8 s</td>
</tr>
</tbody>
</table>

• 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
• **Goal:** track desired longitudinal speed \( v_y \), lateral displacement \( e_y \) and orientation \( \Delta \Psi \)

• **Inputs:** wheel torque \( T_w \) and steering angle \( \delta \)

• **Constraints:** on \( e_y \) and lateral displacement \( s \) (for obstacle avoidance) and manipulated inputs \( T_w, \delta \)

• **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 \( (k_f, k_r) \) ⇒ small integration step required
  
  - learn a **black-box neural-network model**!

  (Boni, Capelli, Frascati @ODYS, 2021)

"Model Predictive Control" - © A. Bemporad. All rights reserved.
• **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 \((T_s = 100 \text{ ms})\)
• Model validation on test data:

One-step ahead prediction on test data

Open-loop predictions

• C-code (network+Jacobians) automatically generated for ODYS MPC
**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
Can we design an MPC controller without first identifying a model of the open-loop process?
• Collect a set of data \( \{u(t), y(t), p(t)\} \), \( t = 1, \ldots, N \)

• Specify a desired closed-loop linear model \( M \) from \( r \) to \( y \)

• Compute \( r_v(t) = M^# y(t) \) from pseudo-inverse model \( M^# \) of \( M \)

• Identify linear (LPV) model \( K_p \) from \( e_v = r_v - y \) (virtual tracking error) to \( u \)
• 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)
Experimental results: MPC handles soft constraints on $u$, $\Delta u$ 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!
**Question:** How to choose the reference model $M$?

Can we improve the closed-loop performance and impose input/output constraints?

Can we choose $M$ 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 $M(\theta)$ and optimize

\[
\min_{\theta} J(\theta) = \frac{1}{N} \sum_{t=0}^{N-1} W_y (r(t) - y_p(\theta, t))^2 + W_{\Delta u} \Delta u_p^2(\theta, t) + W_{fit} (u(t) - u_v(\theta, t))^2
\]

- **Performance index**

- **Identification error**

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

\[
y_p(\theta, t) = M(\theta) r(t) \quad u_p(\theta, t) = K_p(\theta) (r(t) - y_p(\theta, t))
\]

\[
\Delta u_p(\theta, t) = u_p(\theta, t) - u_p(\theta, t - 1)
\]

- Optimal $\theta$ 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

\[ y_L(t) = G(z)u(t) \]
\[ y(t) = |y_L(t)| \arctan(y_L(t)) \]

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**: \( \pi : \mathbb{R}^{n_s + n_p} \rightarrow \mathbb{R}^{n_u} \) deterministic control policy
  \[ u_t = \pi(s_t, p_t) \]

- Closed-loop **performance** of an execution is defined as
  \[ J_\infty(\pi, s_0, \{p_\ell, d_\ell\}_{\ell=0}^\infty) = \sum_{\ell=0}^\infty \rho(s_\ell, p_\ell, \pi(s_\ell, p_\ell)) \]
  \[ \rho(s_\ell, p_\ell, \pi(s_\ell, p_\ell)) = \text{stage cost} \]
Optimal Policy Search Problem

- **Optimal policy:**

\[
\pi^* = \arg\min_{\pi} \mathcal{J}(\pi)
\]

\[
\mathcal{J}(\pi) = \mathbb{E}_{s_0, \{p_\ell, d_\ell\}} [\mathcal{J}_\infty(\pi, s_0, \{p_\ell, d_\ell\})]
\]

(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
The descent direction $D(K_{t-1})$ 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 $L$,

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

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 $\pi$ 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
At each step $t$:

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

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

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

s.t. $K$ stabilizes local linear model

When policy is learned online, **exploration** is guaranteed by the reference $r_t$
SPECIAL CASE: OUTPUT TRACKING

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

\( \Delta u_t = u_t - u_{t-1} \)  \hspace{1em} control input increment

• Stage cost:

\[
\| y_{t+1} - r_t \|_2^2 Q_y + \| \Delta u_t \|_2^2 R + \| q_{t+1} \|_2^2 Q_q
\]

• Integral action dynamics

\[
q_{t+1} = q_t + (y_{t+1} - r_t)
\]

\[
\begin{bmatrix}
 x_t \\
 q_t
\end{bmatrix} \quad \text{and} \quad p_t = r_t
\]

• Linear policy parametrization:

\[
\pi_K(s_t, r_t) = -K^s \cdot s_t - K^r \cdot r_t, \quad K = \begin{bmatrix} K^s \\ K^r \end{bmatrix}
\]
\[
\begin{align*}
{x}_{t+1} &= \begin{bmatrix} -0.669 & 0.378 & 0.233 \\ -0.288 & -0.147 & -0.638 \\ -0.337 & 0.589 & 0.043 \end{bmatrix} {x}_t + \begin{bmatrix} -0.295 \\ -0.325 \\ -0.258 \end{bmatrix} {u}_t \\
{y}_t &= \begin{bmatrix} -1.139 & 0.319 & -0.571 \end{bmatrix} {x}_t
\end{align*}
\]

model is unknown

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

- \(Q_y = 1\)
- \(R = 0.1\)
- \(Q_q = 1\)

<table>
<thead>
<tr>
<th>(n_i)</th>
<th>(n_o)</th>
<th>(L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(N_0)</th>
<th>(N_r)</th>
<th>(N_q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
Evolution of the error $\|K_t - K_{opt}\|_2$:

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

$K_{opt} = [-1.257, 0.219, 0.653, 0.898, 0.050, 1.141, -2.196]$
Continuously Stirred Tank Reactor (CSTR)

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

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

\[
Q_y = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \quad R = 0.1 \quad Q_q = \begin{bmatrix} 0.01 & 0 \\ 0 & 0 \end{bmatrix}
\]
**Nonlinear Example**

Online learning concentration $C_A$ and reference $r_t$

Temperature $T$

Coolant temperature $T_C$

Validation phase

Cost of $K_{SGD} = 4.3 \cdot 10^3$

Cost of $K_{ID} = 2.4 \cdot 10^4$

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

(Ferrarotti, Bemporad, 2020a) (Ferrarotti, Bemporad, 2020b) (Ferrarotti, Breschi, Bemporad, 2021) "Model Predictive Control" - © 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)
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)
**Goal**: solve the global optimization problem

\[
\min_x f(x) \\
\text{s.t. } \ell \leq x \leq u \\
g(x) \leq 0
\]

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

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

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

\(\phi = \text{radial basis function}\)

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

(inverse quadratic)

Vector \(\beta\) solves \(\hat{f}(x_i) = f(x_i)\) for all \(i = 1, \ldots, N\) (=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 \( x \in \{x_1, \ldots, x_N\} \)

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

\( \Delta F = \) observed range of \( f(x_i) \)

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

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

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

\( \delta = \) exploitation vs exploration tradeoff

\[ \Delta F = \] observed range of \( f(x_i) \)

• **Iterate the procedure to get new samples** \( x_{N+2}, \ldots, x_{N_{\text{max}}} \)
GLIS VS BAYESIAN OPTIMIZATION

Results computed on 20 runs per test

BO = MATLAB's bayesopt fcn

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

"Model Predictive Control" - © A. Bemporad. All rights reserved.
We want to auto-tune the linear MPC controller

\[
\begin{align*}
\min_{k=0}^{50-1} & \sum (y_{k+1} - r(t))^2 + (W^{\Delta u}(u_k - u_{k-1}))^2 \\
\text{s.t.} & \quad x_{k+1} = Ax_k + Bu_k \\
& \quad y_c = Cx_k \\
& \quad -1.5 \leq u_k \leq 1.5 \\
& \quad u_k \equiv u_{N_u}, \forall k = N_u, \ldots, N - 1
\end{align*}
\]

- 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:

\[
f(x) = \sum_{t=0}^{T} (y(t) - r(t))^2 + \frac{1}{2}(u(t) - u(t-1))^2 + \frac{2N_u}{2}
\]

\( \underbrace{\text{track well}}_{\text{smooth control action}} \underbrace{\text{smooth control action}}_{\text{small QP}} \)
Result: $\mathbf{x}^* = [-0.2341, 2.3007]$  \quad \Rightarrow \quad 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 optimized for **desktop PC**

- Position: \( p \) vs. \( p_{ref} \)
- Angle: \( \phi \)
- Force: \( u \)

**Optimal sample time:** 6 ms

MPC optimized for **Raspberry PI**

- Position: \( p \) vs. \( p_{ref} \)
- Angle: \( \phi \)
- Force: \( u \)

**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:

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

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

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

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

- 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

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

\[
\min \sum_{k=0}^{50-1} (y_{k+1} - r(t))^2 + (W^{\Delta u} (u_k - u_{k-1}))^2
\]

s.t. \( x_{k+1} = Ax_k + Bu_k \)
\( y_c = Cx_k \)
\( -1.5 \leq u_k \leq 1.5 \)
\( u_k \equiv u_{N_u}, \forall k = N_u, \ldots, N - 1 \)

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

\[
\begin{align*}
\dot{x} &= v \cos(\theta + \delta) \\
\dot{y} &= v \sin(\theta + \delta) \\
\dot{\theta} &= \frac{1}{L} v \sin(\delta)
\end{align*}
\]

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 query window:

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

\[ T_s = 0.243 \text{ s}, N_u = 12, N_p = 17, \log(q_{u11}) = 0.19, \log(q_{u22}) = 0.70, t_{\text{comp}}: 0.0846 \text{ s} \]
• 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)
• **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!