System Identification – OE method

Costandin Marius

1 Theory

Let $t_0, t_1, \ldots, t_N$ be some time moments in arithmetic progression, $t_{i+1} - t_i = T_s$, with the ratio being the sampling time. We denote the output of the process by the letter y and the input of the process by the letter u. Furthermore, for simplicity of notation, let $y_k = y(t_k)$ and $u_k = u(t_k)$, that is the output of the process which is measured at the time moment t_k respectively the input which is given to the process at the time moment t_k.

Then let

$$\mathcal{Y}(z) = \sum_{k\in\mathbb{Z}} y_k \cdot z^{-k} \quad \mathcal{U}(z) = \sum_{k\in\mathbb{Z}} u_k \cdot z^{-k} \tag{1}$$

be the so called $\mathcal{Z}$ transforms of the sequences $(y_k)_{k\in\mathbb{Z}}$ and $(u_k)_{k\in\mathbb{Z}}$ with $y_k = 0$ for an integer $k \leq 0$ and $u_k = 0$ for an integer $k < 0$.

The OE method considers the following input-output relation:

$$\mathcal{Y}(z) = \frac{B(z)}{F(z)} \cdot z^{-n_p} \cdot \mathcal{U}(z) + \mathcal{E}(z) \tag{2}$$

where

$$\begin{aligned} B(z) &= b_1 \cdot z^{-1} + \ldots + b_{n_b} \cdot z^{-n_b} \\ F(z) &= 1 + f_1 \cdot z^{-1} + \ldots + f_{n_f} \cdot z^{-n_a} \end{aligned} \tag{3}$$

with $n_f, n_b, n_p \in \mathbb{N} \cup \{0\}$. Multiplying both sides by $F(z)$ and applying the inverse $\mathcal{Z}$ transform, one obtains:

$$\begin{aligned} y_k + f_1 \cdot y_{k-1} + \ldots + f_{n_f} \cdot y_{k-n_f} =& b_1 \cdot u_{k-n_p-1} + \ldots + b_{n_b} \cdot u_{k-n_p-n_b} + \\ &+ e_k + f_1 \cdot e_{k-1} + \ldots + f_{n_f} \cdot e_{k-n_f} \end{aligned} \tag{4}$$

hence it is obtained:

$$\begin{aligned} e_k = y_k + & f_1 \cdot y_{k-1} + \ldots + f_{n_f} \cdot y_{k-n_f} - b_1 \cdot u_{k-n_p-1} - \ldots - b_{n_b} \cdot u_{k-n_p-n_b} - \\ & - f_1 \cdot e_{k-1} - \ldots - f_{n_f} \cdot e_{k-n_f} \end{aligned} \tag{5}$$

Let, as before,

$$J = \sum_{k=1}^{N} e_k^2 \tag{6}$$

It is obvious, from (5), that once y_k and u_k are known (measured), J depends upon the choice of $\theta = \begin{bmatrix} f_1 & \ldots & f_{n_f} & b_1 & \ldots & b_{n_b} \end{bmatrix}^T$. We search for θ such that J is minimized. This, unlike the ARX case, is a nonlinear optimization problem in the variable θ. Indeed, in the ARX case, one finds θ by simply letting $\frac{\partial J}{\partial \theta} = 0n \times 1$ (where n is the length of θ), which turns out to be a linear equation in θ, see ARX method. In order to further exemplify this problem, assume the simpler case where $n_f = n_b = 1, n_p = 0$:

$$\begin{aligned} e_k &= y_k + f \cdot y_{k-1} + b \cdot u_{k-1} - f \cdot e_{k-1} \\ \frac{\partial e_k}{\partial f} &= y_{k-1} - e_{k-1} - f \cdot \frac{\partial e_{k-1}}{\partial f} \\ \frac{\partial e_k}{\partial b} &= \ldots \end{aligned} \tag{7}$$

If one requires

$$\begin{cases} \frac{\partial J}{\partial f} = 2 \cdot \sum_{k=1}^{N} e_k \cdot \frac{\partial e_k}{\partial f} = 0 \\ \frac{\partial J}{\partial b} = 2 \cdot \sum_{k=1}^{N} e_k \cdot \frac{\partial e_k}{\partial b} = 0 \end{cases} \tag{8}$$

then replacing (7) in (8), one can see that the equations do NOT remain linear in f and b.

In this condition, we search for θ which minimizes J using an iterative algorithm. Start with a 'guess' of θ_0 (for instance initialize θ with random numbers) and then 'improve' at each iteration. There are many ways to make this 'improvement', see optimization algorithms.

Many optimization algorithms function in the following way: at each iteration/improvement l, they provide a direction V_l and a scalar α_l. The new computed θ_{l+1} will be:

$$\theta_{l+1} = \theta_l + \alpha_l \cdot V_l \tag{9}$$

For this laboratory, we will use/be inspired from, the so called, Gauss-Newton algorithm:

$$\theta_{l+1} = \theta_l - \alpha \cdot H^{-1} \cdot \nabla^T J \tag{10}$$

therefore, the scalar will be α which remains constant for all iterations, and the direction is $V_l = -H^{-1} \cdot \nabla^T J$. The exact meaning of these will follow shortly. In order to elegantly explain who H is, we will recall the so called 'nabla' operator:

$$\nabla = \begin{bmatrix} \frac{\partial}{\partial f_1} & \cdots & \frac{\partial}{\partial b_{n_b}} \end{bmatrix} \tag{11}$$

therefore $\nabla^T J = \frac{\partial J}{\partial \theta} = \begin{bmatrix} \frac{\partial J}{\partial f_1} \\ \vdots \\ \frac{\partial J}{\partial b_{n_b}} \end{bmatrix}$. With this in mind, one defines $H = \nabla^T \cdot \nabla \cdot J$, the Hessian matrix. Therefore:

$$\begin{aligned} \nabla^T J &= \sum_{k=1}^{N} \nabla^T e_k^2 = 2 \cdot \sum_{k=1}^{N} e_k \cdot \frac{\partial e_k}{\partial \theta} \\ H = \nabla^T \cdot \nabla J &= \begin{bmatrix} \frac{\partial}{\partial f_1} \\ \vdots \\ \frac{\partial}{\partial b_{n_b}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial}{\partial f_1} & \cdots & \frac{\partial}{\partial b_{n_b}} \end{bmatrix} J = \sum_{k=1}^{N} \nabla^T \cdot \nabla \cdot e_k^2 \\ &= 2 \cdot \sum_{k=1}^{N} \nabla^T \cdot (e_k \cdot \nabla e_k) = 2 \cdot \sum_{k=1}^{N} \nabla^T e_k \cdot \nabla e_k + 2 \cdot \sum_{k=1}^{N} e_k \cdot \nabla^T \nabla e_k \end{aligned} \tag{12}$$

However, for this algorithm, we will approximate (Gauss-Newton) the Hessian matrix with just its first sum (this is convenient, since we are not required to compute the second derivative of the error, which would be even more complicated than the first, as will be seen in the sequel):

$$H \approx 2 \cdot \sum_{k=1}^{N} \frac{\partial e_k}{\partial \theta} \cdot \frac{\partial e_k^T}{\partial \theta} \tag{13}$$

where $\frac{\partial e_k}{\partial \theta}$ is found by integrating the following dynamical system:

$$\begin{cases} \frac{\partial e_k}{\partial f_1} = y_{k-1} - e_{k-1} - f_1 \cdot \frac{\partial e_{k-1}}{\partial f_1} - \ldots - f_{n_f} \cdot \frac{\partial e_{k-n_f}}{\partial f_1} \\ \frac{\partial e_k}{\partial f_2} = y_{k-2} - e_{k-2} - f_1 \cdot \frac{\partial e_{k-1}}{\partial f_2} - \ldots - f_{n_f} \cdot \frac{\partial e_{k-n_f}}{\partial f_2} \\ \vdots \\ \frac{\partial e_k}{\partial b_1} = -u_{k-n_p-1} - f_1 \cdot \frac{\partial e_{k-1}}{\partial b_1} - \ldots - f_{n_f} \cdot \frac{\partial e_{k-n_f}}{\partial b_1} \\ \vdots \\ \frac{\partial e_k}{\partial b_{n_b}} = -u_{k-n_p-n_b} - f_1 \cdot \frac{\partial e_{k-1}}{\partial b_{n_b}} - \ldots - f_{n_f} \cdot \frac{\partial e_{k-n_f}}{\partial b_{n_b}} \end{cases} \tag{14}$$

Remark 1.1 *In order to implement the above dynamical system for the calculation of e, we define the following matrices*

$$\begin{aligned} L_k &= \begin{bmatrix} y_{k-1} & \dots & y_{k-n_f} & -u_{k-1-n_p} & \dots & -u_{k-n_b-n_p} \end{bmatrix}^T \\ Le_k &= \begin{bmatrix} -e_{k-1} & \dots & -e_{k-n_f} & 0 & \dots & 0 \end{bmatrix}^T \\ dE_k &= \begin{bmatrix} \frac{\partial e_{k-1}}{\partial \theta} & \frac{\partial e_{k-2}}{\partial \theta} & \dots & \frac{\partial e_{k-n_f}}{\partial \theta} \end{bmatrix} \\ \theta_f &= \begin{bmatrix} f_1 \\ \vdots \\ f_{n_f} \end{bmatrix} \quad \theta_b = \begin{bmatrix} b_1 \\ \vdots \\ b_{n_b} \end{bmatrix} \end{aligned} \tag{15}$$

Let x_j the the j'th element from the vector θ, then

$$\frac{\partial e_k}{\partial x_j} = L_k(j) + Le_k(j) - dE_k(j,:) \cdot \theta_f \iff \frac{\partial e_k}{\partial \theta} = L_k + Le_k - dE_k \cdot \theta_f \tag{16}$$

where $L_k(j)$ denotes the j'th element from the vector L_k and $dE_k(j,:)$ denotes the j'th column from matrix dE_k.

Before implementing the algorithm, one must answer the following question: how does such an algorithm know when to stop? Well, as you might have noticed each iteration is indexed (by the letter 'l'), so one stopping method is to make at most M iterations. The value of M is chosen by you, based on trial and error ... pretty much. It will however, be indicated by the lab assistant. However, another way of stopping the algorithm is to verify the norm $\|\theta_{l+1} - \theta_l\|$. When this is smaller than a chosen tolerance (chosen again, by trial and error, and indicated by the laboratory assistant), we decide that the 'improvements' should stop. In this laboratory, we will use both, as we do not have a proof that the second method will ever be met, but this is the preferred one, since it actually shows that a (local) solution was found.

2 Implementation

We give the following $MATLAB^{®}$ code for the implemetation of the above algorithm:

```
% offline oe algorithm

% clear console, workspace and close figures
clc;
clear all
close all

% load data
data = load('lab6_1.mat');
u = data.id.u;
y = data.id.y;

% plot data
plot(u);
hold on;
plot(y,'r');
title('input data');
```

```
legend('u', 'y');
xlabel('time [s]')

% choose orders
nf = 2;
nb = 2;
np = 0;

% choose maximum number of iterations
tmax = 200;

% initialize initial guess of parameters
theta = ones(nf+nb,1)/(nf+nb);
% theta = rand(nf+nb,1);      % alternative rand init

% choos step scalar and accuracy
alpha = 0.1;
threshold = 1e-5;

% pad with zeros to account for negative indexes
y_ = [zeros(nf+nb+np,1);y];
u_ = [zeros(nf+nb+np,1);u];

% miscelanous initializations
stop = 0;
t = 1;
N = length(y);
while (t < tmax) && (stop < 1)
    t = t + 1;

    % collect actual parameters (found so far)
    th_f = theta(1:nf);
    th_b = theta(nf+1:end);

%    compute error and derivatives, together with dJ/d_theta
%    and (Gauss approximation of) d2J/d_theta^2

%   initialize
    dJ = zeros(nf+nb, 1);
    H = zeros(nf+nb, nf+nb);

    % init 'e' vector
    e_ = zeros(nf+nb+np,1); % for now
    dE = zeros(nf+nb,nf);

%   compute e
    for i = nf+nb+np+1:N+nf+nb+np
        e_(i) = y_(i) + y_(i-1:-1:i-nf)'*th_f - ...
                u_(i-1-np:-1:i-nb-np)'*th_b - ...
                e_(i-1:-1:i-nf)'*th_f;
    end
```

```
%   compute de_k, de_{k-1}, ..., de_{k-na}
    for i = nf+nb+np+1:N+nf+nb+np
        L = [y_(i-1:-1:i-nf); -u_(i-1-np:-1:i-nb-np)];
        Le = [-e_(i-1:-1:i-nf); zeros(nb,1)];

        de = L + Le - dE*th_f;

        dE = [de dE(:,1:nf-1)];

        dJ = dJ + e_(i)*de;
        H = H + de*de';
    end

    % apply Gauss-Newton formula
    theta_ = theta;
    theta = theta - alpha*inv(H)*dJ;
    if norm(theta - theta_) < threshold
        stop = 1;
    end
end

% retrieve near-optimal parameters
F = [1 theta(1:nf)'];
B = [0 theta(nf+1:end)'];

% create model
oemodel = idpoly(1,B,1,1,F,0,data.id.Ts);
 figure
compare(oemodel, data.val);

% MATLAB solution
figure
moe = oe(data.id,[nb,nf,np+1]);
compare(moe,data.val);
```

3 Results

Upon execution, the above script produces the following figures:

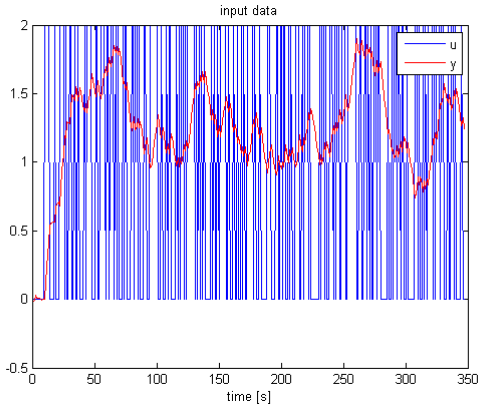

Figure 1: Identification Data

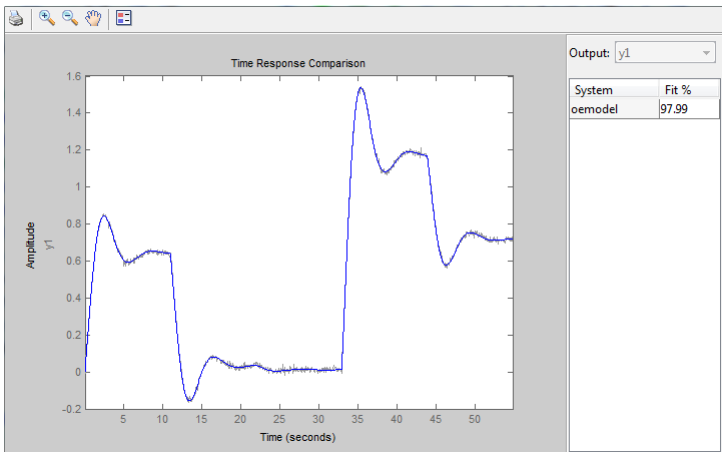

Figure 2: The presented code

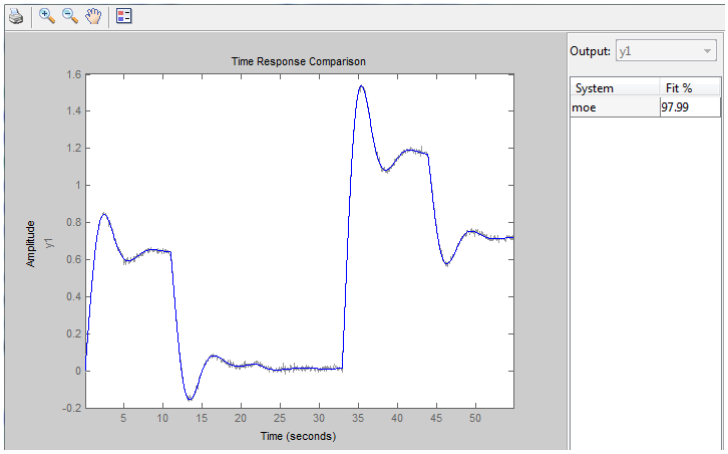

Figure 3: The $MATLAB^{®}$ solution