NON LINEAR CONTROL AND AEROSPACE APPLICATIONS

Libraries documentation

Simone Carletti

A.Y. 2023/2024

Disclaimer

These notes are unofficial and have not been reviewed nor approved by the professor. They are based on my personal understanding and interpretation of the material. As such, they are provided "as is" and may contain errors, omissions, or misinterpretations.

Contents

Part I

lib_fl

1.1 io_fl

$$\begin{cases} \dot{x} &= f(x) + g(x)u \\ y &= h(x) \end{cases}$$

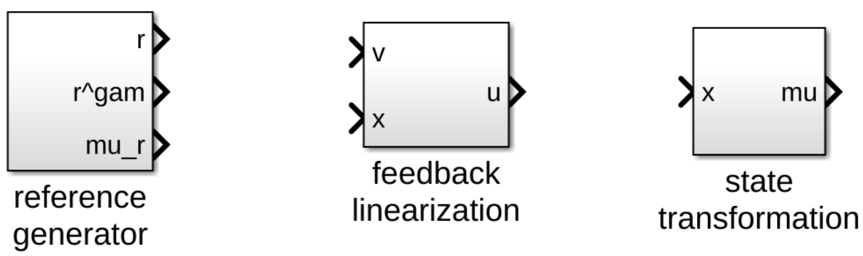
```
[u, mu, ga, a, b, MU] = io_fl(f, g, h, name)

% Inputs:
% f,g,h: functions defining the system
% name (optional): suffix of the Matlab files that will be
%                  generated (u_<name>, mu_<name>)

% Outputs:
% u: feedback linearization control law
% mu: state transformation
% ga: relative degree
% MU: augmented state transformation (MU(1:r)=mu)
% a,b: functions of the normal form
```

If the `name` argument is given calling this function will create 2 files (`u_<name>.m`, `mu_<name>.m`) with the symbolic expression for u and μ.

1.2 ref_gen

```
yr = ref_gen(r, ff, name)

% r: the relative degree (gamma)
% ff: a function that describes the reference (will be
%     evaluated with eval())
% name: suffix of the Matlab files that will be
%       generated (ref_<name>)
%
% Example:
%   yr = ref_gen(2, '1', 'example')  % step input => ff = '1'
```

1.3 fl_blocks.slx



N.B. By default they point to matlab functions with `name = "chua"` (`ref_chua()`, `u_chua()`, `mu_chua()`). **Remember to change the names**.

Part II

lib_rotations

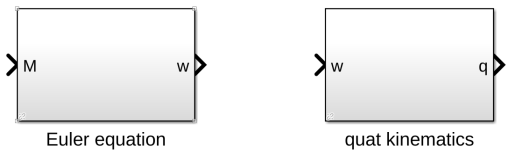
2.1 `rot_mat(order, angles)`

```
R = rot_mat(order, angles)

% Examples:
%   A = rot_mat([1 2 3], [phi theta psi])
%   B = rot_mat([2 2 1], [phi theta psi])
```

2.2 Dynamics and kinematics

2.2.1 `att_kin_dyn_2017b.slx`

SIMULINK blocks for **Euler equations** and **quaternion kinematics**.



Required paramters:

```
J = ...                  % Inertia matrix
IJ = inv(J);             % Inverse of the inertia matrix

q0 = [1 0 0 0];          % initial quaternion q_0
w0 = [0 0 0];            % omega_0
```

The input to the 1^{st} block (Euler equation) is M = external torques, that is exactly what our control gives us (we'll connect `u --> M`).

2.2.2 `x_dot = dyn_kin_quat(t,x)`

```
x_dot = dyn_kin_quat(t,x)
% x = [q; omega];
```

2.2.3 `x_dot = dyn_kin_321(t,x,u,lin)`

```
x_dot = dyn_kin_321(t,x,u,lin)
```

2.3 Kinematic equations

2.3.1 `q_dot = kin_quat(t,q)`

Does:

$$\dot{\mathrm{q}} = \frac{1}{2}\boldsymbol{Q}\boldsymbol{\omega}(t)$$

N.B. the vector $\boldsymbol{\omega}(t) = (\omega_1(t), \omega_2(t), \omega_3(t))$ is **hardcoded** inside the function.
Also it automatically launches `animation_rot(hr,q)` where `hr` is a global variable with the content of `cube.mat`.

2.3.2 `x_dot = kin_tb321(t,x)`

Does:

$$\dot{\boldsymbol{x}} \equiv \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = T\boldsymbol{\omega}(t)$$

N.B. the vector $\boldsymbol{\omega}(t) = (\omega_1(t), \omega_2(t), \omega_3(t))$ is hardcoded inside the function.

2.4 Euler equation (dynamics)

$$\boldsymbol{J}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \boldsymbol{J}\boldsymbol{\omega} = \boldsymbol{M}$$

```
omega_dot = euler_eq(t, omega)
```

N.B. it has $\boldsymbol{J}$ **hardcoded** inside the function.

2.5 Quaternion functions

2.5.1 `[dq, dq_vec] = quat_error(q_ref, q)`

Calculates

$$\tilde{\mathfrak{q}} = (\tilde{q}_0,\ \tilde{\boldsymbol{q}}) = \mathfrak{q}^{-1} \otimes \mathfrak{q}_r = \mathfrak{q}^* \otimes \mathfrak{q}_r$$

where

$$\tilde{\mathfrak{q}} = \texttt{dq} \qquad \tilde{\boldsymbol{q}} = \texttt{dq\_vec} = \textit{complex part}$$

2.5.2 `r = quatprod(q, p)`

Calculates the quaternion product

$$q \otimes p$$

2.5.3 `ele_quat`

Useful also for converting euler angles to a quaternion.

```
[q, p] = ele_quat(I, ang)

% Input:
% I = order of composition ([1 2 3], [3 1 3], ...)
% ang = euler parameters angles (phi, theta, psi)

% Output:
% q = the columns are the 3 elementary quaternions
% p = the resulting quaternion of the composition (q1*q2*q3)
```

2.5.4 vec_rot_quat

Rotates a vector given by the given quaternion. To apply the inverse rotation simply pass $\mathfrak{q}^*$ instead of $\mathfrak{q}$.

```
r2 = vec_rot_quat(q, r1)

% Input:
%       q: quaternion to apply for the rotatoin
%       r1: vector(s) to rotate (3xn)
%
% Output:
%       r2: rotated vector(s) (3xn)
```

2.5.5 omega = quad2omega(q,q_dot)

Calculates $\boldsymbol{\omega}$ from the quaternion using the inverse of the kinematic euqation.

$$\dot{\mathfrak{q}} = \frac{1}{2}\boldsymbol{Q}\boldsymbol{\omega}(t) \implies \boldsymbol{\omega}(t) = 2\boldsymbol{Q}^{-1}\dot{\mathfrak{q}}$$

2.5.6 qua2euler(q)

Allow us to make the conversion from a quaternion to the euler angles associated to 321 rotation.

```
ea = qua2euler(q)
```

2.5.7 qua2dcm(q)

Allow us to convert a rotation described by a quaternion into a rotation described by a Direction Cosine Matrices (DCM).

```
T = qua2dcm(q)
```

2.5.8 dcm2qua(T)

```
q = dcm2qua(T)
```

2.5.9 qua2axes(q)

It allow us to convert a rotation described by a quaternion into a rotation described by an angle-axis representation

```
ax = qua2axes(q)
```

2.5.10 axes2qua(a, A)

Given 2 reference frames `a` and `A` it returns the quaternion that describes how much one is rotated with respect to the other (both 3x3 matrices).

```
q = axes2qua(a, A)
```

2.6 Angle-axis

2.6.1 vec_rot_ax

Rotates one or more vectors using the angle-axis representation.

```
[v2,q] = vec_rot_ax(ax, ang, v1)

% Input:
%       ax: axes of rotation (3x1 matrix)
%       ang: angle in radiants (scalar)
%       v1: vector(s) to be rotated (3xn)
%
% Output:
%       v2: rotated vector(s) (3xn)
%       q:  quaternion that represent the associateted
   rotation
%               to the angle-axis rotation given in input
```

2.6.2 axes2dcm(a0, A)

Given 2 reference frames a0 and A it returns the DCM that describes how much one is rotated with respect to the other (both 3x3 matrices).

```
T = axes2dcm(a0, A)
```

2.7 Plotting

2.7.1 plot_axes

```
plot_axes(OR, a, cc, sw, ta)

% OR: origin of the 3D-plane
% a: for each column specify the end of the corresponding
%    axis [x,y,z] (3x3 matrix)
% cc: color of the axes (string)
% sw: arrow width
% ta: not mandatory
```

2.7.2 mArrow

```
mArrow(p1, p2)

% syntax:    h = mArrow3(p1,p2)
%            h = mArrow3(p1,p2,'propertyName',propertyValue,...)
%
% with:      p1:         starting point
%            p2:         end point
%            properties: 'color':      color of the arrow
%                        'stemWidth':  width of the line
%                        'tipWidth':   width of the cone
%                        'facealpha':  transparency
```

```
% example1:
h = mArrow3([0 0 0],[1 1 1])

% example2:
h = mArrow3([0 0 0],[1 1 1],'color','red','stemWidth',0.02,'
   facealpha',0.5)
```

2.7.3 circle(x,y,r,col,lw)

```
circle(x,y,r,col,lw)

% x:   origin x
% y:   origin y
% r:   radius
% col: color
% lw:  line width
```

2.7.4 animation_rot(hr, q)

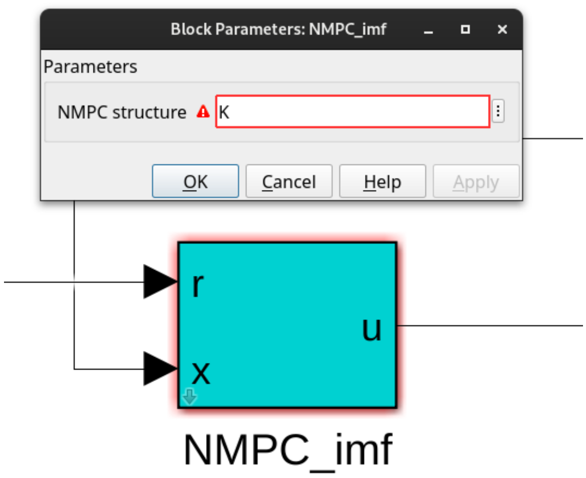
```
animation_rot(hr, q)

% hr:  content of the file cube.mat
% q:   quaternion
```

Part III

lib_nmpc

3.1 NMPC Simulink block

```
% Constraints flag:
%    - 0 = "no constraints" (default)
%    - 1 = "with constraints"
par.nlc = ...

% Prediction model order (number of entries of x)
par.n = ...

% Sampling time and prediction horizon
par.Ts = ...
par.Tp = ...

% Weigth matrices (P,Q = Ny x Ny, R = Nu x Nu)
par.P = ...
par.Q = ...
par.R = ...

% Command input lower and upper bounds (Nu x 1)
par.lb = ...
par.ub = ...

% Time at which the NMPC controller is switched on (default = 0)
par.Tstart = 0;

% Matrix with the tolerances
par.tol = ...

% Name of the matlab file/functions (by default set as below)
% par.model = @pred_model
% parm.nlc = @nlcon

K = nmpc_design_st2(par);
```



3.1.1 Prediction model

The prediction model must be defined in the file `pred_model.m` (if not specified otherwise). In the prof's template `xdot` $\equiv$ `f`, `y` $\equiv$ `h`.

```
function [xdot, y] = pred_model(t,x,u)
  % t: time (scalar). Useful only in the case of time-varying
  %    system.
  % x: state of the system (dimension n*1).
  % u: input of the system (dimension nu*1).

  % Example
  A = ...
  xdot = A*x + [ 0; 0; 0; u ];
  y = x;
```

3.1.2 Constraints

The constrains must be defined in the file `nlcon.m` (if not specified otherwise). Remember to set `par.nlc = 1`.
The various constraints must be written in the **standard form**:

$$\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{y}) \leq \boldsymbol{0}$$

Example:

$$\|x\| \geq R \implies R - \|x\| \leq 0$$

(this example is also what shown below in the the matlab code)

```
function F = nlcon(x,y)
  % x: state of the system (matrix of dimension nx*N)
  % y: output of the system (matrix of dimension ny*N)

  % N is the number of samples in the time interval [t,t+Tp], and
  % it's automatically chosen by the NMPC solver.

  % Initialization  (Nc = number of constraints)
  N = size(x, 2);
  Nc = 1;
  F = zeros(Nc, N);

  % Constraint function (Nc*N matrix)
  F(1,:) = 10 - vecnorm(x(1:3,:));
```

p.s. N is the number of samples in the time interval $[t, t + Tp]$, and it's automatically chosen by the NMPC solver.

Part IV

lib_aerospace

4.1 `fr2b(t,x,MU)`

`x = [r; v]`. Internally it's written as

$$\dot{x}_{1:3} = x_{4:6}$$
$$\dot{x}_{4:6} = \frac{-\mu \cdot x_{1:3}}{|x_{1:3}|^3}$$

with $x_0 =$ `zeros(6,1)`.

4.2 `rv2oe`, `oe2rv`

$$\boldsymbol{x} = (\boldsymbol{r}, \boldsymbol{v}) \in \mathbb{R}^6$$
$$\boldsymbol{y} = (a, \boldsymbol{e}, ci) \in \mathbb{R}^5$$
$$\boldsymbol{el} = (a, e, i, \Omega_m, \omega_m) \in \mathbb{R}^5$$

where $ci \triangleq cos(i)$. The **vectors are expressed in the GE frame**.

p.s. `x = [pos; vel] = [r; v] = [r; r_dot]`. `rv2oe` also accepts `[r; v; m]` $\in \mathbb{R}^7$.

4.2.1 `rv2oe`: state vector → orbital elements

```
[y, el, th] = rv2oe(x, mu)

% Inputs:
% x: state vector [r; v]
% mu (optional): if not given it's going to use "global MU"

% Outputs:
% y: orbital elements used for NMPC (a, e, ci)
% el: "classical" Keplerian orbital elements
% th: true anomaly
```

4.2.2 `oe2rv`: orbital elements → state vector

```
[r, v, p, T] = oe2rv(y, th, mu)

% Outputs
% r: position
% v: velocity
% p: semilatus rectum
% T: trasformation matrix PF frame -> GE frame
%    (do "T^-1 * r" to go back to PF frame, same for v)
```

4.3 spacecraft_dynamics

```
x_dot = spacecraft_dynamics(t, x, u, d)

% Note: the input "t" is unused
% x = [r; v; m] (7x1 vector);
% u,d 3x1 vectors
```

It requires **GLOBAL matlab variables** (and so it's possible to use it only inside an interpreted matlab function):

```
global MU RE ve

% MU = standard gravitational parameter
% RE = radius of the Earth
% ve = engine exhaust velocity
```

And also it has the spacecraft body mass **hardcoded** as $m_b = 4000kg$.

4.4 Plotting and animation

```
orbit_animation(x, ang, L);

% x:    [x, y, z] position
% ang:  view angles [AZ,EL]  (passing to view())
% L:    +- limits for the axes
```

```
% Example
% out.x = Nx7 array (with N = number of samples)

for i=1:length(out.x)
    orbit_animation(out.x(i, :), [70, 40], Inf)
end
```