MATLAB FOR THE EAGLE PROJECT

PANTELIS SOPASAKIS AND PANOS PATRINOS

Abstract. This is a brief document describing certain MATLAB functions which are particularly useful for the EAGLE project. We provide an overview of some basic MATLAB functions (e.g., `rank`, `eig`, etc), we explain how to create high quality graphics, next, we move one to functions which are used to design controllers and observers and we present some useful functions to work with quaternions — a recent addition to MATLAB. We dedicate a few pages on MATLAB classes which help us structure our code; something of high value in large projects. We explain the basics of MEX interfaces that allow us to run C code from MATLAB and we discuss how to go about code generation. The last section is about interfacing MATLAB with the quadcopter and what tools we may use for that purpose.

Contents

1. Coding in MATLAB

1.1. **Good practices.** There are certain well-established good practices in software development which are essential for this project as it is a collaborative project, it involves several layers of abstraction (acquisition of measurements, filtering, attitude control, altitude control, navigation and many more) and it is likely that you will need to experiment with alternative approaches.

We recommend going through this Wikipedia article on "Best coding practices." More specific to the EAGLE project, we would highlight the following points (not in order of importance):

(1) Use a private source code versioning system (preferably based on git) and actively collaborate using it. Add README files, make branches when necessary, write meaningful commit messages. Do not forget to add a `.gitingore` file and do not track binary files.

(P. Sopasakis and P. Patrinos) KU Leuven, Department of Electrical Engineering (ESAT), STADIUS Center for Dynamical Systems, Signal Processing and Data Analytics & Optimization in Engineering (OPTEC), Kasteelpark Arenberg 10, 3001 Leuven, Belgium. Email addresses: pantelis.sopasakis@kuleuven.be and panos.patrinos@esat.kuleuven.be.

Lecture Notes for the 3rd-year BSc project "EAGLE" at KU Leuven. version 1.1.1. Last updated: November 7, 2017.

(2) Document your functions properly and clearly. If you change the function template (inputs/outputs), update the documentation too.
(3) Write comments as you code.
(4) Do not comment-out code (use git).
(5) Write unit tests — for the MATLAB, C and Python code — and opt for 100% coverage. Write *functional tests* for your module and *integration tests* for the whole project. Do not test using print statements.
(6) Do not use globals (e.g., the `global` keyword in MATLAB). Use functions and classes instead. Similarly, in `C`, do not use global variables (use `static` file-scope variables).
(7) Establish a meaningful naming convention (and make sure all team members abide by it).
(8) Before you use any third-party functionality, read the documentation and make sure you understand how it works. For example, in MATLAB, not spending 30 minutes time to study the documentation of `ss` and `lqr` or `kalman` may delay the progress of the project for weeks.
(9) Establish your coding conventions before you begin coding. Take some time with your team member and write down what functionality you need to implement, sketch out a workflow, identify the entities of your software and — if you work with classes — sketch a UML.
(10) Create milestones and use an issue tracker to know at every moment what needs to be done and who is responsible for what.

You may also further read the articles MATLAB programming style guidelines and Best Practices for Scientific Computing. Part of the grading will be based on your adherence to good development practices.

Note. Simulink does not combine well with code versioning (git), this is why it should not be used. Additionally, prefer functions over scripts. Before you start coding, read this document and sketch out the basic structure of your implementation.

1.2. **General purpose functionality.** Let us start by an overview of certain core functionality of MATLAB. This is a list of basic MATLAB functions that you need to be aware of before we proceed:

(1) `ones`, `zeros`, `eye`: generates matrices with ones, the zero matrix and the identity matrix of specified dimensions. For example `zeros(4,5);` will generate a 4×5 zero matrix.
(2) `eig`: returns an array with the eigenvalues of a matrix. When called with two output arguments it also returns its eigenvectors. See also the similar function `eigs`.
(3) `rank`: returns the rank of a matrix. The second input argument is a tolerance. When used with two input arguments it returns the number of singular values of the given matrix which are above the specified tolerance.
(4) Generating random numbers: `randn(n,1)` generates a vector of normally distributed random numbers which are uncorrelated. Use `mvnrnd(mu, sigma)` to generate a random sample from the vector-valued distribution $\mathcal{N}(\mu, \Sigma)$.
(5) The backslash operator (\) is used to solve linear systems. When we need to solve a linear system of the form $Ax = b$ where $x = A^{-1}b$ (assuming that A is invertible), it is very bad practice to run `x = inv(A)*b;`. MATLAB will indeed give a warning. The reason is that not only is `inv` computationally expensive, but it is also numerically unstable and may lead to bad results. Besides, $Ax = b$ may have solutions without A being invertible or even square. Instead, we should be using the backslash operator: `x=A\b`.
(6) `find`: finds all elements in a vector or matrix which satisfy a condition, e.g., `find(A>0)` returns an array of indices `i1, i2, ...` in `A` so that `A(i1), A(i2), ...` are positive.
(7) `save` saves the current workspace in a binary `.mat` file that can be later retrieved with `load`. You may select which variables to store (instead of saving the whole workspace). Binary files are typically not added in code versioning systems, but it's wise to back them up somewhere.
(8) Use `help` and `doc` to access the documentation of a MATLAB function. By writing comments in the beginning of your function implementations, you create documentation that is accessible via `help`.
(9) `csvwrite/csvread`: read/write from/to CSV files.
(10) `savejson/loadjson`: saves structures to JSON files and loads data from JSON files. Certain restrictions apply; e.g., it is not possible to store structures with function handles or anonymous functions. These functions are part of jsonlab which is not part of MATLAB's distribution.

Function *handles* are data types which "store associations to functions," are it is reported in the official documentation page. The allow functions to be treated as variables, so they can be passed to functions as input arguments, they can be constructed by functions, they can be combined and so on. Several core MATLAB functions such as `ode45`, which we use in Section 1.5, or the popular `fmincon` and `fminunc` expect function handles as input arguments.

Function handles can be constructed on the fly as follows:

```
% Constuct the function f(x) = sin(x):
f = @(x) sin(x);

% We may then call f:
y = f(0.1);

% We may constuct function handles with two arguments
g = @(x,y) cos(x + y.^2);
z = g([1;2], [-2;0]);
```

These are known as *anonymous functions*. But we may also create a handle from any MATLAB function. This is similar to creating an *alias* for a function:

```
myrand = @randn;
y = myrand(3,4);  % calls `randn`.
```

Note that although we may create copies of function handles (`myrand2 = myrand`), handle arithmetic is limited. For instance, we may not do `fun = f1 + f2;` for two function handles `f1` and `f2`. We may, however, do `fun = @(x) f1(x) + f2(x);`.

Function handles can be passed to other functions. For instance, MATLAB's *integral* is a function that computes the definite integral $\int_a^b f(x)\mathrm{d}x$ for a function f which is passed as a handle. Here is an example:

```
% Create a handle from `sin`
f = @sin;
% Compute the integral of `sin(x)` from 0 to 2*pi
% The result is practically 0.
intF = integral(f, 0, 2*pi);

% But we may also create an anonymous function and
% pass it to `integral`:
g = @(s) sin(2*s) + cos(s);
intG = integral(g, 0, 1);
```

A known bug: When running MATLAB on laptops in Linux you might see the message `MEvent. CASE!` in the command window as you use the touchpad to scroll. You may disable it using `!synclient HorizTwoFingerScroll=0`.

1.3. **The Symbolic Toolbox.** The symbolic toolbox of MATLAB can be used to compute Jacobians and determine the linearisation of a nonlinear system very easily:

```
% Define the symbols
x = sym('x', [3, 1], 'real'); % 3 states
u = sym('u', [2, 1], 'real'); % 2 inputs

% Define the nonlinear system dynamics
f = [  sin(x(1))*cos(x(2)) + (1+x(1))*u(1);
       sin(x(2))^2 + x(2)*u(2) + u(1)
       x(3) + 2*cos(x(2))*x(1) + u(2)       ];

% Make sure that the origin is an equilibrium point
f00 = double(subs(f, [x;u], zeros(5,1)));
assert(all(f00==0), 'the origin is not an equilibrium point');

% Compute the Jacobian of f with respect to x and u:
Jfx = jacobian(f,x);
Jfu = jacobian(f,u);

% Find the linearization of the system at the origin:
A = double(subs(Jfx, [x;u], zeros(5,1)));
B = double(subs(Jfu, [x;u], zeros(5,1)));
```

Here we used `jacobian` to determine a symbolic Jacobian matrix of a nonlinear function and we computed the values $\mathrm{J}_x f(0,0)$ and $\mathrm{J}_u f(0,0)$ using `subs`. Note that the result of `subs` is still a symbolic value. This is why we apply `double` on it to get a numeric value.

Symbolic functions can be cast as function handles (anonymous functions) using `matlabFunction`. It is often convenient to pass the optional parameter `Vars` to specify the exact order of variables that the function handle should expect. We use `matlabFunction` in Section 1.6, but here is a short example:

```
% System dimensions
nx = 2; % number of states
nu = 1; % number of inputs

% Define the symbols `x` and `u` for the state and input
% variables
x = sym('x', [nx, 1], 'real');
u = sym('u', 'real');

% Define the system dynamics:
% f = [ x1^2 + x1 + u + 2*x2
%       x2^2 + x2/10 - x1 ]
A = [1 2; -1 0.1];
f = A*x + x.^2 + [1;0]*u;

% Construct a function handle from `f` where the first
% input argument is the state `x` and the second it the
% input `u`
fHandle = matlabFunction(f, 'Vars', {x, u});

% Now we may call fHandle as follows:
xDot = fHandle([.1;-.2], 0.1);

% We may further create a MATLAB function from `f` and save
% it to a file ('functDynamics.m')
matlabFunction(f, 'File', 'functDynamics', ...
    'Vars', {x, u}, 'Outputs', {'dyn'});
```

1.4. **Plotting.** The command `plot` is used to create plots in MATLAB. It is good practice to plot setting `linewidth` to 2 or 3 for the result to look clearer. Using a *grid* allows us to understand the plots better (use `grid on`;). Add axis labels using `xlabel('my axis label [unit]');` and `ylabel('my y axis [unit]');`. Increase the font size using `set(0,'DefaultAxesFontSize',12);`. If you need to overlay multiple lines use `hold on;`. This is useful when, for example, you need to plot the actual system states x_k and their estimates $\hat{x}_k$ in the same plot. Once no more data needs to be added to the plot, switch off the hold mode using `hold off;`. Use `axis tight;` for the axes to be tight, or use the command `axis([XMIN XMAX YMIN YMAX])` to specify the axes' limits.

Legends can be added with the command `legend`, e.g., `legend('x', 'y');` will clear a legend with two entries. Furthermore, `legend` accepts the option `'Location'` that allows us to specify the position of the legend (e.g., `SouthEast`, `NorthEast`, etc).

The MATLAB documentation[1] explains how to modify the line colors, style (solid, dashed, etc), add markers and a lot more. For example `plot(x, y, '-.ob');` plots `y` versus `x` using a dash-dot line (`-.`), places circular markers (`o`) at the data points, and colors both line and marker blue (`b`).

If you need to create multiple plots there are two main options. One is to use `figure;` which start a new figure. You can then plot your data, use `hold on;` to plot multiple data in the same figure and so on. Then, call `figure;` again to create a new figure. You may assign numerical IDs to the figures — e.g., `figure(101);`, 200, etc.

If you need to export a figure from MATLAB you have several options. You can get high quality graphics (that you can use to make posters or reports) by going to `File`, then `Export Setup`, `Rendering`, choose `painters (vector format)` next to `Custom renderer` (check the box), the `Export` and choose `EPS` under `Files of Type`. Choose a filename and save the figure in EPS format. Save the figure in `.fig` format too.

The second option is to use matlab2tikz and export the figure in Tikz format and include it in your LaTeX code. This way you can create high-quality customisable graphics for your posters and reports.

You may have multiple plots in the same figure using `subplot`. There exist also various types of plots. You might want to have a look at `stairs`.

[1] Read more about the line specification and colour specification.

1.5. **Simulating.** We need to simulate a parametric dynamical system (in our case, the attitude and translational dynamics of a quadcopter). The first step in that direction is to create a structure with the system parameters. In fact, it is wise to make a function that returns the quadcopter parameters (its mass, moments of inertia, thrust coefficients, etc) together with a *perturbation factor* (or, more precisely, *perturbation factors*). Let us have a look at a very simple example where the system dynamics is described by:

$$\dot{x}_1 = a\cos(x_2)x_1 + bu$$
$$\dot{x}_2 = cx_2^2 - x_1^3,$$

where a, b and c are some parameters with nominal values $a = 1$, $b = 0.5$ and $c = 0.1$. We construct a MATLAB function which returns a structure `p` as follows

```
function p = systemParameters(f)
%SYSTEMPARAMETERS returns a structure with the system parameters, possibly
%perturbed by a factor f.
%
%Syntax:
% p = systemParameters()
% p = systemParameters(f)
%
%Input arguments:
% f:    perturbation parameter (optional --- default value: 0)
%
%Output arguments:
% p     structure with the system parameters. It contains the fields
%          - a : explain what this parameter is and specify its units
%                of measurement.
%          - b : ditto
%          - c : ditto
%
%See also:
% systemDynamics
%

if nargin == 0, f = 0; end
p.a = 1 * (1+f*randn);
p.b = 0.5 * (1+f*randn);
p.c = 0.1 * (1+f*randn);
```

Note that several lines are dedicated to explaining the input and output arguments of this function. Documentation is an essential and inextricable part of software development.

The perturbation factor `p` allows us to create random systems (which are “close” to the original system) by perturbing the system parameters by multiplying by `1+f*randn`. This is just an example; the form of the perturbation is application-specific. Next, we define the system dynamics:

```
function xdot = systemDynamics(t,x,u,p)
%SYSTEMDYNAMICS retuns the derivative of x, f(t,x,u) at time t, state x and
%for an input value u. The system parameters are provided also as an input
%argument.
%
%Syntax:
% xdot = systemDynamics(t,x,u)
% xdot = systemDynamics(t,x,u,p)
%
%Input arguments:
% t :     time
% x :     state
% u :     input
% p :     system parameters
%
%Ouput arguments:
% xdot : value of f(t,x,u)
%
%See also:
```

```
% systemParameters
%

if nargin < 4, p = systemParameters(); end
xdot = [ p.a * cos(x(2))*x(1) + p.b * u;
         p.c * x(2)^2 - x(1)^3];
```

Let us now simulate the above dynamical starting from an initial state $x(0) = x_0$ and using a control function $u(t, x)$.

In order to simulate the system dynamics in continuous time, we use `ode45`; a solver for non-stiff differential equations. We need to specify the closed-loop function $f(t, x, u) = f(t, x, u(t, x))$ as a function handle with two arguments: `t` and `x` (type `help ode45` for details). Here, we assume that $u(t, x) = \sin(t)$ and we simulate the system

$$\begin{aligned}\dot{x}_1 &= a\cos(x_2)x_1 + b\sin(t),\\ \dot{x}_2 &= cx_2^2 - x_1^3.\end{aligned}$$

```
% Initial condition x(t0) = x0
t0 = 0;
x0 = [1;-1];

% Input signal
u = @(t,x) sin(t);

% The system dynamics is described by
sysPars = systemParameters();
functDynamics = @(t,x) systemDynamics(t, x, u(t,x), sysPars);

% Simulate the system using ode45
tmax = 5;
[t,x] = ode45(@(t,x) functDynamics(t,x), [t0 tmax], x0);

% Plot the system trajectory
plot(t, x, 'linewidth', 2);
grid on;
xlabel('Time');
ylabel('State');
```

1.6. **Systems and control.** MATLAB offers several functions that facilitate controller and observer design including discretisation routines and LQR/KF design. We first need to define the system dynamics in state space form. We typically begin by defining the continuous-time linearised dynamics of the form

$$\begin{aligned}\tfrac{\mathrm{d}}{\mathrm{d}t}x &= Ax + Bu\\ y &= Cx + Du.\end{aligned}$$

Let us give an example where we define a random continuous-time system

```
nx = 4;   % number of states
nu = 3;   % number of inputs
ny = 2;   % number of outputs

A = randn(nx, nx);
B = randn(nx, nu);
C = randn(ny, nx);
D = randn(ny, nu);

systemContinuous = ss(A, B, C, D);
```

We may now discretise the continuous-time system and obtain the discrete-time representation

$$\begin{aligned}x_{k+1} &= A_d x_k + B_d u_k,\\ y_k &= C_d x_k + D_d u_k,\end{aligned}$$

for a given sampling period $T_s > 0$. For that, we use MATLAB's `c2d`. Here is an example of use where discretise the above continuous system with sampling rate 200Hz, that is, sampling period $T_s = 1/200$s.

```
Ts = 1/200;
systemDiscrete = c2d(systemContinuous , Ts, 'zoh');
Ad = systemDiscrete.A;
Bd = systemDiscrete.B;
Cd = systemDiscrete.C;
Dd = systemDiscrete.D;
```

The last argument means that we the discretisation is based on a zero-order hold element. The sampling time is also stored in `systemDiscrete.Ts`.

If we want to define a *discrete-time* system in the first place, then, we must use `ss` with `-1` as a fifth argument, that is, `systemDiscrete = ss(Ad, Bd, Cd, Dd, -1);`. This is useful when designing Kalman fitlers using `kalman`.

Moreover, `ss` allows to specify the names of the state, input and output variable. Read the documentation for details (type `help ss` or `doc ss`).

The controllability and observability matrices of a system are returned by `ctrb` and `obsv` respectively. Here is an example where we check whether a given discrete-time system is controllable:

```
% For given matrices Ad, Bd, Cd, Dd:
systemDiscrete = ss(Ad, Bd, Cd, Dd, -1);
ctrbSystemDiscrete = ctrb(systemDiscrete);
% Assert that the system is controllable
assert ( rank(ctrbSystemDiscrete, 1e-6) == size(Ad, 1), ...
  'Error: The system is not controllable');

% The same can be achieved using:
ctrbSystemDiscrete = ctrb(Ad, Bd);
```

To design an LQR controller for a discrete-time system with matrices `(A, B, C, D)` we may use `dlqr`. Here is an example of use:

```
% System data
A = [1 1; 0 0.5];
B = [0;1];
C = [1 0.1];
D = 0.1;

% System dimensions
nx = length(A);    % number of states
nu = size(B,2);    % number of inputs
ny = size(C,1);    % number of outputs

% First check whether the system is controllable:
if rank(ctrb(A,B),1e-6)~=nx,
  error('System not controllable');
end

% Define the weight matrices Q and R
Q = eye(nx);
R = 2*eye(nu);
Kcontroller = -dlqr(A,B,Q,R,0);

% Assert that Kcontroller is a stabilising gain
assert( all(abs(eig(A+B*Kcontroller)) <= 1-1e-7),...
  'K is not stabilising');
```

It is important to underline that `dlqr` returns a matrix K so that the feedback law $u(x) = -Kx$ solve the LQR optimisation problem (not $u(x) = Kx$). This is why there is a minus in the code above. We have added an assertion to verify that the eigenvalues of $A + BK$ are indeed within the unit circle.

In fact, `dlqr` returns up to 3 arguments. The second one is the associated positive definite symmetric matrix P which defines the optimal infinite-horizon cost function $J^\star = x'Px$. The third argument is an array of the eigenvalues of $A + BK$ in our notational convention (again, recall that `dlqr` returns $-K$).

MATLAB's `kalman` enables us to design Kalman filters for linear dynamical systems. Suppose we have the discrete-time linear time-invariant system

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k + Gw_k, \\ y_k &= Cx_k + Du_k + Hw_k + v_k, \end{aligned}$$

where w_k and v_k are zero-mean independent random processes which follow the normal distribution with

$$\begin{aligned} \mathbb{E}[w_k w_k'] &= Q, \\ \mathbb{E}[v_k v_k'] &= R, \\ \mathbb{E}[w_k v_k'] &= N. \end{aligned}$$

In order to call `kalman` for the above discrete-time system we need to construct a discrete-time dynamical system with matrices `(A, [B G], C, [D H])`.

Function `kalman` allows to specify which outputs are *deterministic*, that is, known without error.

Let us give a complete example where we design an LQR controller and a KF observer for a continuous-time nonlinear system with 4 states, 1 input and 3 output variables. The system state is $x = (x_1, x_2, x_3, x_4) \in \mathbb{R}^4$. The system dynamics is described in continuous time by

$$\frac{\mathrm{d}}{\mathrm{d}t}x = \begin{bmatrix} x_2 + x_1^2 \\ a(x_3 - x_1) - b(x_4 - x_2) + c\sin(x_2)\sin(x_3) + x_2^2 \\ x_4 + x_3^2 - \sin^2(x_1) \\ u + \frac{a}{10}x_1 + \frac{b}{10}x_2 - \frac{a}{10}x_3 - \frac{b}{10}x_4 \end{bmatrix} =: f(x,u), \tag{1.1}$$

with parameters $a = 2$, $b = 0.5$ and $c = 0.1$. The system output is described by

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} x,$$

that is, we measure x_1 and x_3. We shall describe how noise affects the system state and dynamics in discrete time in what follows.

We start by defining the nonlinear system in MATLAB and linearising it using the symbolic toolbox:

```
% Define the system dimensions
nx = 4; % number of states
nu = 1; % number of inputs
ny = 2; % number of outputs

% Define the system parameters
a = 2;
b = 0.5;
c = 0.1;

% Define the symbols xSymb (states) and uSymb (inputs)
xSymb = sym('x', [nx, 1], 'real');
uSymb = sym('u', [nu, 1], 'real');

% Define the nonlinear dynamics
f = [xSymb(2) + xSymb(1)^2
     a*(xSymb(3)-xSymb(1)) + b*(xSymb(4)-xSymb(2)) + ...
         c*sin(xSymb(2))*sin(xSymb(3)) + xSymb(2)^2
     xSymb(4) + xSymb(3)^2 + sin(xSymb(1))^2
     uSymb + a*xSymb(1)/10 + b*xSymb(2)/10 - a*xSymb(3)/10 - b*xSymb(4)/10];

% Define matrices C and D
C = [1 0 0 0;
     0 0 1 0];
D = [0;0];

% Symbolic Jacobians:
Jfx = jacobian(f, xSymb); % Jacobian with respect to x (xSymb)
Jfu = jacobian(f, uSymb); % Jacobian with respect to u (uSymb)

% Linearise f at the origin:
A = double(subs(Jfx, [xSymb; uSymb], zeros(nx+nu,1)));
B = double(subs(Jfu, [xSymb; uSymb], zeros(nx+nu,1)));
```

It would be good to test whether the pair (A, B) is controllable and the pair (A, C) is observable. We may now discretise the linearised system (with matrices `(A, B, C, D)`) using `c2d` specifying the sampling time T_s.

```
% Define the linearised system using `ss`.
% This is a continuous-time system
linearisedSystem = ss(A, B, C, D);

% Discretise the linearised system:
Ts = 0.4;
ZOH = c2d(linearisedSystem, Ts,'zoh');
[Ad, Bd, Cd, Dd] = ssdata(ZOH);
```

Next, we design an LQR controller for the discretised system with given matrices Q and R:

```
% We choose matrices R and Q to be diagonal. R is a scalar
% since nu = 1 and Q is allowed to have zeros on its diagonal
% (it is only required to be positive semidefinite)
R   = 1;
Q   = diag([1 0.1 1 0.1]);
Kcontroller  = -dlqr(Ad, Bd, Q, R); % mind the minus sign
assert( all(abs(eig(Ad+Bd*Kcontroller)) < 1-1e-6), ...
    'A+BK not stable');
```

We shall now design a Kalman filter for the discrete-time which has the form[2]

$$\begin{aligned} x_{k+1} &= A_d x_k + B_d(u_k + w_k) \\ y_k &= C x_k + v_k. \end{aligned}$$

This formulation means that the control command u_k is sent to an actuator which introduces noise to the system. In the case of the quadcopter, that would be that the ESCs have not been modelled perfectly. The term v_k reflects the measurement error on y_k. Following the notation we introduced above, it is $G = B_d$ and $H = 0$. The observer has the form

$$\hat{x}_{k+1} = A_d \hat{x}_k + L(C\hat{x}_k - y_k) + B_d u_k.$$

```
% We specify the covariances for `v` and `w`
covarSensors   = 0.02^2 * eye(ny);
covarDynamics = 0.0001^2 * eye(nu);

% Next we build the Kalman filter:
kalmanSystem       = ss(Ad, [Bd Bd], C, [D D], -1);
[~, Lobserver]    = kalman(kalmanSystem, covarDynamics, covarSensors, 0);
Lobserver = -Lobserver;

% Let us make sure that A+Lobserver*C is a stable matrix
assert( all(abs(eig(Ad+Lobserver*Cd)) < 1-1e-6), ...
    'A+LC not stable');
```

Last, we simulate the system

```
% First we construct a function handle out of the symbolic function `f`
fun = matlabFunction(f, 'Vars', {xSymb , uSymb});

% We define the initial condition and initial state estimate
x     = [0.5; -0.1; 0.1; -0.1]/10;
xEst  = x;

T = 100; % simulation time

% Initialise caches for the state and its estimates. We will
% store the sequence of states and state estimates in a matrix to
% be able to plot them afterwards. Here, we pre-allocate memory.
xCache  = zeros(nx, T);
xEstCache  = zeros(nx,T);
```

[2]When using the exact linearisation method assuming a zero-order hold element, $C_d = C$ and $D_d = D$. Note also that it is very common to have $D = 0$.

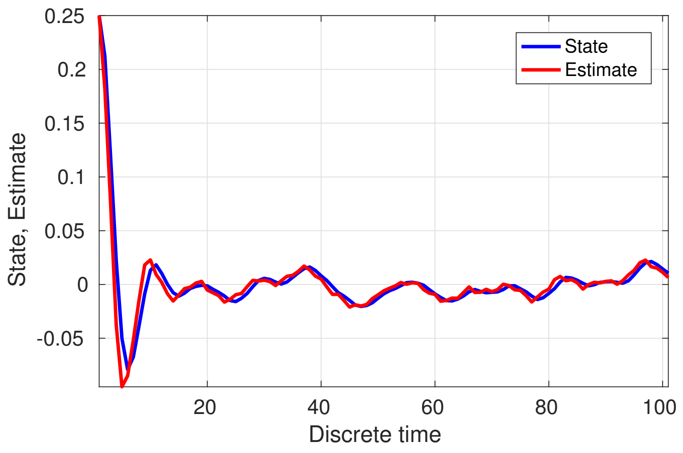

FIGURE 1. Actual and estimated states x_1 and $\hat{x}_1$ for the above Kalman filter. The figure is produced using `plot` with options `linewidth=2` and using `grid on` and `axis tight`. The graphic was exported in EPS format. The legend was added with `legend('State', 'Estimate');`.

```
% Put the initial state and the initial state estimate in the cache.
xCache(:,1) = x;
xEstCache(:,1) = xEst;

% We simulate the system...
for k=1:T,
    % Noises
    w = mvnrnd(0, covarDynamics)';
    v = mvnrnd(zeros(ny,1), covarSensors)';

    % Compute the control action using the estimated state
    u = Kcontroller * xEst;

    % Simulate the nonlinear system using ode45 starting from x
    % and for time Ts. The actuation that is applied to the system
    % is u + w (perturbed by random noise)
    [~,xi] = ode45(@(t,s) fun(s, u + w), [0, Ts], x);
    x = xi(end,:)';

    % Take the system output
    y = Cd * x + Dd * u + v;

    % Update the state estimates
    xEst = Ad * xEst + Lobserver * (Cd * xEst - y) + Bd * u;

    % Store the states in the cache
    xEstCache(:,k+1) = xEst;
    xCache(:,k+1) = x;
end
```

We may now plot the actual states against their estimates as shown in Figure 1. It is, furthermore, interesting to see how (or whether) the controller and the observer still work if the actual system parameters are different from their nominal values (without re-designing the controller/observer). This will be part of your design and simulations for the EAGLE project. For example, you have measured that the thrust coefficient is $\hat{c}_t = 0.1$ and you have design a controller and an observer which — in simulations for the nominal system — seem to work satisfactorily. Will the closed-loop system still work well if the actual value is $c_t = 0.09$? What if the quadcopter is slightly heavier or lighter?

1.7. **Operations with quaternions.** MATLAB offers a collection of functions to perform operations with quaternions:

(1) `quatnorm(q)`: returns $\|q\|$
(2) `quatnormalize(q)`: returns $q/\|q\|$
(3) `quatmultiply(p,q)`: performs $p \otimes q$
(4) `quatdivide`: multiply by inverse quaternion
(5) `quatconj`: conjugate quaternion
(6) `quatinv`: returns the inverse quaternion
(7) `quatrotate`: rotates a vector by (the conjugate of) a quaternion. Note, however, that because MATLAB uses the JPL convention, we need to provide `quatconj(q)` to `quadrotate` instead of `q`.
(8) `atan2`: four-quadrant arctangent. Use this function instead of `atan`.

2. Classes

The object-oriented programming approach is great way to organise our code and provide easy-to-use functionality, clear abstractions and extensibility. MATLAB supports classes and class hierarchy. Classes carry *field* definitions and *methods*, both of which can be either *private*, that is, hidden from the end-user or *public*. Only the functionality that is absolutely necessary for the user should be exported (made public), while the rest should be stored internally.

First, we need to identify the attributes (fields) of our class. These are best to remain hidden (private). The user should not be allowed to access them directly, let alone, modify them. We will see that this prevents the user from updating these attributes with erroneous values.

Let us have a look at a simple example of how classes can be used before we discuss how we can construct them. Instances of classes are called objects and are constructed as `mypet = Dog('Charlie');`: this command will construct an instance of `Dog` which we store in the variable `mypet` whose name is `'Charlie'`. This object has a `name` which is stored internally and cannot be accessed directly (we cannot access `mypet.name`). We need to be able to *get* the name of `mypet`. This should be supported by a function called `getName()`, so `mypet.getName()` will return the string `'Charlie'`. Say we need to change the name of `mypet` to `'Bob'`. We cannot do `mypet.name='Bob'` as if `mypet` were a structure, because the field `name` is hidden (private) and cannot be accessed from outside the class. Changing `name` should be supported by a function of the form `mypet.setName('Bob')`. By doing so, it is now possible to (i) document the behaviour of `getName()` and `setName()` and (ii) throw an exception if the provided name is not acceptable (e.g., it is not a string).

It is probably best to understand how classes work through an example. Let us construct a simple class which implements a linear state observer, that is, a linear time-invariant dynamical system with state $\hat{x}_k$. For classes to have an internal state, they need to be derived from `handle`. Here is an example of a class called `EagleObserver`:

```matlab
classdef EagleObserver < handle
% EAGLEOBSERVER is a class which (...)
% Detailed documentation goes here

    % Private properties (not accessible from outside)
    properties (Access = private)
        xest;  % current state estimate
        Ad;    % Matrix Ad of the discrete-time system
        Bd;    % Matrix Bd of the discrete-time system
        Cd;    % Matrix Cd of the discrete-time system
        Dd;    % Matrix Dd of the discrete-time system
        L;     % Gain of the linear observer
    end % ... end of properties

    methods (Access = private)
        % '''''''''''''''''''''''''''
        % Methods for class use only
        % '''''''''''''''''''''''''''
    end % ... end of private methods

    methods (Access = public)
        % '''''''''''''''''''''''''''
        % Public class methods go here
        %  - Constructor
        %  - Getters and Setters
        %  - Other methods
        % ''''''''''''''''''''''''''''''
    end % ... end of public properties

end % ... end of class definition
```

There are two placeholders where we may add our custom methods. Classes are saved in folder which start with the @ symbol followed by the class name. For instance, this class is saved in `@EagleObserver/EagleObserver.m`.

An essential part of a class is its constructor with which we may create instances of that class. A constructor initialised the attributes of the class from user-specified data. This is how a constructor for `EagleObserver` may look:

```
function o = EagleObserver(Ad, Bd, Cd, Dd, L, xest0)
    %This is the constructor of EagleObserver
    %
    %Provide documentation
    %

    if nargin~=6,
     error('Exactly 6 arguments must be provided');
    end
    % Test whether (Ad, Bd, Cd, Dd) are of compatible sizes
    o.xest = xest0;
    o.Ad = Ad;
    o.Bd = Bd;
    o.Cd = Cd;
    o.Dd = Dd;
    o.L = L;
end
```

A constructor is typically — except for certain special cases — a public method so that users can use it to create instances of the class. Here, users can create instances of `EagleObserver` specifying the system matrices `Ad, Bd, Cd, Dd`, the observer gain `L` and the initial state estimate `xest0`. These data are stored in internal attributes of the class (to which the user does not have access). We may construct objects of `EagleObserver` by `o = EagleObserver(Ad, Bd, Cd, Dd, L, xest0);`. Note that all functions must be terminated with an `end`.

In order to have access to the current state estimate that the `EagleObserver` stores we need a `getter` method:

```
function x = getCurrentStateEstimate(o)
%GETCURRENTSTATEESTIMATE returns the current state estimate
%which is stored internally

  x = o.xest;
end
```

Whatever method we implement, its first argument must be the object on which the method operates. We may then invoke the method as `xest = o.getCurrentStateEstimate();`.

Let us now introduce a public method with which we can update the internally stored state estimate by providing the output y_k and the input u_k. This will update $\hat{x}$ with $A_d\hat{x} + L(\hat{y} - y) + B_d u$, where $\hat{y} = C_d\hat{x} + D_d u$. We will in fact delegate the computation of $\hat{y}$ to a private method which is of no interest to the user. We then implement the public method:

```
function updateStateEstimate(o, y, u)
    yest = o.estimateY(u);
    o.xest = o.Ad * o.xest + ...
        o.L * (yest - y) + ...
        o.Bd * u;
end
```

and the private method

```
function yest = estimateY(o, u)
    yest = o.Cd * o.xest + o.Dd * u;
end
```

If the class has several function definitions, to better organise the source code, we may define them in separate files as explained in the documentation.

There is a lot more one can do using classes. We may overload `disp` to customise how the objects display, or even overload operators to be able to add, subtract, multiply objects.

3. The MEX interface

3.1. **Calling C code from MATLAB.** A MEX interface allows us to call C functions from MATLAB. For that we need to implement a simple C function which maps the input arguments from MATLAB to C and the output arguments from C back to MATLAB. A MEX function implements the following function template:

```c
/**
 * The gateway function
 *
 * @param nlhs Number of output arguments
 * @param plhs Array of output argument pointers
 * @param nrhs Number of input arguments
 * @param prhs Array of input argument pointers
 */
void mexFunction(
    int nlhs,
    mxArray *plhs[],
    int nrhs,
    const mxArray *prhs[]);
```

A typical implementation is structured as follows:

(1) Declarations of function-scope variables
(2) Check input arguments: check the number of input arguments, their types and sizes. If the user provides an array for the wrong size to a MEX function, it is very likely that MATLAB will crash and you will have to restart it. To avoid such situations, you need to check, inside the MEX function, that all arguments are of the expected type and size and throw an error otherwise using `mexErrMsgIdAndTxt`.
(3) Cast MATLAB data into the appropriate types in C. MEX provides several functions such as `mxGetScalar` with which we can get the value of a scalar from MATLAB, `mxGetPr` which returns a pointer-to-double where the data of an array or matrix are stored and `mxGetM` and `mxGetN` to get the row- and column-dimensions of the MATLAB matrix. MATLAB Matrices are stored in memory in *column-major* order.
(4) Calling the C function: using the input arguments from MATLAB, we now invoke the C function. Here we need to properly handle exceptions. It is best if the C function returns a status code and if it does not succeed we need to throw an error and exit gracefully.
(5) Return the result to MATLAB:
(6) Free memory that was allocated inside the MEX function and is no longer needed (neither by MATLAB nor by the MEX function).

3.2. **Input-output.** The MEX function template (`void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])`) provides the number of right hand side arguments (input arguments) `nrhs` and an array of pointers to `mxArray`. Here, we discuss how to access the values of these input arguments in C. What is more, we discuss how to return data from C to MATLAB using `plhs`.

We provide a here full example: we have implemented in C a function which performs the operation $Y_{i,j} = 2X_{i,j}^2$ for a matrix $X \in \mathbb{R}^{m\times n}$. This is implemented by the C function `myFunction` which is shown below.

```c
/*
 * --- File: eagleMexFunction.c
 */

/* We must include mex.h */
#include "mex.h"

/**
 * This is the C function we need to interface from MATLAB.
 */
```

```c
void myFunction(double *x, double *y, mwSize nElements) {
    mwSize i; /* Index used in for loop   */
    for (i=0; i < nElements; ++i){
        double temp = x[i];
        y[i] = 2 * temp * temp;
    }
}

/**
 * This is a MEX interface for the function y = eagleMexFunction(x),
 * which accepts a matrix x and returns a matrix y = 2*x.^2.
 */
void mexFunction(
        int nlhs,
        mxArray *plhs[],
        int nrhs,
        const mxArray *prhs[]) {

    const mxArray * inputMatrix;    /* Pointer to RHS argument   */
    double * inputMatrixArray;      /* Data of RHS argument      */
    double * outputMatrixArray;     /* Data of LHS argument      */
    mwSize nrowsInputMatrix;        /* Rows of input matrix      */
    mwSize ncolsInputMatrix;        /* Columns of input matrix   */

    /* Make sure that the number of input arguments is exactly 1 */
    if (nrhs != 1) {
        mexErrMsgIdAndTxt("EAGLE:nrhs",
                "This function requires exactly one input argument");
    }

    /* Make sure that there are no more than 1 output arguments */
    if (nlhs > 1) {
        mexErrMsgIdAndTxt("EAGLE nlhs",
                "Too many output arguments (max. 1)");
    }

    /* Get the pointer of the input argument */
    inputMatrix = prhs[0];

    /* Verify that it is a matrix */
    if ( !mxIsDouble(inputMatrix) ) {
        mexErrMsgIdAndTxt("EAGLE:rhsType",
                "Input must be a matrix");
    }

    /* If the input is empty, print a warning message, but proceed */
    if (mxIsEmpty(inputMatrix)){
        mexWarnMsgIdAndTxt("EAGLE:emptyInput", "Input is empty!");
    }

    /*
     * Acquire a pointer to the data of the array. MATLAB packs the
     * data in column-major order and stores it as `double`
     */
    inputMatrixArray = mxGetPr(inputMatrix);

    /* Get the number of rows and columns of the input matrix */
    nrowsInputMatrix = mxGetM(inputMatrix);
    ncolsInputMatrix = mxGetN(inputMatrix);

    /* Create the output matrix */
    plhs[0] = mxCreateDoubleMatrix(
            nrowsInputMatrix,ncolsInputMatrix,mxREAL);
    outputMatrixArray = mxGetPr(plhs[0]);
    /* Perform computations */
    myFunction(inputMatrixArray, outputMatrixArray,
            nrowsInputMatrix * ncolsInputMatrix);

}
```

Let us now see how we can compile and use the above MEX interface. Compilation is as simple as executing `mex eagleMexFunction.c` in the command window of MATLAB. This will create an executable file with extension `.mexa64` (on Linux) or a similar on Windows and MacOSX systems. We will then be able to call this function as an ordinary MATLAB function: `y = eagleMexFunction(x);`. We may, however, compile with additional options such as gcc flags: `mex CFLAGS='$CFLAGS -Wall -O3' eagleMexFunction.c`. As a finishing touch we need to write documentation for our function (which should be accessible from MATLAB when typing `help eagleMexFunction`. For that, we need to create a `.m` file with no functionality, just containing documentation:

```
function y = eagleMexFunction(x)
%EAGLEMEXFUNCTION implements the operation y = 2 * x.^2.
%
%Syntax:
% y = eagleMexFunction(x)
%
%Input arguements:
% x   an m-by-n real matrix
%
%Output arguments:
% y   the result of the operation y = 2 * x.^2, or an
%     empty matrix is x is empty. NaNs and Inf are supported.
%

% Built-in function
```

Verify that the documentation is available by typing `help eagleMexFunction`.

Last, but not least, we must verify our implementation. Here is a unit test function where we use simple assertions (there exist more elaborate unit testing frameworks in MATLAB. If you are interested, visit this page.)

```
nmax = 50;          % maximum size of columns to be tested
mmax = 20;          % maximum size of rows
repetitions = 5;    % repetitions of each (random) test

% Test with random matrices of different dimensions
% including empty matrices
for n = 0:nmax,
    for m = 0:mmax,
        for i = 1:repetitions,
            xRandom = randn(m, n);
            yCorrect = 2*xRandom.^2;
            y = eagleMexFunction(xRandom);
            err = norm( yCorrect - y, Inf );
            assert( err < 1e-16, 'y not correct (up to tolerance)' );
        end
    end
end

% Test with a matrix with NaN and Inf/-Inf entries:
X = [1    2 -Inf;
     Inf  3   NaN];
Y = eagleMexFunction(X);
for i=1:2,
    for j=1:3,
        % isnan(X) <=> isnan(Y)
        assert ( ~isnan(X(i,j)) || isnan(Y(i,j)) );
        assert ( isnan(X(i,j)) || ~isnan(Y(i,j)) );
        % isinf(X) => isinf(Y)
        assert ( ~isinf(X(i,j)) || isinf(Y(i,j)) );
        % isinf(-X) => isinf(-Y)
        assert ( ~isinf(-X(i,j)) || isinf(-Y(i,j)) );
    end
end
```

We may provide structures and cells to a MEX interface, as well as pack the results in a structure. Although this functionality is very useful, we will not be using it in this project as we only need to implement simple functions such as linear controllers and observers which involve simple algebraic operations (matrix-vector multiplications, scalar-vector multiplications and vector-vector additions).

4. Code generation

4.1. **Code generation principles.** In the EAGLE project you will have to generate C code from MATLAB which will run on the BareMetal core of Zybo. For that purpose, you need to be able to modify the tuning parameters of your controllers/observers, build new controller/observer gains and with an one-line command generate C code.

This code needs to have a *mutable* and an *immutable* part. The *immutable part* contains pieces of code which do not change if you modify the tuning parameters. Header files, for example, should be immutable. The mutable part consists of those parts of the code which change when you modify the tuning parameters.

Examples of mutable code are controller and observer functions. A very handy utility for code generation is a function that generates code which performs the operation $y \leftarrow \alpha z + Ax$, over a constant matrix A and variables x, y and z which essentially correspond to memory positions. Variable z can be equal to variable y, but y cannot point to the same memory position as x. A MATLAB function which implements this functionality would have the following template:

```
function printMatVec(f, A, alpha, nameA, nameX, nameY, nameZ)
%PRINTMATVEC prints a matrix-vector product (hard-coded)
%
% y := alpha * z + A * x;
%
%Syntax:
% printMatVec(f, A, alpha, nameA, nameX, nameY)
% printMatVec(f, A, alpha, nameA, nameX, nameY, nameZ)
%
%Arguments:
% f          file handler (use fopen to produce one)
% A          matrix A (the dimensions of x, y and z are inferred from the
%            dimensions of A)
% alpha      scalar
% nameA      name of variable A
% nameX      name of variable x
% nameY      name of variable y (cannot be equal to nameX)
% nameZ      name of variable z (can be equal to nameY). If nameZ is not
%            specified and alpha is not set to zero, then it is assumed that
%            nameZ = nameY and the operation y := alpha * y + A * x is
%            generated.
%
```

The actual implementation of such a function is left to you as an exercise. Here is an example of use:

```
A = randn(3,2);   % random matrix
filep = 1;        % print to the system output
printMatVec(filep, A, -1.5, 'A', 'x', 'y');
```

The output of this function will be C code of the following form (the code generator adds some elementary comments to make the C code more readable):

```
/* y <-- -1.5 * y + A * x */
y[0] = (-1.5) * y[0] + (+0.333510833066) * x[0] + (+0.391353604433) * x[1] ;
y[1] = (-1.5) * y[1] + (+0.451679418928) * x[0] + (-0.130284653146) * x[1] ;
y[2] = (-1.5) * y[2] + (+0.183689095862) * x[0] + (-0.476153016619) * x[1] ;
```

We may of course specify a file identifier obtained using `fopen`. Here is an example of use:

```
% Open file for writing
fileID = fopen('eagle.c', 'w');
% If the file cannot be opened,
if fileID < 0,
    error('EagleCodegen:cannotOpenFile', 'Cannot open file');
end
% Generate code for matrix—vector multiplication
printMatVec(fileID , A, -1.5, 'A', 'x', 'y');
% Close the file stream
fclose(fileID);
```

The code that MATLAB produces will run on Zybo. Therefore, it needs to adhere to the ANSI-C standard. A few guidelines regarding the C implementation:

(1) Declarations and code should be separated
(2) Use file-scope variables (using the keyword `static`), but not global variables which can be accessed externally.
(3) It is not allowed to allocate static variables (i.e., we cannot do `static float x[2] = 0.5, 1.3;` Instead, we need to write initialisers. These are methods which are called once the program starts and initialise the values of our static variables.
(4) When compiling the auto-generated C code on your system and when running unit tests, compile with `-ansi`, `-pedantic` and `-Wall`. It is safer to compile with `-Werror` which will turn all warnings into errors.
(5) There is no need for dynamic memory allocation. All array sizes are known beforehand.

4.2. **Testing the generated code.** In order to make sure that your implementation of `printMatVec` is correct, you need to test it. Does it return the same value `y` for fixed inputs? For that purpose, you need to write unit tests.

Testing should also happen at a higher level. Once you have automatically generated C code for your controller, you have to make sure that it returns correct control actions for given (estimated) states. Again, this calls for unit tests on the automatically generated code.

It is highly advisable to write a MEX function to interface the automatically generated C code from MATLAB. Then, using the C implementations of your controllers and observers you will be able to test them against the nonlinear continuous time system which runs in MATLAB (see Section 1.5). This will serve as an integration test to ensure that the C implementation of the controller behaves as it is expected to (with respect to the simulation environment in MATLAB). This should be the last step before running the code on the quadcopter.

5. Interfacing the EAGLE

A simple way to interface the quadcopter from MATLAB is to use the logs. Flight data can be stored in the on-board SD card and parsed from MATLAB afterwards. The data format needs to be decided early in the project because inspecting the logs provides great insight and is invaluable for problem solving.

JSON is a really handy solution as there exist parsers for MATLAB as well as all major programming languages including Python. You can export data from Python into JSON files and store them into the SD card. Then you can parse the data from MATLAB, plot them and post-process them as you wish.

Here is a simple piece of JSON which encodes flight information

```
{
    "data": {
            "timestamp" : 12.0102,
                    "q" : [1,0,0,0],
               "vision" : [0.45,1.12],
                "sonar" : 19.010,
        "sonarFiltered" : 1.500,
                 "vest" : [0.01,-0.03,0.05],
                 "pest" : [0.44,1.13,1.52],
                "omega" : [0.001,0.001,0.003]
    }
}
```

In this example, we log the timestamp when the data was logged, the quaterion (`q`), the position measurements from the vision module (`vision`), the raw sonar measurements (`sonar`), the filtered sonar measurements after the application of a median filter (`sonarFiltered`), the estimated position (`pest` — no pun intended), the estimated velocity (`vest`) and the angular velocity (`omega`). Here, the raw measurement was 19.010m, but the filtered altitude turned out to be 1.500m which is also close to the estimated altitude of 1.52m. We may use jsonlab to load these data into MATLAB.

You will most likely receive data in real-time from the quadcopter while it flies and it is very convenient to be able to visualise them. Sending the data while the quadcopter flies has a few important advantages:

(1) You do not have to rely on the on-board logging system which might fail if, for example, the quadcopter crashes. Transmitting the data in real time reduces significantly the risk of losing them.
(2) You will be able to gain better insight. For example, you will be able to tell whether some sudden jumps or oscillations are because of bad measurements. You will also be able to see whether the median filters rejects the outliers in the sonar measurements.
(3) You will be able to demonstrate how the quadcopter works while it flies.

When it comes to visualising data that arrive from a data stream, you may use MATLAB's `addpoints`. Here is an example of use taken from the online documentation (with a slight modification):

```
h = animatedline;
axis([0,40*pi,-1,1])
x = linspace(0,40*pi,10000);
y = sin(x);
for k = 1:length(x)
    % Replace x(k) and y(k) with streaming data
    addpoints(h,x(k),y(k));
    if ~mod(k,100), drawnow; end
end
```

The call to `drawnow` updates the plot. This is likely to slow down the animation. This is why we prefer to call it periodically using `if ~mod(k,100), drawnow; end`.

6. Self-evaluation checklist

Preparation and planning

- ☐ Everyone has read this document
- ☐ Sketch of project structure (what is going to be implemented)
- ☐ You have studied the documentation of all major functions such as `kalman`, `lqr`, `lqi`, `ss`

Code versioning system

- ☐ Git repository is set up
- ☐ Git repository is regularly used (total number of commits: ___)
- ☐ Git repository actively updated
- ☐ README file
- ☐ `.gitignore` file
- ☐ No binary files
- ☐ Branch created
- ☐ Pull request created
- ☐ Pull request reviewed; merge performed

Project management

- ☐ Use of issue tracker
- ☐ Use of trello or similar project management system

Documentation and comments

- ☐ All functions are documented
- ☐ All variables have explanatory comments
- ☐ All implementations contain comments

Good coding practices

- ☐ No Simulink
- ☐ Adoption of naming convention
- ☐ No hard-coded data (e.g., system parameters)
- ☐ No global variables
- ☐ No commented-out code or print statements for debugging purposes
- ☐ No functions with too many arguments (use structures instead)

- ☐ (Optional) use of classes

Testing

- ☐ Unit tests