Adaptive Algorithms and Parallel Computing
Object-Oriented Programming in MATLAB (Primer)

Academic year 2014-2015

Simone Scardapane
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - Overloading indexing
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
What is OOP?

• **Procedural programming** languages (e.g. C) focus on defining *procedures* that implement the core logic of your problem.
• This is decoupled from identifying suitable *data structures*.
• **Object-oriented programming** (OOP) is a programming paradigm where the focus is on *representing* the domain of your problem using *objects*.
• An object encapsulates an internal state, and exposes some methods through an interface. In this sense, it combines handling data and operations.
• Note: a *class* is the source code, from which multiple *objects* are instantiated.
Benefits of OOP

**Modularity**
OOP forces the developer to design modular software.

**Data encapsulation**
An object can hide part of its information at run-time.

**Logic encapsulation**
It is not necessary to understand the internal logic of an object before using it.

**Maintainability**
OOP code is easier to maintain/document/debug.

**Extensibility**
Features such as *inheritance* makes OOP code easier to extend.
OOP in MATLAB

• Full support for OOP (with a completely rewritten syntax) was introduced in MATLAB R2008a.
• Since then, many basic functionalities have been re-introduced to be compatible with an OOP standard.
• As an example, the graphic engine from MATLAB R2014b is class-based (so-called HG2).
• MATLAB R2015a has introduced additional functionalities in term of editing capabilities.
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - Overloading indexing
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
Defining a class

A class is instantiated with the `classdef` keyword:

```matlab
classdef Rectangle
    properties
        % Properties go here
    end

    methods
        % Methods go here
    end
end
```

The source code must go inside a file with the same name (e.g. ‘Rectangle.m’).
Defining a property

The internal state of the class is saved as *properties* of the class:

```matlab
properties
    width;  % The width of the rectangle
    height; % The height of the rectangle
end
```

MATLAB is *weakly typed*, meaning there is no need of defining the type of each property. A class without methods is equivalent to a *struct*.

```matlab
>> r = Rectangle
r =
    Rectangle with properties:
        width: []
        height: []

>> r.width = 5
r =
    Rectangle with properties:
        width: 5
        height: []
```
Defining a constructor

An object is created by invoking a special method known as *constructor*. Let us define a custom constructor for our class:

```matlab
methods
    function obj = Rectangle(w, h)
        obj.width = w;
        obj.height = h;
    end
end
```

The constructor has a very specific syntax. By default, MATLAB generates a default constructor with no input arguments. Since each class can only have a single constructor, our definition overrides the MATLAB one:

```matlab
>> r = Rectangle(3, 2)

r =
    Rectangle with properties:
        width: 3
        height: 2

>> r = Rectangle

Error using Rectangle (line 10)
Not enough input arguments.
```
Defining a method

The third essential element of a class definition is a set of methods:

```
methods

% Constructor goes here

function p = get_perimeter(obj)
    p = 2*(obj.width + obj.height);  
end

function obj = scale(obj, n)
    obj.width = n*obj.width;
    obj.height = n*obj.height;
end

end
```

>> r = Rectangle(5, 3);
>> per = r.get_perimeter()
per =
   16
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - Overloading indexing
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
Defining a static method

A static method is associated to a class, but not to any specific instance of that class. Here is an example of a static method:

```matlab
classdef Rectangle
    % Previous code goes here

    methods (Static)
        function s = get_description()
            s = 'Class for representing rectangles';
        end
    end
end
```

Note that a static method does not need an instance of the class as argument. In this way, you can call a static method without first declaring an object:

```matlab
>> Rectangle.get_description()
ans =
     'Class for representing rectangles'
```
Defining a constant property

A class in MATLAB can have a constant property (or more than one), i.e. a property whose value is never modified after the first assignment:

```matlab
classdef Circle
    properties (Constant)
        pi = 3.14;
    end
    % Other definitions
end
```

```
>> Circle.pi
ans =
    3.1400

>> r.pi = 2
You cannot set the read-only property 'pi' of Rectangle.
```
Defining a private property

Finally, a MATLAB class can have one or more *private* properties, i.e. properties that should not be visible (nor modifiable) from the outside:

```matlab
classdef Rectangle
    properties (Access = private)
        width;
        height;
    end
    % Other definitions go here
end
```

```matlab
>> r = Rectangle(3,2)
```

```
    r = 
    Rectangle with no properties.
```

You can also have private methods (using an equivalent syntax). Similarly, you can have properties that are visible but not modifiable: [Specifying Property Attributes].
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - Overloading indexing
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
Writing the documentation

When documenting your class, you should be careful in respecting MATLAB conventions:

classdef Rectangle
    % Rectangle — Class for defining rectangles
    % This class can be used to define objects representing rectangles.
    % Initialize the class as:
    %   r = Rectangle(3, 2);
    % Then, [...].
    % See also: UINT32

    properties
        width; % Width of the rectangle
        height; % Height of the rectangle
    end

    methods
        function p = get_perimeter(obj)
            % Compute the perimeter of the rectangle
            p = 2*(obj.width + obj.height);
        end
    end
end
Accessing the documentation

>> help Rectangle
   Rectangle - Class for defining rectangles
   This class can be used to define objects representing rectangles.
   Initialize the class as:
       r = Rectangle(3, 2);
   Then, [...].
   See also: uint32

>> help Rectangle.width
   width - Width of the rectangle

>> help Rectangle.get_perimeter
   Compute the perimeter of the rectangle

>> % See for yourself
    doc Rectangle
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - Overloading indexing
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
Difference between handle and value classes

• All the classes that we defined up to now are value classes. When you pass a value class’ object to a function, a copy is created (similarly when you assign to another variable).

• You can create handle classes that work in the opposite way. Passing an object of an handle class to a function passes a reference to the object itself.

• Value classes are useful when the semantics of your objects is similar to numerical classes or data structures. Similarly, handle classes are useful when there is often the need of sharing objects.

• This is especially confusing because the resulting syntax is mixed with respect to other common programming languages (e.g. Java or C++).
Defining an handle class

An handle class should *derive from the basic* handle class (more on this syntax later on):

```matlab
classdef Rectangle < handle

    properties
        width;  % Width of the rectangle
        height; % Height of the rectangle
    end

    methods
        function obj = Rectangle(w, h)
            obj.width = w;
            obj.height = h;
        end

        function scale(obj, n)
            obj.width = n*obj.width;
            obj.height = n*obj.height;
        end
    end
end
```
Using an handle class

```matlab
>> r = Rectangle(3, 2);
>> r2 = r;
>> r2.scale(2)
>> r2

r2 =

    Rectangle with properties:

      width: 6
      height: 4

>> r

r =

    Rectangle with properties:

      width: 6
      height: 4
```
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - Overloading indexing
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
What is an event?

- A very interesting aspect of handle classes is their ability of defining **events**.
- Roughly, an event is a specific message which is triggered at certain points in the code. This is known as **notification**.
- It is possible to associate one (or more) **listeners** (also known as **callbacks**) to a specific notification.
- In this way, you can define a dynamic behavior that is executed at runtime depending on how the class is used.
- As an example, we will show how to keep track of how many times a specific function is called.
Defining an event (1/3)

First of all, you must define any event that can be triggered using a similar syntax with respect to a property:

```matlab
classdef Rectangle < handle
    events
        ComputePerimeter
    end

    % Other definitions
end
```

At this point, it is possible to trigger this event whenever required:

```matlab
function p = compute_perimeter(obj)
    p = 2*(obj.width + obj.height);
    notify(obj, 'ComputePerimeter');
end
```
Defining an event (2/3)

We keep track of how many times the function is called by using an internal private counter:

```matlab
properties (Access = private)
    nComputedPerimeter;
end
```

We add a property to initialize the counter and link the event to its specific listener:

```matlab
function track_perimeter_method(obj)
    obj.nComputedPerimeter = 0;
    addlistener(obj, 'ComputePerimeter', @Rectangle.update_nComputedPerimeter);
end
```
Defining an event (3/3)

Finally, we define the listener:

```matlab
methods(Static)
    function update_nComputedPerimeter(r, ~)
        r.nComputedPerimeter = r.nComputedPerimeter + 1;
    end
end
```

We also need a function to print the result:

```matlab
function print_tracking_info(obj)
    fprintf('The ''get_perimeter'' function has been called %i times.
', obj.nComputedPerimeter);
end
```
The event in action

```matlab
>> r = Rectangle(3, 2);
>> r.track_perimeter_method
>> r.compute_perimeter;
>> r.compute_perimeter;
>> r.print_tracking_info
The 'get_perimeter' function has been called 2 times.
>> r.nComputedPerimeter
You cannot get the 'nComputedPerimeter' property of Rectangle.
```
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - Overloading indexing
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
Overloading the times operator

By using OOP, it is possible to overload several basic MATLAB behaviors for your custom object. As an example, we can overload the operator * for our rectangle class:

```matlab
function r = mtimes(obj1, obj2)
    if (isa(obj1, 'Rectangle') && isnumeric(obj2))
        r = obj1.scale(obj2);
    elseif (isnumeric(obj1) && isa(obj2, 'Rectangle'))
        r = obj2.scale(obj1);
    else
        error('One operator must be a rectangle and the other numeric');
    end
end
```

Note that in our implementation we have to explicitly check for the order of the operands. There is a large number of overloadable operators: [MATLAB Operators and Associated Functions].
Rectangular multiplication

```plaintext
>> r = Rectangle(3, 2);
>> r = r*2
r =

    Rectangle with properties:
        width: 6
        height: 4

>> r = 1/2*r
r =

    Rectangle with properties:
        width: 3
        height: 2

>> r = 'a'*r
Error using Rectangle/mtimes (line 27)
One operator must be a rectangle and the other numeric
```
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - **Overloading indexing**
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
Indexing in MATLAB

Just like operators, every indexing operation in MATLAB has an associated function:

- **subsref**: subscripted reference, e.g. `A(1:3)` or `A{1:3}` or `A.area`.
- **subsasgn**: subscripted assignment, e.g. `A(3) = 1`.
- **subsindex**: subscripted indexing using an object.

Each of these functions can be overloaded to implement custom behavior.
Example: a class for damped oscillations

Consider the following class representing the function \( y = \frac{\sin(x)}{x} \).

```matlab
classdef DampedSine
    properties
        limit; % x−limit for function
    end
    methods
        function obj = DampedSine(limit)
            obj.limit = limit;
        end
        function v = get_value(obj, x)
            % Get the value of the function
            if (any(x < 0) || any(x > obj.limit))
                error('Argument is outside specified bounds');
            end
            v = sin(x)./x;
        end
        function plot(obj)
            % Plot the function
            x = 0:0.01:obj.limit; plot(x, obj.get_value(x));
        end
    end
end
```
Overloading indexing

We can implement a custom indexing operation to create a sort of “callable” object:

```matlab
function B = subsref(A,S)
    if(strcmp(S(1).type, '()') && length(S(1).subs) == 1)
        B = A.get_value(S(1).subs{1});
        if(length(S) > 1)
            B = builtin('subsref', B, S(2:end));
        end
    else
        B = builtin('subsref', A, S);
    end
end
```

```matlab
>> ds = DampedSine(20);
>> ds.get_value(5)
ans =
    -0.1918
>> ds(5)
ans =
    -0.1918
```
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - Overloading indexing
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
Custom displaying

You can overload how a variable is displayed on the console:

```matlab
function disp(obj)
    fprintf('Rectangle with width=%.2f and height=%.2f\n', obj.width, obj.height);
end
```

```bash
>> r = Rectangle(3, 2)

r =

Rectangle with width=3.00 and height=2.00
```

There are many other custom behaviors that you can implement: [Customize MATLAB Behavior].
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - Overloading indexing
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
What is inheritance?

- **Inheritance** is a feature of OOP languages that lets one class to ‘inherits’ the behavior from a specific super-class.
- This allows a strong amount of code reuse, and the possibility of creating hierarchies of classes for your application (e.g. the hierarchy of data types in MATLAB).
- You can think of class B inheriting from class A as the implementation of the semantic “B is-a A” (not to be confused with the classical “B has-a A”).
- Closely connected is the idea of **polymorphism**: a single interface is declined in multiple forms, and the specific implementation is decided only at run-time.
An example of inheritance

A square “is-a” rectangle, whose width and height are equivalent:

```matlab
classdef Square < Rectangle
    properties
    end

    methods
        function obj = Square(l)
            obj = obj@Rectangle(l, l);
        end
    end
end
```

Note the syntax of the call to a constructor of the super-class.
Method overriding

The class `Square` inherits all the behavior of its superclass, thus allowing a strong amount of code reusing between implementations:

```matlab
>> s = Square(3);
>> s.compute_perimeter

ans =
    12
```

You can also `override` a specific method (in this case, for computational reasons):

```matlab
classdef Square < Rectangle
    % ...
    methods
    % ...
    function p = compute_perimeter(obj)
        p = 4*obj.height;
    end
end
end
```
Adding methods to the subclass

You can define methods that are exclusive of a particular subclass:

```matlab
classdef Square < Rectangle
    methods
        function l = get_length(obj)
            l = obj.width;
        end
    end
end
```

```matlab
>> s = Square(3);
>> s.get_length
ans =
    3
>> r = Rectangle(3, 2);
>> r.get_length
No appropriate method, property, or field get_length for class Rectangle.
```
Sealed and protected attributes

You can specify that a property (or method) is accessible only to classes inheriting the super-class:

```matlab
classdef A
    properties (Access=protected)
        foo;
    end
end
```

Additionally, you can specify that a particular class cannot be inherited from:

```matlab
classdef A (Sealed = true)
    % ...
end
classdef B < A
    % ERROR
end
```
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - Overloading indexing
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
Abstract classes

Inheritance allows to define *abstract* classes. These cannot be instantiated, but they can:

- Abstract code from any specific sub-class.
- Provide an interface for a set of functions that any sub-class must implement.

Let us consider the following class hierarchy:

```
Shape (Abstract)

  Rectangle

    Square

  Circle
```
Defining the abstract class

```matlab
classdef Shape
    properties
        n_sides; % Number of sides
    end

    methods (Abstract=true)
        p = compute_perimeter(obj); % Compute the perimeter
        obj = scale(obj, n); % Scale the object
    end

    methods
        function obj = Shape(n)
            obj.n_sides = n;
        end
        function print_info(obj)
            fprintf('This is a geometric shape with %.2f sides.\n', obj.n_sides);
        end
    end
end
```
Abstract classes

Defining the first sub-class

```matlab
classdef Rectangle < Shape
    properties
        width;  % Width of the rectangle
        height;  % Height of the rectangle
    end
    methods

        function obj = Rectangle(w, h)
            obj = obj@Shape(4);
            obj.width = w;
            obj.height = h;
        end

        function obj = scale(obj, n)
            obj.width = n*obj.width;
            obj.height = n*obj.height;
        end

        function p = compute_perimeter(obj)
            p = 2*(obj.width + obj.height);
        end
    end
end
```
Defining a new sub-class

Note: the class for the Square is unchanged with respect to before.

classdef Circle < Shape
    properties
        radius;  % Radius of the circle
    end
    methods
        function obj = Circle(r)
            obj = obj@Shape(Inf);
            obj.radius = r;
        end
        function obj = scale(obj, n)
            obj.radius = n*obj.radius;
        end
        function p = compute_perimeter(obj)
            p = 2*pi*obj.radius;
        end
    end
end
Using the class hierarchy

```matlab
>> r = Rectangle(3,2);
>> r.scale(2)

ans =

    Rectangle with properties:

        width: 6
        height: 4
    n_sides: 4

>> s = Shape(3);
Error using Shape
Abstract classes cannot be instantiated.
    Class 'Shape' defines abstract methods and/or properties.
```
Refactoring

We can see that the code of the `scale` method is semantically equivalent in all cases: it multiplies by $n$ all properties, except the number of sides. We can, in principle, abstract this into the class `Shape` to provide a default implementation:

```matlab
classdef Shape
    % ...
    function obj = scale(obj, n)
        pr = properties(class(obj));
        for ii = 1:length(pr)
            if (~strcmp(pr{ii}, 'n_sides'))
                obj.(pr{ii}) = n*obj.(pr{ii});
            end
        end
    end
end
end
```

This is known as code refactoring.
Overview

1. **Fundamentals of OOP**
   - What is OOP?
   - Syntax: defining a basic class

2. **More OOP**
   - Static and private elements
   - Class documentation
   - Handle and Value classes
   - Events and notifications

3. **Custom MATLAB Behavior**
   - Overloading an operator
   - Overloading indexing
   - Custom displaying

4. **Inheritance**
   - Inheritance and Polymorphism
   - Abstract classes

5. **Example**
   - Simple Filtering Application
Description of the example

As a final example, we implement a relatively simple (but with lots of possible extensions) filtering application. We have the following specifics:

- A class for generating the underlying system to be identified. For simplicity, the system is supposed to be linear, and the output corrupted with Gaussian noise.
- An abstract class for implementing any linear adaptive filter. One particular implementation of this class implementing the standard LMS filter with constant step size.
- One class taking care of plotting the results. For simplicity, we implement only one method for plotting the MSE in logarithmic scale.
Signal generation

```matlab
classdef SignalGenerator
    % SignalGenerator – Class for generating input and output signals
    % properties
    taps; % taps of the filter
    noise_var; % variance of the noise
    wo; % optimal weights of the linear system
    end

    methods
        function obj = SignalGenerator(taps, noise_var)
            obj.taps = taps;
            if nargin < 2
                obj.noise_var = 0.01;
            else
                obj.noise_var = noise_var;
            end
            obj.wo = rand(taps, 1)*2 - 1;
        end
    end
end
```
Signal generation (cntd)

```matlab
classdef SignalGenerator
    % ...

    function [X, d] = generate_signal(obj, N)
        % Generate signal of length N
        % The outputs are:
        % X: a N\times M input matrix, where M is the number of taps
        % d: a N\times 1 vector of output values, corrupted by noise
        X = randn(N, obj.taps);
        d = X*obj.wo + randn(N, 1)*sqrt(obj.noise_var);
    end

    % ...
end
```
Abstract class for filters

```matlab
classdef AdaptiveFilter
    % AdaptiveFilter — Abstract class for adaptive filters

    properties
        w_estimated;        % Estimated weights
        error_history;      % Vector of a-priori errors
    end

    methods
        function obj = AdaptiveFilter(taps)
            obj.w_estimated = zeros(taps, 1);
        end

        % Here we need:
        % 1) one function for processing the full dataset one item at a time.
        % 2) one (abstract) function for implementing the single adaptation step.
    end
end
```
Abstract class for filters (cntd)

```matlab
classdef AdaptiveFilter
    % ...
    methods
        % ...
        function obj = process(obj, X, d)
            % Process the overall signal X/d
            N = length(d);
            obj.error_history = zeros(N, 1);
            for ii = 1:N
                obj.error_history(ii) = d(ii) - X(ii, :) * obj.w_estimated;
                obj = obj.adapt(X(ii, :)', d(ii), obj.error_history(ii));
            end
        end
    end

    methods(_abstract)
        obj = adapt(obj, x, y, e); % Signal adaptation step
    end
end
```
Least Mean-Square Filter

```matlab
classdef LMSFilter < AdaptiveFilter
    % LmsFilter -- Least Mean-Square adaptive filter with constant step
    size
    properties
        step_size; % Step size
    end
    methods
        function obj = LMSFilter(taps, step_size)
            obj = obj@AdaptiveFilter(taps);
            obj.step_size = step_size;
        end
        function obj = adapt(obj, x, ~, e)
            obj.w_estimated = obj.w_estimated + obj.step_size*e*x;
        end
    end
end
```
classdef PlotHelper
    % PlotHelper — Helper class for plotting

    properties (Constant)
        font_size = 8;          % Fontsize
        font_size_leg = 6;      % Font size (legend)
        font_name = 'TimesNewRoman';  % Font name
        line_width = 1;          % LineWidth
    end

    methods (Static)

        function plot_MSE_db(xlab, ylab, leg, varargin)
            % Inputs are: x-label y-label, cell array of strings for the
            % legend. Each additional input is a time-series to be plotted.

            % Code goes here
    end
end
end
Helper function for plotting (cntd)

```matlab
function plot_MSE_db(xlab, ylab, leg, varargin)
    % Each additional argument is a time series to be plotted

    % Simple filter for smoothing the results
    [bb, aa] = butter(2, 0.02);

    N_series = length(varargin);
    cmap = hsv(N_series);
    figure(); hold on; box on; grid on;

    for ii = 1:N_series
        plot(filter(bb, aa, 10*log10(varargin{ii}.^2)), 'LineWidth', PlotHelper.line_width, 'Color', cmap(ii, :));
    end

    xlabel(xlab, 'FontSize', PlotHelper.font_size, 'FontName', PlotHelper.font_name);
    ylabel(ylab, 'FontSize', PlotHelper.font_size, 'FontName', PlotHelper.font_name);
    legend(leg, 'FontSize', PlotHelper.font_size, 'FontName', PlotHelper.font_name);
end
```
Testing the application

```matlab
% Repeatable experiment
rng(1);

% Generate data for simulation
sg = SignalGenerator(5, 0.1);
[X, y] = sg.generate_signal(5000);

% Create two filters
f1 = LMSFilter(5, 0.01);
f2 = LMSFilter(5, 0.001);

% Adapt the filter
f1 = f1.process(X, y);
f2 = f2.process(X, y);

% Plot
PlotHelper.plot_MSE_db('Sample', 'MSE', {'Filter 1', 'Filter 2'}, f1.error_history, f2.error_history);
```
Result

![Graph showing MSE for Filter 1 and Filter 2 over samples from 0 to 5000. The graph illustrates the performance of both filters, with Filter 2 generally showing a lower MSE than Filter 1.]