Industrial Software Development (ISDe) Course
Evaluation of Programming Skills

Name: …… **Surname**: …… **Student ID**: ……

Programming Exercise: Implement the code as described below.

Create a package 'conv_1d_kernels' that will contain an abstract class and three inherited classes of objects. Each class will represent a different one-dimensional convolutional kernel, used to filter one-dimensional signals.

The abstract class is named `CConvKernel`. It stores **`self.`**`_kernel_size` as an attribute (an odd integer, by default set to 3) representing the size of the kernel mask. The kernel mask will be dynamically generated by the inherited classes, depending on the chosen filter (see below). The abstract class requires implementing the method `kernel_mask` to generate and store the mask in the attribute **`self.`**`_mask`.

The abstract class `CConvKernel` also implements the *concrete* method `kernel`. This method takes as input a flatten `numpy` array `x` (the input signal) and returns `xp`, namely, the result of applying the given kernel filter to `x`. The value `xp[i]` is computed by centering the kernel mask at `x[i]` and then computing the scalar product between the elements of the kernel mask and the corresponding ones in `x`. For simplicity, the first and last `(kernel_size-1)/2` elements of `x` can remain unchanged.

Each inherited class will contain a different kernel mask, corresponding to a different filter. The mask has to be dynamically generated depending on the `kernel_size` value only once, when the class is constructed, and re-generated only if the parameter `kernel_size` is changed (*hence, be sure to account for this behavior when defining the corresponding setter*). It is then stored in **`self.`**`_mask`.
The kernel classes to be implemented are:

- `CConvKernelMovingAverage` whose kernel mask is all ones, e.g., k=1/3*[1 1 1]; and
- `CConvKernelTriangle` whose kernel mask increases linearly up to the midpoint, and then decreases linearly, e.g., k=1/4*[1 2 1] for `kernel_size=3` and k=1/9*[1 2 3 2 1] for `kernel_size=5`. Note that kernel masks are always normalized to sum up to one.

In addition, create the class `CConvKernelCombo`, which takes a sequence of convolutional kernels as inputs and applies them in the given order to the input signal `x`.

For all the parameters, setters and getters should be available (except for **`self.`**`_mask`, for which only the getter has to be available, being not possible to set it from outside the class). Use deepcopy to make sure your methods do not change the original data.

In the 'main' program:

- Load the MNIST digit data and instantiate objects.
- Instantiate the combo kernel by passing the moving average kernel, the triangle kernel and the moving average kernel again as the kernel input sequence.
- Randomly draw one image per class, for a total of ten images, plot and save them in a PDF file.
- Apply the moving average kernel to the selected ten images, plot and save them in a PDF file.
- Apply the triangle kernel to the selected ten images, plot and save them in a PDF file.
- Apply the combo kernel to the selected ten images, plot and save them in a PDF file.

(hint: use plt.savefig('digit' + str(k) + '.pdf')