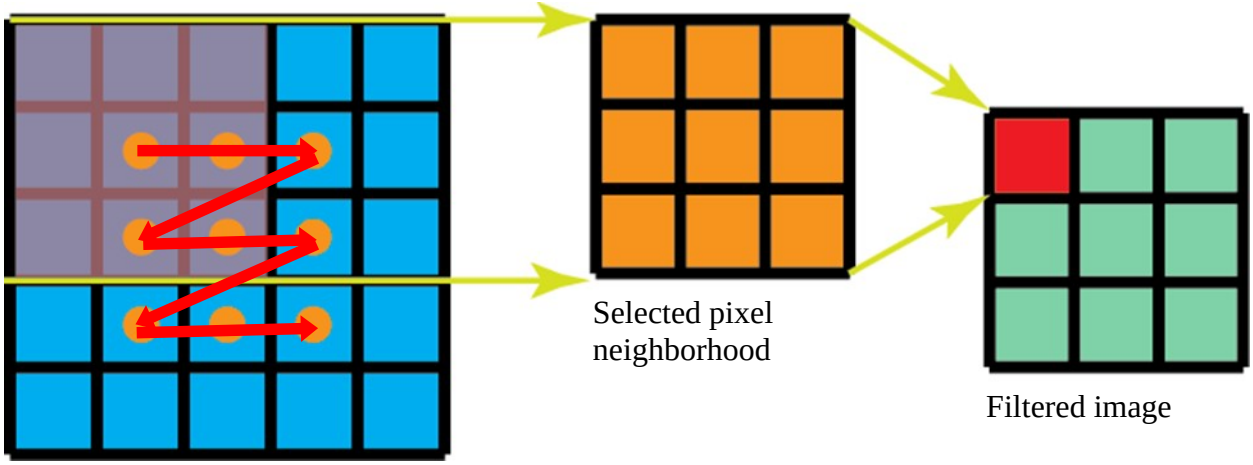
Exercise 06

1) Understanding filtering

First, we are going to start with implementing some basic filters. For that, open: src/e6/filter.py.
Each filter that we are going to implement is considering a region (called kernel) of "active" pixels (e.g., defined by a sliding window). This region can be any shape, but for simplicity, we are assuming a square neighborhood. A filter with a kernel size of 3 thus considers the 3x3 region around the active pixel for the filter function.



3x3 sliding kernel over the input image (blue)

When filtering an image, we are moving the kernel like a sliding window over the input image (red arrows), and apply some sort of function (e.g., summing, averaging, etc.) to the pixels in the kernel. The result is a single value, which is then written to the output (filtered) image at the corresponding location (i.e., the pixel that is "below the kernels center pixel"). Note that any filter with a kernel size >1 will either reduce the output image size, or must make some assumptions for the border pixels (e.g., wrap the filter, assume all pixels to be zero).

2) The code base

I have provided you with a basic interface for any Filter that you may need to implement called **IFilter2D**. I recommend to extend it to a new class (e.g., BaseFilter2D), that handles the kernel (i.e., sliding window creation), and pixel selection based on the sliding window and image for you. Also make sure to handle border behavior of the filter kernel here! Use the selected pixels in an (abstract) `_apply_filter_func` method that processes only the current subset of pixels and creates the output pixel at the current filter location. You can then use this as a new base class for your filters and avoid a lot of code duplication. If you are having trouble, you can also use the pre-implemented base-filter classes from the solution:

- **BaseFilter2D**, which handles the sliding window and pixel selection for you. You only need to implement `_apply_filter_func` method, which is called only on the selected pixels within the kernel region (i.e., for a 3x3 filter, this method will only receive a 3x3 array of pixel values for the current location of the sliding kernel).
- **BaseKernelFilter2D**, which is a specialisation of **BaseFilter2D**. You still need to implement the `_apply_filter_func` as well as the `_create_kernel` method (which gets called during initialisation). This class is purely convenience to have a dedicated kernel creation for non-trivial filters like Gauss- or Sobel-Filter.

However, I highly recommend to implement the sliding-window creation, pixel selecting and border-pixel handling at least once yourself to understand the process.

To add your filter to the ui, please add one instance of your filter to the (empty) array in the `get_filters`-method in filter.py.

3) Task 1: Median filtering

For the first task, we are going to implement a median filter. Here, the kernel is purely for selecting pixels (i.e., every entry in the kernel matrix is 1). Create a new filter-class (e.g., MedianFilter), either subclass IFilter2D or BaseFilter2D and implement the corresponding filter methods (have a look at the definition of the median).

4) Task 2: Advanced kernels

Our next three filters will be gaussian filter which is a noise reduction filter, the sobel filter, which detects edges in cardinal directions and a motion blur filter.

```
1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1,
```

First, let's start with the motion blur filter. Create a kernel that has 1s along only one axis, e.g. the diagonal or the horizontal. Depending on the orientation of the 1s, the motion blur is applied (in the example cases either diagonally blurred or horizontal). Make sure to normalize the filter to not increase or alter the pixel values!

Next, for the gaussian filter, we need to create a kernel that represents a discrete 2D gaussian bell curve. For that, we need an additional parameter, the variance of the gauss curve. We then can simply apply a 1D gauss in both x and y direction (or combine the functions into a single, 2D function) and determine the kernel values. Again, make sure to normalize the kernel!

Finally, read carefully the Formulation-Section of the sobel-operator! Notice that we need to filter both in x and y direction separately, and that we need to apply a convolution rather than a simple multiplication! For simplicity, consider adding two additional filters that show only the result of the x and y sobel filter respectively, before combining the result into the final image.

5) Task 3: Morphological operators

For the final task, we will implement basic morphological operations, namely erosion, dilation, opening and closing. These filters are the bread-and-butter filter for noise reduction and image pre-processing before applying more complex methods (e.g., segmentation). Erosion and dilation are the basic building blocks, that are then combined for the opening and closing filters, so let's start with them. Implement erosion and dilation before implement opening and closing, using the previous two filters.