Digital Pathology Approaches in Melanoma Care

Enrico Sarneri [a] ENRICO.SARNERI@ETU.SORBONNE-UNIVERSITE.FR

[a] *M2 Master Informatique, Sorbonne Université, France (FR)*

Abstract

The purpose of this work is to manipulate and play around with Whole Slide Images (WSI), which are microscopic slides of tissues stained with dyes that help us to visualize cellular components and conditions. The dyes used for these slides are called H&E (Hematoxylin and Eosin). Several steps have been faced in the image analysis procedure: semantic segmentation, feature extraction as well as cell detection and classification. Moreover, in order to facilitate such processes so that they get the best possible accuracy and quality, different pre-processing methods have been tackled. Among them we highlight image normalization, that aims to put images into a common statistical distribution in terms of size and pixel values, and minimization of ink artifacts and marks, which allows to produce cleaner clinical images. After that, the obtained results have been evaluated with different image quality metrics in order to see the before-after-effects of the applied techniques.

1. Introduction

1.1. WSI

In the past few decades, significant technologic improvements have led to the introduction of several digital image solutions in pathology. Among them, the adoption of Whole Slide Images (WSIs) in 1999, provided the possibility of “digitally converting the entire tissue on glass slide into a high-resolution Virtual Side (VS)”. In practice, the process of obtaining WSI consists of four main steps: image acquisition, storage, processing, and visualization. Image capture utilizes specialized digital scanners that capture sequential images either in a tiled or line-scanning manner, which are subsequently assembled or stitched into a VS, an exact replica of the glass slide. In this case, the digital slide scanners may vary in some of their features and capabilities, such as scanning capacity, object availability and image resolution. The second step instead harnesses specific software (VS Viewers) in order to visualize and/or analyze these enormous digital files. These viewers often provide the functionality to annotate the image and export to other file formats, and additionally, the viewing process itself as well as the managing of VS is driven by the intended use of the WSI system.

WSIs have been used for several clinical purposes, even though their validation is fundamental to ensure that diagnostic performance is at least equivalent to that of glass slides analyzed under conventional light microscopy. Telepathology has been one of the first uses of WSIs for primary diagnosis, consultation, and interpreting frozen sections. Furthermore, WSIs added advantages in enhancing objectivity in the interpretation of immunohistochemistry and electron microscopy used in tumor diagnosis, prognosis, and evaluation of biomarkers for targeted therapy. WSIs have also been used for remotely viewing immunostains, facilitating the preparation and conduct of tumor boards through avoiding the need of a multiheaded microscope, microscope with projection attachment or acquisition of multiple static images of a case. In addition, WSIs are frequently used for numerous nonclinical applications such as educational activities, offering several advantages over conventional glass slides since they are more interactive, instantaneously available to multiple remote users, can be easily annotated, and promote standardization of training materials.

However, WSIs still suffer from many unresolved issues that need to be addressed before they are applied in routine application across pathology. First of all, we have the cost of procurement, implementation, and operational cost that may be prohibitive, in particular for small pathology laboratories (initial cost of scanners, hidden costs of training of staff and pathologists, technical support, and digital slide storage systems). Secondly, WSIs in pathology do not reduce the laboratory’s workload since glass slides still need to be scanned. Other commonly encountered issues are the available bandwidth of the network at the pathologists’ workplace, installation of compatible browsers and security issues related to information technology. Furthermore, we must keep in mind that WSIs would be as good as the original glass slides, in this direction we must concern about pre-imaging steps such as tissue collection, handling, fixation, processing, sectioning, and staining/labelling. For example, uneven tissue staining can lead to significant variations in pattern recognition of image analysis and in the tissue segmentation process. Another limitation is the fact that massive data storage capacity is required. Consequently, WSI systems utilize image compression algorithms to reduce the file size, introducing image artifacts. Finally, the quality of the digital image may be compromised by prior ink markings, extra or wet mounting media, plastic coverslips, or thick slide labels. In the current work we are going to face this last problem of image quality, trying to enhance it with the removal of artifact, in order to present the highest-quality digital images possible.

1.2. Deep Learning and Digital Pathology

Recent developments in software and hardware and the exponential growth of computerized approaches have led deep learning (DL) and deep neural networks (DNNs) to reach the state-of-the-art in the field of medical image analysis. In particular, the combination between these algorithms and WSIs can perform histopathological assessment of large tissue sections in a significantly more time-efficient way, and it will make tissue evaluation more sensitive and more specific.

Due to the increase of both dataset sizes and computing power, the most common DNNs used for image analysis are Convolution Neural Networks (CNNs), which apply a filter to the image and feeding that filtered output to the next layer. These filters are applied to an image by performing a convolution operation of the filter with the image. This will slide the filter across the pixels of the image, evaluating the product of the filter parameters with the image. In such a case, a model is not applied to every pixel, but rather the filter parameters for each filter are shared across a neighborhood of pixels, reducing the size of the image, and saving memory.

The main elements of a DL models are essentially three:

- **The input of the model** – Input size influences the architecture of the model since filtering may change the size of images.
- **The architecture of the model –** It constitutes the number of filters, of layers, and the overall connections from input to output.
- **The output of the model –** Usually, it is a classification of the input into one or more classes or a segmentation of the image into separated classes.

Additionally, the problem of such models is converted to an optimization problem by defining a loss function which quantifies how good or bad the fit was. This loss function is identified based on the endpoint being used and this is then optimized using an iterative approach. For what concerns the development and testing of DL algorithms, the entire dataset is split into training, validation, and test sets. The optimization of the network is given by the training and validation set, while the test set is kept equal. The parameter settings are optimized iteratively, and at the same time, the network aims to achieve better performance on the training set. Once the performance of the model on the training and validation sets is optimized, the training procedure is stopped in order to avoid overfitting. This is caused by the fact that the majority of CNNs take as input millions of parameters.

More specifically, DL approach has been extensively adopted in digital pathology. This will lighten the load of work associated to pathologists that is often characterized by an attentive analysis of a large number of slides and issues that bring a very high inter and intra operator variability. Due to these factors, over the years numerous DNNs have been developed to process the digitized histology slides in the three main computer vision tasks: classification, detection, and segmentation. However, if from one side these approaches have brought several advantages, they still suffer from different issues related to input preparation of such networks.

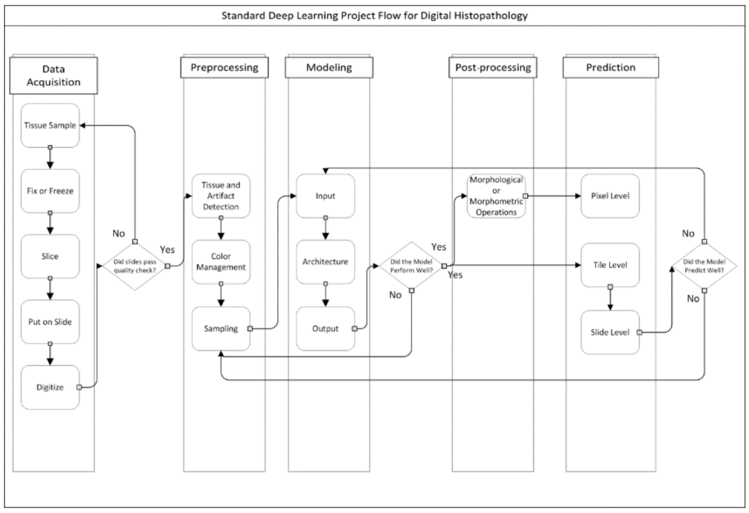

Standard Deep Learning Project Flow for Digital Histopathology
Data Acquisition
Preprocessing
Modeling
Post-processing
Prediction
Tissue Sample
Fix or Freeze
Slice
Put on Slide
Digitize
No
Yes
Did slides pass quality check?
Tissue and Artifact Detection
Color Management
Sampling
Input
Architecture
Output
Did the Model Perform Well?
Yes
No
Yes
Morphological or Morphometric Operations
Pixel Level
Tile Level
Slide Level
No
Did the Model Predict Well?
No

Figure 1: *Flow of a WSI pipeline* **(Byron Smith, 2020)**.

More recently, numerous DNNs have been integrated with pre-processing techniques, that aim to optimally prepare the input for the network, allowing a stronger increase in the performances when compared to the network by itself. The phase of obtaining digitalized histological images must follows a chain of steps, which are usually carried out manually by technicians. The histological tissue should be processed to preserve its internal architecture and to showcase an appearance like its aspect inside the living organism. In ***Figure 2****:* Sequential steps that must be applied to a biological sample in *order to produce the corresponding histological slide. During these steps several artifacts can be generated* . we can see the different steps applied to histological samples in order to obtain the corresponding digital image.

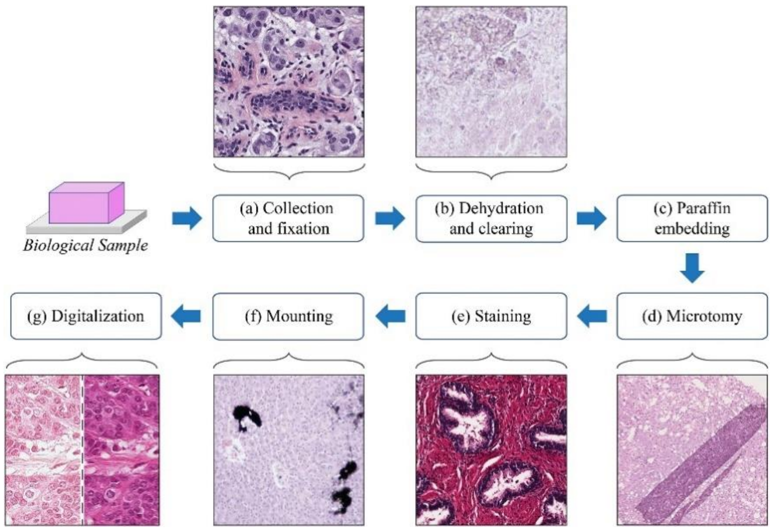


Figure 2: *Sequential steps that must be applied to a biological sample in order to produce the corresponding histological slide. During these steps several artifacts can be generated* **(Massimo Salvi, 2021).**

All these steps can generate different artifacts that can lower the quality of the histological image. For example, during staining or digitalization processes there could be significant color variations related to the length of time the sample spends in contact with the dyes, or to the fact that different scanning platforms or acquisition technology may be used or even to the alignment of the sample with the focal plane of the scanner. Color variability in this case can limit the interpretation of images by pathologists and it can also dramatically affect the results and performances of DL image analysis algorithms. Furthermore, during dehydration, water drops could make a histological sample opaque, during microtomy an imprecise placing of the tissue sample on the microscope slide could cause a folding of the tissue, and during mounting coverslip placement can generate artifacts such as the presence of dust or air bubbles.

1.3. Pre-processing

Complete standardization of these processes is needed, but it cannot unfortunately be achieved due to some manual sectioning variability and stains fading over time. Since these challenges require to be faced, specific pre-processing steps are mandatory to train a robust DL model. The aim of this project, is to provide an overview on some typologies of pre-processing techniques showing their application and results to different examples of digital histological samples.

The term "pre-processing" refers to all the techniques applied to the raw data in order to optimally prepare the network input so as to obtain a more robust and accurate final model. The general purpose of such steps is to discard data that is not informative or useful, to create a consistent dataset and to enable processing on WSI during modeling. In this direction, we can highlight three main groups of pre-processing methods:

- **Tissue and artifact detection** – Necessary to utilize resources in the most efficient way avoiding generation of noise.
- **Stain color normalization** – Used to reduce disturbing sources of image variability.
- **Patch selection** – Achieved by tissue sampling, dividing images into tiles. This is necessary since DL models are computationally expensive and they cannot process the entire WSIs that are enormous in terms of gigapixels.

2. WSIs used

At the beginning of the work, three different BioImagene Image Files (BIF) were provided with the following characteristics:

Images:	**Size (KB):**	**Uncompressed size (GB):**	**Pixel width:**	**Pixel height:**	**Image type:**	**Pixel type:**
PYRUVATE PS_B1700773.bif	365.097 KB	2.7 GB	12423.97 µm	36177 µm	Brightfield (H&E)	Uint8 (rgb)
PYRUVATE PS_B1701293.bif	1.310.126 KB	9 GB	36332.31 µm	19219.38 µm	Brightfield (H&E)	Uint8 (rgb)
PYRUVATE_PS_B1701352.bif	655.780 KB	4.5 GB	23042.15 µm	14999.97 µm	Brightfield (H&E)	Uint8 (rgb)

BIF format is the one used by Roche Tissue Diagnostics Digital Pathology scanner. With this format, the images can have up to 200.000 x 200.000 pixels and a 40x magnification. Also considering the possibility of volumetric scans with multiple focus layers, the WSIs can easily break the 4 GB size limit imposed by the use of 32-bit file pointers defined by the TIFF-standard, even when compressed (Gasuad). More info can be found in (Gasuad).

Consequently, in order to use a more portable and standardized format, it has been decided to adopt the TIFF-standard and work on such files. As first attempt, a conversion of BIF Images with QuPath Image Exportation has been tried. In the "Export Image" section, "Original Pixel" → "OME TIFF" option has been chosen with a down sample factor of 1.0. In this way we write the image as an OME-TIFF image using Bio-Formats preserving image metadata (Melissa Linkert, 2010). With any other export format (TIFF-ImageJ, Zip-ImageJ, PNG, JPEG) we'll have to use a down sample factor in order to respect the respective constraints, losing quality as well as metadata. Another way to get around this issue is to scan the WSIs directly into TIF format from the scanner instead of formatting it into BIF.

To demonstrate that, original TIF scanned images have been then provided:

Images:	**Size (KB):**	**Uncompressed size (GB):**	**Pixel width:**	**Pixel height:**	**Image type:**	**Pixel type:**
Pyruvate_serie3_B1700726.tif	1.286.956 KB	9.4 GB	39335.28 µm	18488.40 µm	Brightfield (H&E)	Uint8 (rgb)
Pyruvate_serie3_B1700828.tif	296.572 KB	2 GB	11926.32 µm	13183.68 µm	Brightfield (H&E)	Uint8 (rgb)
Pyruvate_serie3_B1701244.tif	1.290.192 KB	9.4 GB	39342.72 µm	18480.96 µm	Brightfield (H&E)	Uint8 (rgb)
Pyruvate_serie3_B1701264.tif	1.079.591 KB	7.5 GB	25615.92 µm	22721.76 µm	Brightfield (H&E)	Uint8 (rgb)
Pyruvate_serie3_B1701273.tif	1.215.042 KB	9.5 GB	33137.76 µm	22111.68 µm	Brightfield (H&E)	Uint8 (rgb)
Pyruvate_serie3_B1701381.tif	796.881 KB	5.8 GB	20311.20 µm	22141.44 µm	Brightfield (H&E)	Uint8 (rgb)
Pyruvate_serie3_B1800048.tif	228.729 KB	1.6 GB	12365.28 µm	10155.60 µm	Brightfield (H&E)	Uint8 (rgb)
Pyruvate_serie3_B1900657.tif	910.212 KB	6.8 GB	28279.44 µm	18518.16 µm	Brightfield (H&E)	Uint8 (rgb)

However, even though the TIF format is fairly usual and can be visually displayed by default by common applications and operating systems, we could still face the problems of the size of the data with the previous dataset due to computational limitations. This is the reason why the images could be then converted to smaller images, reformulating the big data problem as a small data problem. Thus, the width and the height of images could be shrunk for example by a factor of 32x, converting the files into .PNG files at 1/32 scale. Fortunately, the load of images in this case is not so high (just few dozens of GBs), so we proceeded with the .TIF files. In order to load, display and save images, the Pillow package has been used.

```python
def open_image(filename):
  """

  returns:
    A PIL.Image.Image object representing an image.
  """

  image = Image.open(filename)
  return image
```

In order to handle the images, the NumPy arrays have been used.

```
def pil_to_np_rgb(pil_img):
  """
  Convert a PIL Image to a NumPy array.

  Args:
    pil_img: The PIL Image.

  """
  rgb = np.asarray(pil_img)
  return rgb
```

This function will convert a PIL Image into a 3-dimensional NumPy array in RGB format.

3. Artifacts Removal

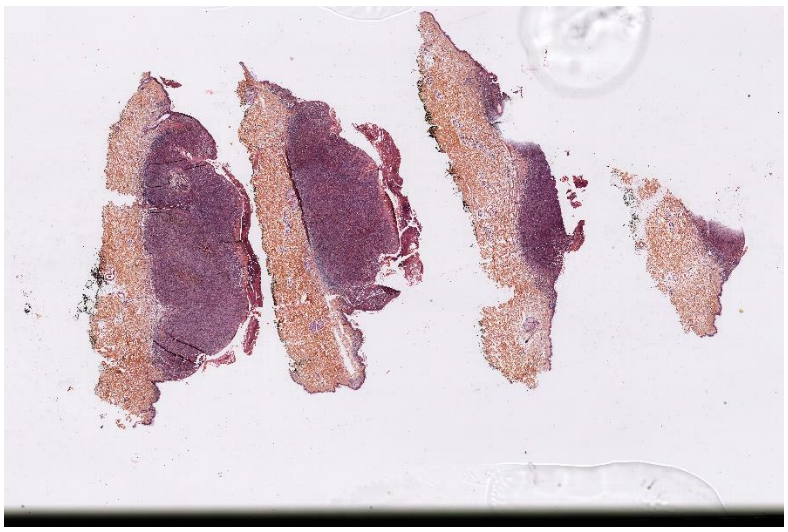
In this section we'll investigate the topic of the artifact removal, achieved with image filtering. The process for doing that consists in mainly masking out non-tissue portions setting their pixels to 0 for the 3 RGB channels. We will use different kind of masks applied both singularly and together with AND condition, so as to obtain more accurate results. A presentation of them follows below.

3.1. Thresholding

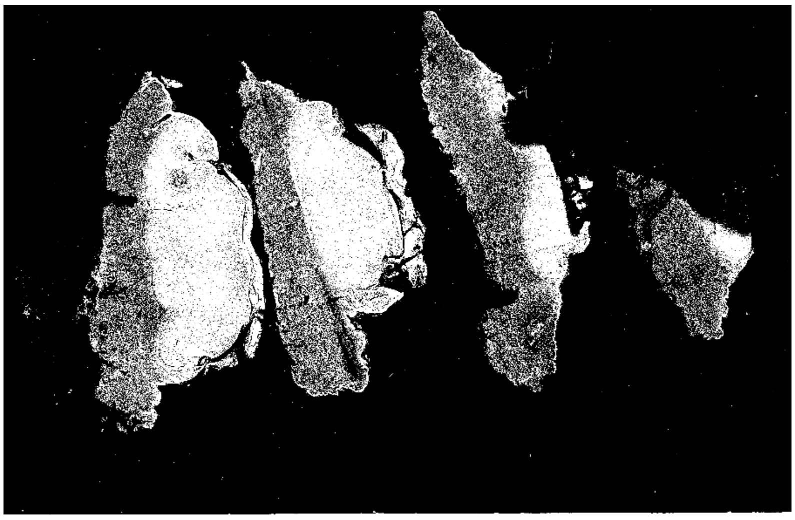
The concept of thresholding is really common in image processing since it allows to detect if a certain pixel is above a particular value. In our case it's common that the slide background is illuminated by white light, this means that the value of the correspondent pixel is close to 255. To work better having a clearer image it useful to have those values close to 0.

3.1.1. Basic Threshold

Through the basic threshold we generate a binary image in which each value of the image array is a true if the corresponding pixel is above a certain threshold, false otherwise.



If we take as example *Pyruvate_serie3_B1701273.tif* we can see that at the bottom center there's a black horizonal line probably caused by some illumination o disposition problem,

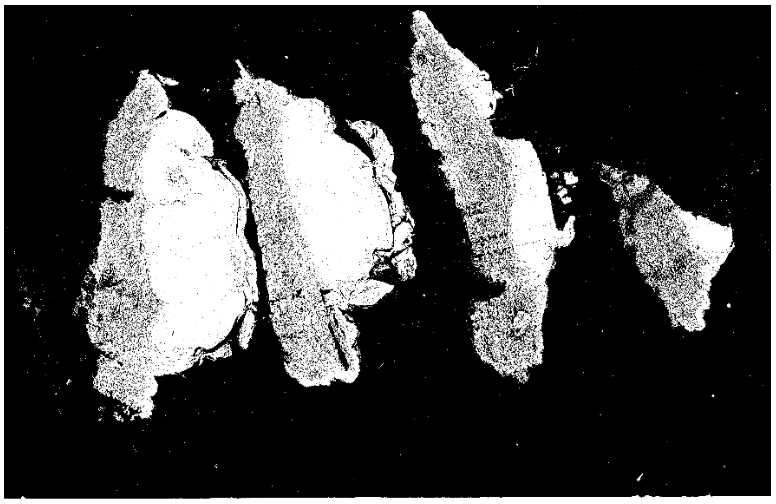


Then, if we use the basic filter with a threshold of 100, we can see that those bottom line is completely highlighted in white and for example also the little gray portions are eliminated since all the pixel values that were above 100 are shown in white, while those that were above 100 are shown in black.

```
def filter_threshold(np_img, threshold, output_type="bool"):
  """
  Return mask where a pixel has a value if it exceeds the threshold value.

  Args:
    np_img: Binary image as a NumPy array.
    threshold: The threshold value to exceed.
    output_type: Type of array to return (bool, float, or uint8).

  Returns:
    NumPy array representing a mask where a pixel has a value (T, 1.0, or 255) if the
corresponding input array
    pixel exceeds the threshold value.
  """
  result = (np_img > threshold)
  if output_type == "bool":
    pass
  elif output_type == "float":
    result = result.astype(float)
  else:
    result = result.astype("uint8") * 255
  return result
```

3.1.2. Hysteresis Threshold

Through the hysteresis threshold we are able to control two levels of thresholding. In this context, areas above some low thresholds are considered to be above the threshold if they are also connected to areas above a higher, more stringent threshold. They can be seen as continuations of these high-confidence areas. The right values for the top and bottom thresholds can be found through experimentation.

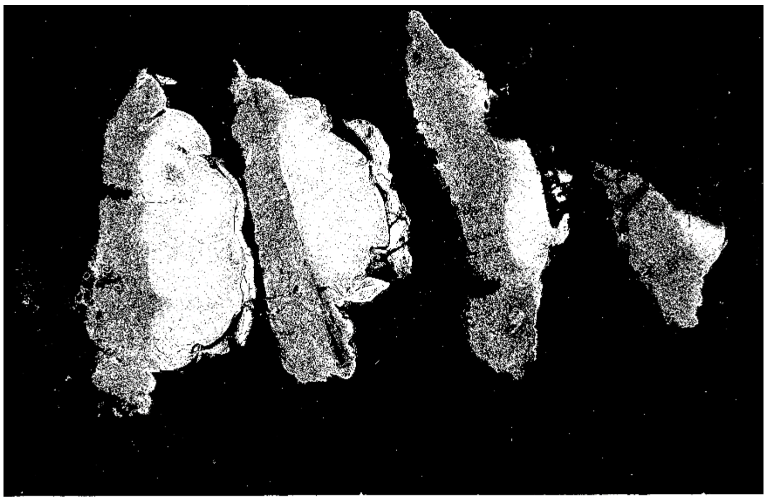


In this case as parameters of threshold have been used 50 (low) and 100(high).

```
def filter_hysteresis_threshold(np_img, low=50, high=100, output_type="uint8"):
  """
  Apply two-level (hysteresis) threshold to an image as a NumPy array, returning a binary image.

  Args:
    np_img: Image as a NumPy array.
    low: Low threshold.
    high: High threshold.
    output_type: Type of array to return (bool, float, or uint8).

  Returns:
    NumPy array (bool, float, or uint8) where True, 1.0, and 255 represent a pixel above
hysteresis threshold.
  """
  hyst = sk_filters.apply_hysteresis_threshold(np_img, low, high)
  if output_type == "bool":
    pass
  elif output_type == "float":
    hyst = hyst.astype(float)
  else:
    hyst = (255 * hyst).astype("uint8")
  return hyst
```

3.1.3. Otsu Threshold

Through Otsu thresholding we are able to perform automatic thresholding. It simply processes the image histogram, segmenting the objects by minimization of the variance on each of the classes (background and foreground). It produces appropriate results for bimodal images. So, we basically process the input image, we obtain its histogram, we compute the threshold value *T* and then we replace image pixels into those regions where saturation is greater than *T* and into the black in the opposite case.



The result is roughly similar as hysteresis thresholding, and we can see as the bottom center shadow area is passed through the filter in a similar way as hysteresis thresholding. Moreover, the slides here considered don’t have high heightened shadow areas, but it would be challenging to process those kinds of images with shadow area as background.

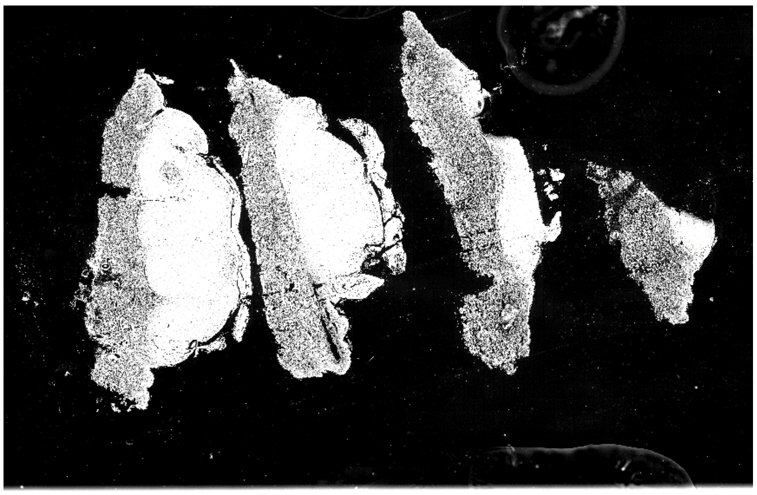
```
def filter_otsu_threshold(np_img, output_type="uint8"):
  """
  Compute Otsu threshold on image as a NumPy array and return binary image based on pixels above
threshold.

  Args:
    np_img: Image as a NumPy array.
    output_type: Type of array to return (bool, float, or uint8).

  Returns:
    NumPy array (bool, float, or uint8) where True, 1.0, and 255 represent a pixel above Otsu
threshold.
  """
  otsu_thresh_value = sk_filters.threshold_otsu(np_img)
  otsu = (np_img > otsu_thresh_value)
  if output_type == "bool":
    pass
  elif output_type == "float":
    otsu = otsu.astype(float)
  else:
    otsu = otsu.astype("uint8") * 255
  return otsu
```

3.2. Contrast

Another important metric that we can apply is the contrast. Contrast measures the difference in intensities of pixel with respect to their possible range values. If an image has a low contrast, it means that probably it’s dull and the details are hidden. From the other side, if the contrast is high, it means that it’s sharp and the details are more clearly visible. If we increase the contrast in an image, we can bring out different details.

3.2.1. Contrast Stretching

Contrast stretching consists in “stretching” the range of intensity values the image contains to span a desired range values. For example, if we have that all the intensities in an image occur between 50 and 100 (on a scale from 0 to 255), if we rescale the intensities so that 50 now corresponds to 0 and 100 corresponds to 255 and we linearly rescale the intensities between these values, we have now increased the contrast in the image.

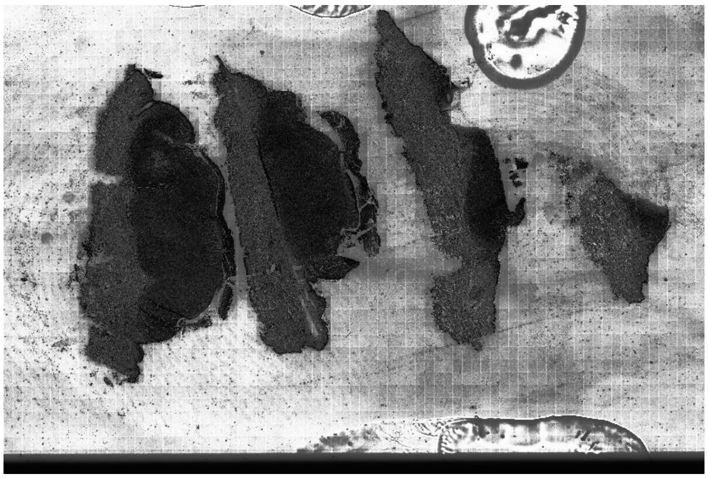


In this case, if we use a low pixel value of 100 and a high value of 200, we can visually inspect details in the range from 100 to 200 thanks to the spread of this range across the full spectrum by the contrast filter.

```
def filter_contrast_stretch(np_img, low=100, high=200):
  """
  Filter image (gray or RGB) using contrast stretching to increase contrast in image based on the
intensities in
  a specified range.

  Args:
    np_img: Image as a NumPy array (gray or RGB).
    low: Range low value (0 to 255).
    high: Range high value (0 to 255).

  Returns:
    Image as NumPy array with contrast enhanced.
  """
  low_p, high_p = np.percentile(np_img, (low * 100 / 255, high * 100 / 255))
  contrast_stretch = sk_exposure.rescale_intensity(np_img, in_range=(low_p, high_p))
  return contrast_stretch
```

3.2.2. Histogram Equalization

Histogram equalization allow us to improve contrast in images by effectively spreading out the most frequent intensity values (stretching out the intensity range of the image). It usually increases the global contrast of images when its usable data is represented by close contrast values. Unlike the previous case, which has a linear distribution, here the transformation is based on probabilities and is non-linear.

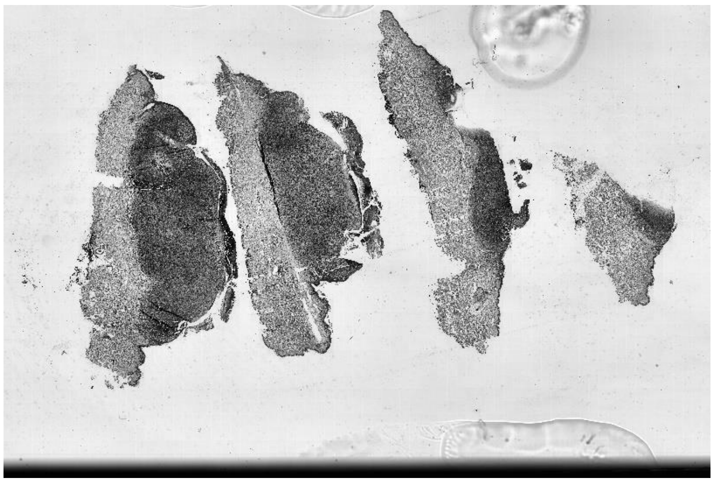


```
def filter_histogram_equalization(np_img, nbins=256, output_type="uint8"):
  """
  Filter image (gray or RGB) using histogram equalization to increase contrast in image.

  Args:
    np_img: Image as a NumPy array (gray or RGB).
    nbins: Number of histogram bins.
    output_type: Type of array to return (float or uint8).

  Returns:
     NumPy array (float or uint8) with contrast enhanced by histogram equalization.
  """
  if np_img.dtype == "uint8" and nbins != 256:
    np_img = np_img / 255
  hist_equ = sk_exposure.equalize_hist(np_img, nbins=nbins)
  if output_type == "float":
    pass
  else:
    hist_equ = (hist_equ * 255).astype("uint8")
  return hist_equ
```

3.2.3. Adaptive Equalization and Contrast Limited Adaptive Equalization

Differently from histogram equalization, adaptive equalization computes several histograms, each corresponding to a distinct section of the image, and the uses them to redistribute the lightness values of the image. It's convenient if we want to improve the local contrast and enhancing the definitions of edges in each region of an image.

Contrast limited adaptive equalization from the other side, applies the contrast limiting procedure to each neighborhood from which a transformation function is derived. It is used in order to prevent the amplification of noise caused by the previous equalization.



```
def filter_adaptive_equalization(np_img, nbins=256, clip_limit=0.01, output_type="uint8"):
  """
  Filter image (gray or RGB) using adaptive equalization to increase contrast in image, where
contrast in local regions
  is enhanced.

  Args:
    np_img: Image as a NumPy array (gray or RGB).
    nbins: Number of histogram bins.
    clip_limit: Clipping limit where higher value increases contrast.
    output_type: Type of array to return (float or uint8).

  Returns:
     NumPy array (float or uint8) with contrast enhanced by adaptive equalization.
  """
  adapt_equ = sk_exposure.equalize_adapthist(np_img, nbins=nbins, clip_limit=clip_limit)
  if output_type == "float":
    pass
  else:
    adapt_equ = (adapt_equ * 255).astype("uint8")
  return adapt_equ
```

3.3. Color Components

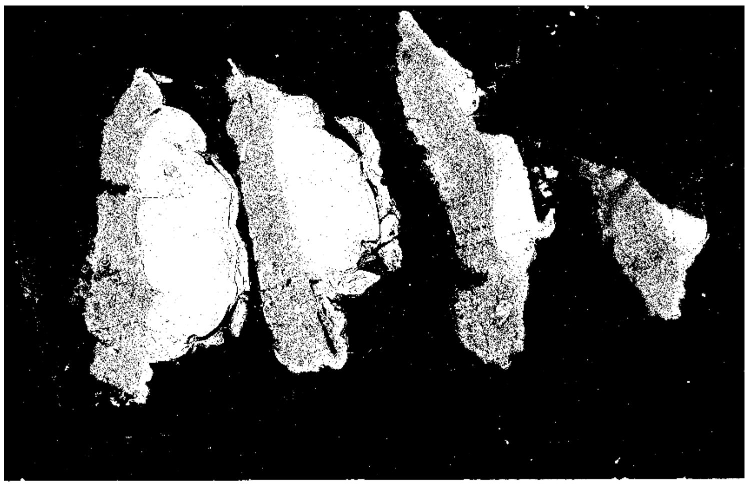
As cited at the beginning, the images we work with are all H&E stained. In general, this means that Hematoxylin will stain acidic structures as DNA or RNA with purple tonality while Eosin will stain basic structures as cytoplasm with pink tonality. As a consequence, cells are going to be stained pink and nuclei tend to be stained purple. However, the specific color often depends on the types of cells that are stained, and the quantity of stain utilized. The above-mentioned differentiation is just a general guideline.

Furthermore, the main problem is the fact that the slides may have been marked with colored inks and pens. Hereafter we are going to illustrate different filters to exclude red, green and blue colors, since they don't represent any tissue portions, the only one we are interested in.

However, different challenges are presented in front of us trying to apply such filters. First of all, they need to be general appliable to the images in the dataset and they should manage the variations in shadows and lights. Then, some variations have to be faced: the amount of H&E staining is variable from slide to slide and the pen mark colors vary due to lighting and pen marks over tissue. Finally, there can be overlaps of color between stained tissue and pen marks. In this case we need to balance the intensity of how strong pen colors are exclusively filtered and stain colors are inclusively filtered.

3.3.1. Green Channel Filter

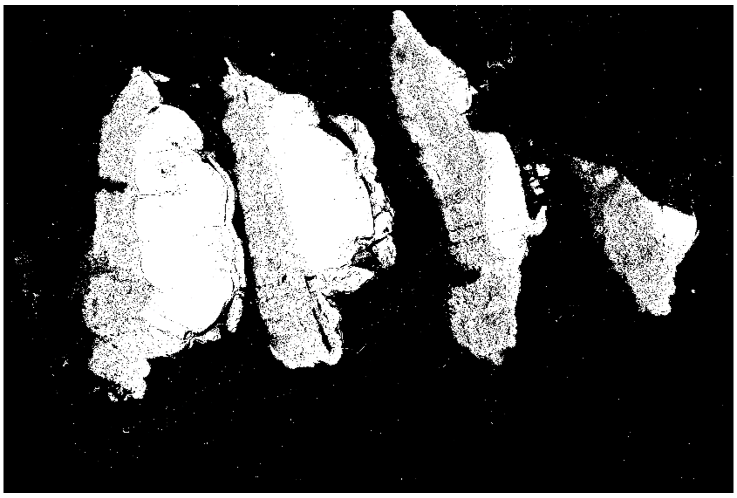
We know that on the RGB wheel, pink and purple colors are on the opposite side w.r.t green, in fact filtering out pixels that have a high value of green can be one way to filter out those parts of WSIs where pink o purple are not present. These parts also include white background since white color has a high green channel value together with blue and red channel values. In this case, the filter can be used in order to separate the tissue from the background.



```
def filter_green_channel(np_img, green_thresh=200, avoid_overmask=True, overmask_thresh=90, 
output_type="bool"):
  """
  Create a mask to filter out pixels with a green channel value greater than a particular 
threshold, since hematoxylin
  and eosin are purplish and pinkish, which do not have much green to them.

  Args:
    np_img: RGB image as a NumPy array.
    green_thresh: Green channel threshold value (0 to 255). If value is greater than 
green_thresh, mask out pixel.
    avoid_overmask: If True, avoid masking above the overmask_thresh percentage.
    overmask_thresh: If avoid_overmask is True, avoid masking above this threshold percentage 
value.
    output_type: Type of array to return (bool, float, or uint8).

  Returns:
    NumPy array representing a mask where pixels above a particular green channel threshold have 
been masked out.
  """
  g = np_img[:, :, 1]
  gr_ch_mask = (g < green_thresh) & (g > 0)
  mask_percentage = mask_percent(gr_ch_mask)
  if (mask_percentage >= overmask_thresh) and (green_thresh < 255) and (avoid_overmask is True):
    new_green_thresh = math.ceil((255 - green_thresh) / 2 + green_thresh)
    print(
      "Mask percentage %3.2f%% >= overmask threshold %3.2f%% for Remove Green Channel
green_thresh=%d, so try %d" % (
        mask_percentage, overmask_thresh, green_thresh, new_green_thresh))
    gr_ch_mask = filter_green_channel(np_img, new_green_thresh, avoid_overmask, overmask_thresh,
output_type)
  np_img = gr_ch_mask

  if output_type == "bool":
    pass
  elif output_type == "float":
    np_img = np_img.astype(float)
  else:
    np_img = np_img.astype("uint8") * 255
  return np_img
```

3.3.2. Grays Filter

This filter is able to filter out pixels that have red, blue, and green value that are within a certain tolerance of each other in order to remove shadow areas in the slides (which consist in just gradient of dark-to-light-grays).



```
def filter_grays(rgb, tolerance=15, output_type="bool"):
  """
  Create a mask to filter out pixels where the red, green, and blue channel values are similar.

  Args:
    np_img: RGB image as a NumPy array.
    tolerance: Tolerance value to determine how similar the values must be in order to be
filtered out
    output_type: Type of array to return (bool, float, or uint8).
  Returns:
    NumPy array representing a mask where pixels with similar red, green, and blue values have
been masked out.
  """
  (h, w, c) = rgb.shape

  rgb = rgb.astype(np.int)
  rg_diff = abs(rgb[:, :, 0] - rgb[:, :, 1]) <= tolerance
  rb_diff = abs(rgb[:, :, 0] - rgb[:, :, 2]) <= tolerance
  gb_diff = abs(rgb[:, :, 1] - rgb[:, :, 2]) <= tolerance
  result = ~(rg_diff & rb_diff & gb_diff)

  if output_type == "bool":
    pass
  elif output_type == "float":
    result = result.astype(float)
  else:
    result = result.astype("uint8") * 255
  return result
```

3.3.3. Red Pen Filter

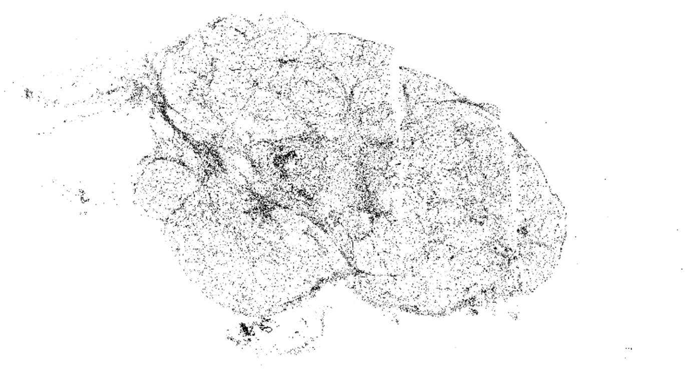
In this case our aim is to filter out all the pen marks characterized by more shades of red. We will use a function able to filter out reddish color, where the mask is based on a pixel being above a red channel threshold value, below a green channel threshold value, and below a blue channel threshold value.

```
def filter_red(rgb, red_lower_thresh, green_upper_thresh, blue_upper_thresh, output_type="bool",
               display_np_info=False):
  """
  Args:
    rgb: RGB image as a NumPy array.
    red_lower_thresh: Red channel lower threshold value.
    green_upper_thresh: Green channel upper threshold value.
    blue_upper_thresh: Blue channel upper threshold value.
    output_type: Type of array to return (bool, float, or uint8).
    display_np_info: If True, display NumPy array info and filter time.

  Returns:
    NumPy array representing the mask.
  """
  if display_np_info:
    t = Time()
  r = rgb[:, :, 0] > red_lower_thresh
  g = rgb[:, :, 1] < green_upper_thresh
  b = rgb[:, :, 2] < blue_upper_thresh
  result = ~(r & g & b)
  if output_type == "bool":
    pass
  elif output_type == "float":
    result = result.astype(float)
  else:
    result = result.astype("uint8") * 255
  return result
```

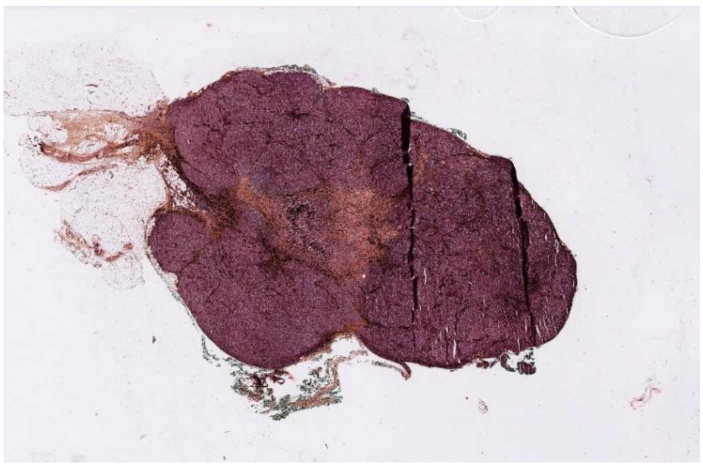
We will combine multiple sets of this function threshold values using “and” operator. For example:

```
result = filter_red(rgb, red_lower_thresh=150, green_upper_thresh=80, blue_upper_thresh=90) & \
           filter_red(rgb, red_lower_thresh=110, green_upper_thresh=20, blue_upper_thresh=30) & \
           filter_red(rgb, red_lower_thresh=185, green_upper_thresh=65, blue_upper_thresh=105) &
\
           filter_red(rgb, red_lower_thresh=195, green_upper_thresh=85, blue_upper_thresh=125)
```

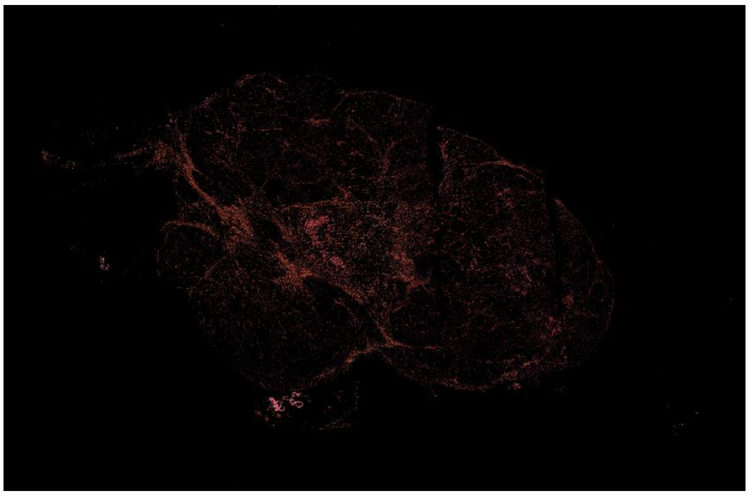
We also have to keep particular attention to the fact that pinkish shades of red are similar to eosin staining.



This image represents the red filter resulting from the parameters applied and we can see as it is much more inclusive in terms of the shades of red accepted. In this case also some of the pinkish red from eosin-stained tissue is also included as a result of this aggressive filtering.



This is image represents the image without the red pen marks.



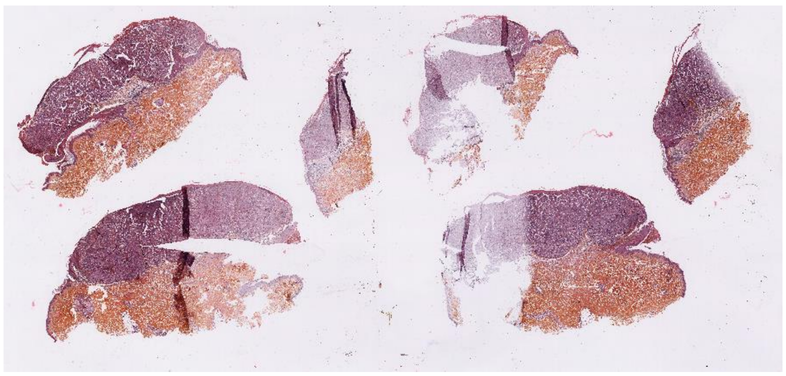
This image displays the mask with all the red shades captured.

3.3.4. Blue Pen Filter

The same approach can be applied for blue pen inks. We will use a function able to filter out blue colors. It takes a red channel upper threshold value, a green channel upper threshold value, and a blue channel lower threshold value. The generated mask is based on a pixel being below the red channel threshold value, below the green channel threshold value, and above the blue channel threshold value.

```
def filter_blue(rgb, red_upper_thresh, green_upper_thresh, blue_lower_thresh, output_type="bool",
                display_np_info=False):
  Args:
    rgb: RGB image as a NumPy array.
    red_upper_thresh: Red channel upper threshold value.
    green_upper_thresh: Green channel upper threshold value.
    blue_lower_thresh: Blue channel lower threshold value.
    output_type: Type of array to return (bool, float, or uint8).
    display_np_info: If True, display NumPy array info and filter time.

  Returns:
    NumPy array representing the mask.
  """
  if display_np_info:
    t = Time()
  r = rgb[:, :, 0] < red_upper_thresh
  g = rgb[:, :, 1] < green_upper_thresh
  b = rgb[:, :, 2] > blue_lower_thresh
  result = ~(r & g & b)
  if output_type == "bool":
    pass
  elif output_type == "float":
    result = result.astype(float)
  else:
    result = result.astype("uint8") * 255
  if display_np_info:
    util.np_info(result, "Filter Blue", t.elapsed())
  return result
```
We will combine multiple sets of this function threshold values using “and” operator. For example:

```
result = filter_blue(rgb, red_upper_thresh=60, green_upper_thresh=120, blue_lower_thresh=190) & \
           filter_blue(rgb, red_upper_thresh=120, green_upper_thresh=170, blue_lower_thresh=200)
& \
           filter_blue(rgb, red_upper_thresh=175, green_upper_thresh=210, blue_lower_thresh=230)
& \
           filter_blue(rgb, red_upper_thresh=145, green_upper_thresh=180, blue_lower_thresh=210)
```



In our case, we were lucky since inside our dataset there were not images with blue pen marks, in fact as we can see here in this example the mask doesn’t capture any track of color. But it could be still useful for future cases.

3.3.5. Green Pen Filter

Still the same approach can be applied for green pen marks. We will use a function able to filter green color shades. The mask is based on a pixel being below a red channel threshold value, above a green channel threshold value, and above a blue channel threshold value. Note that for the green ink, the green and blue channels tend to track together, so we use a blue channel lower threshold value rather than a blue channel upper threshold value.

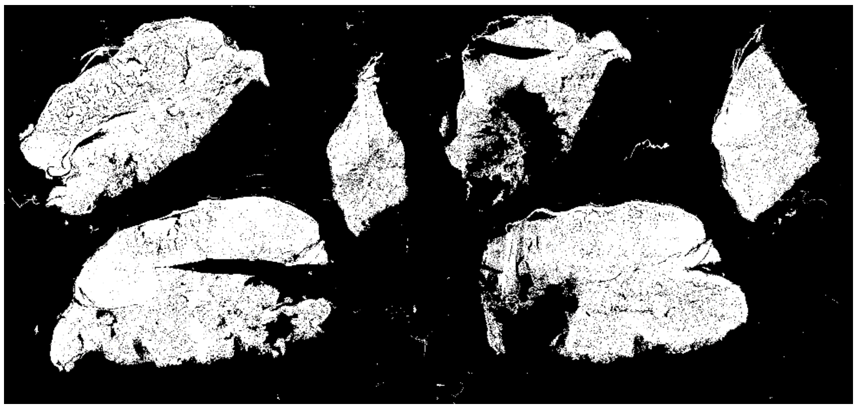
```
def filter_green(rgb, red_upper_thresh, green_lower_thresh, blue_lower_thresh,
output_type="bool",
                 display_np_info=False):
  """
  Args:
    rgb: RGB image as a NumPy array.
    red_upper_thresh: Red channel upper threshold value.
    green_lower_thresh: Green channel lower threshold value.
    blue_lower_thresh: Blue channel lower threshold value.
    output_type: Type of array to return (bool, float, or uint8).
    display_np_info: If True, display NumPy array info and filter time.

  Returns:
    NumPy array representing the mask.
  """
  if display_np_info:
    t = Time()
  r = rgb[:, :, 0] < red_upper_thresh
  g = rgb[:, :, 1] > green_lower_thresh
  b = rgb[:, :, 2] > blue_lower_thresh
  result = ~(r & g & b)
  if output_type == "bool":
    pass
  elif output_type == "float":
    result = result.astype(float)
  else:
    result = result.astype("uint8") * 255
  return result
```

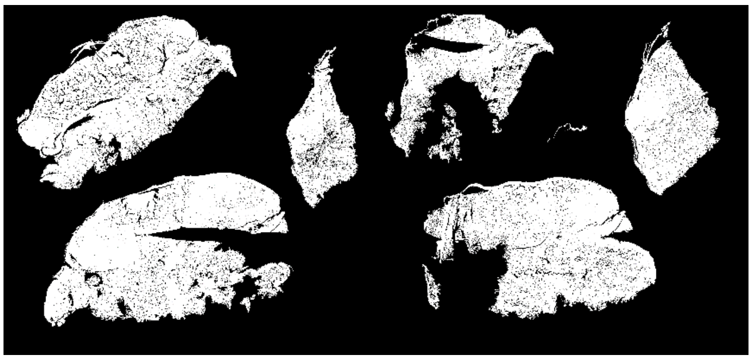
To handle different shades of green, we will combine different functions with different parameters as here:

```
result = filter_green(rgb, red_upper_thresh=150, green_lower_thresh=160, blue_lower_thresh=140) &
\
           filter_green(rgb, red_upper_thresh=70, green_lower_thresh=110, blue_lower_thresh=110)
& \
           filter_green(rgb, red_upper_thresh=45, green_lower_thresh=115, blue_lower_thresh=100)
& \
           filter_green(rgb, red_upper_thresh=30, green_lower_thresh=75, blue_lower_thresh=60)
```

3.4. Morphology

Thanks to mathematical morphology, we are able to analyze and process the geometrical structures, based on set theory, lattice theory, topology, and random functions. Mathematical morphology is also the foundation of morphological image processing, which consists of a set of operators that transform images according to the above characterizations. In our case, the morphological operator we used is the removal of small objects, able to remove objects less than a particular minimum size. This can be useful for removing small islands of noise from images.



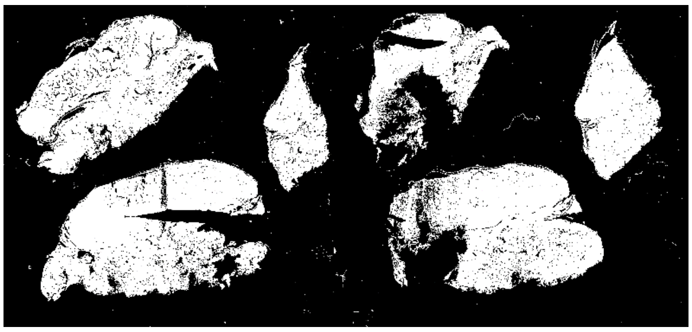


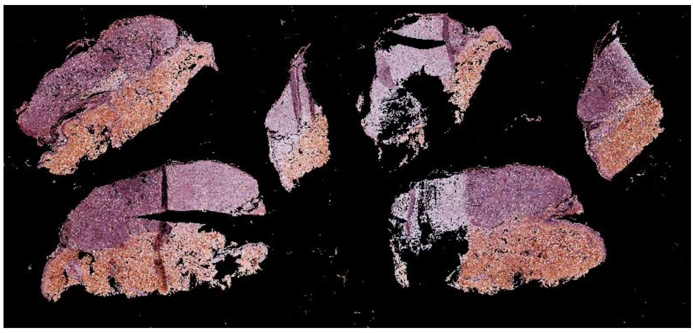
```
def filter_remove_small_objects(np_img, min_size=3000, avoid_overmask=True, overmask_thresh=95,
output_type="uint8"):
  """
  Args:
    np_img: Image as a NumPy array of type bool.
    min_size: Minimum size of small object to remove.
    avoid_overmask: If True, avoid masking above the overmask_thresh percentage.
    overmask_thresh: If avoid_overmask is True, avoid masking above this threshold percentage
value.
    output_type: Type of array to return (bool, float, or uint8).

  Returns:
    NumPy array (bool, float, or uint8).
  """
  rem_sm = np_img.astype(bool)  # make sure mask is boolean
  rem_sm = sk_morphology.remove_small_objects(rem_sm, min_size=min_size)
  mask_percentage = mask_percent(rem_sm)
  if (mask_percentage >= overmask_thresh) and (min_size >= 1) and (avoid_overmask is True):
    new_min_size = min_size / 2
    print("Mask percentage %3.2f%% >= overmask threshold %3.2f%% for Remove Small Objs size %d,
so try %d" % (
      mask_percentage, overmask_thresh, min_size, new_min_size))
    rem_sm = filter_remove_small_objects(np_img, new_min_size, avoid_overmask, overmask_thresh,
output_type)
  np_img = rem_sm

  if output_type == "bool":
    pass
  elif output_type == "float":
    np_img = np_img.astype(float)
  else:
    np_img = np_img.astype("uint8") * 255
  return np_img
```

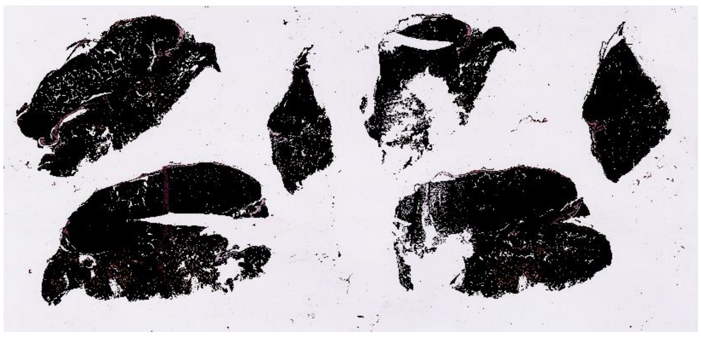
Here, if avoid_overmask is True, the function can recursively call itself with progressively smaller minimum size objects to remove to reduce the amount of masking that this filter performs.

3.5. Entropy

The entropy filter allows us to filter images based on complexity. Areas as slide backgrounds are less complex than area of interest such as cell nuclei, in this way filtering basing on entropy offers the interesting possibility to identify tissue.





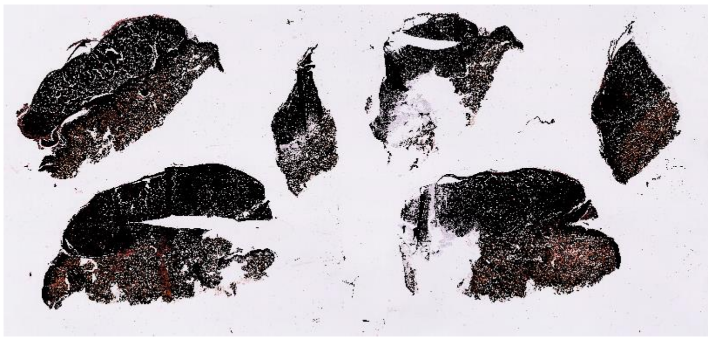


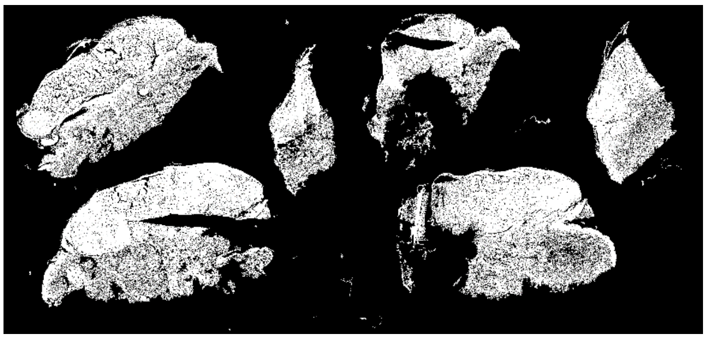
The first image represents the entropy filter applied, the second one is the original image with entropy mask and the last one is characterized by the inverse of the entropy mask. We can notice as much of the white background including the different shadow areas, are removed. In addition, for stained regions, a significant amount of the pink-eosin-stained area has been filtered out while a smaller proportion of the purple-stained hematoxylin area has been filtered out. This is in line with the fact that hematoxylin stains regions such as cell nuclei, which are structures with a significant complexity. As a consequence, this filter is a strong tool that could be used to identify regions of interest where mitoses are occurring.

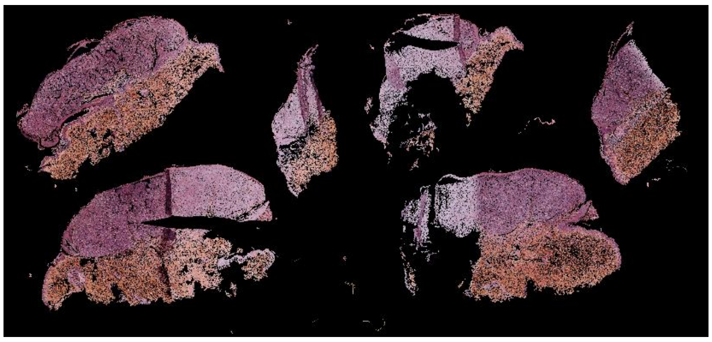
```
def filter_entropy(np_img, neighborhood=9, threshold=5, output_type="uint8"):
  """
  Filter image based on entropy (complexity).

  Args:
    np_img: Image as a NumPy array.
    neighborhood: Neighborhood size (defines height and width of 2D array of 1's).
    threshold: Threshold value.
    output_type: Type of array to return (bool, float, or uint8).
  Returns:
    NumPy array (bool, float, or uint8) where True, 1.0, and 255 represent a measure of
complexity.
  """
  t = Time()
  entr = sk_filters.rank.entropy(np_img, np.ones((neighborhood, neighborhood))) > threshold
  if output_type == "bool":
    pass
  elif output_type == "float":
    entr = entr.astype(float)
  else:
    entr = entr.astype("uint8") * 255
  return entr
```

3.6. Combining Filters

A strong tool that we can use, is the possibility to combine all these filters together though simple boolean algebra and logic operators as AND, OR, XOR, and NOT. An interesting combination of filters that should give good results for segmentation is the following: removal of color pen marks and slide background. To do this we use a “No Grays Filter”, a “Green Channel Filter” and “Green Pen”, “Blue Pen” and “Red Pen” filters. In addition, we can apply “Remove of Small Objects” to remove the small islands from the mask. We display the resulting mask.



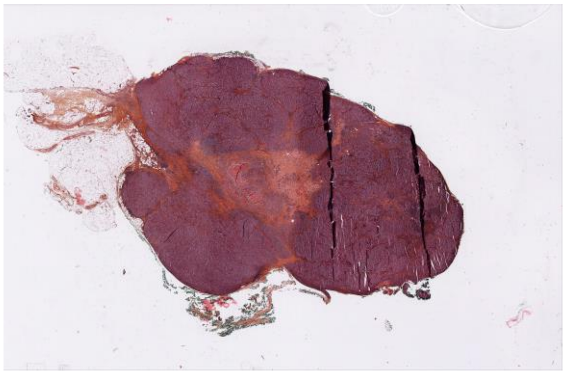




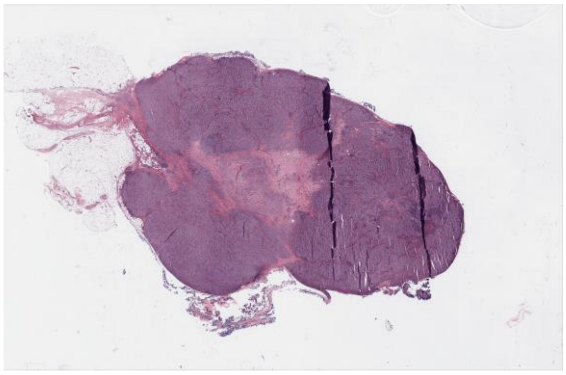
```
rgb = util.pil_to_np_rgb(img)
util.display_img(rgb, "Original")
mask_not_green = filter_green_channel(rgb)
mask_not_gray = filter_grays(rgb)
mask_no_red_pen = filter_red_pen(rgb)
mask_no_green_pen = filter_green_pen(rgb)
mask_no_blue_pen = filter_blue_pen(rgb)
mask_gray_green_pens = mask_not_gray & mask_not_green & mask_no_red_pen & mask_no_green_pen &
mask_no_blue_pen
mask_remove_small = filter_remove_small_objects(mask_gray_green_pens, min_size=500,
output_type="bool")
```

4. Normalization

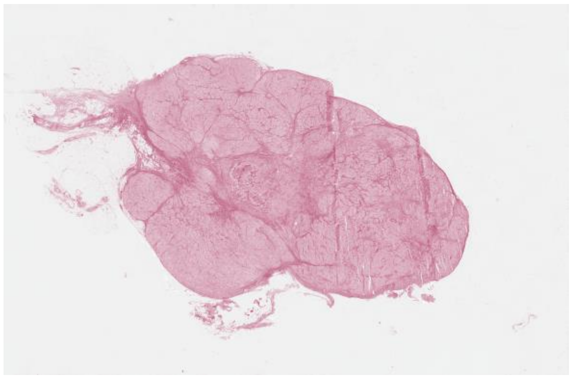
An important step that also must be done is the normalization of the images. We know that in the staining process the are several inconsistencies, what we have to do is bringing slides that were processed under different conditions into a common normalized space to enable improved quantitative analysis. What we did is basically to estimate the hematoxylin and eosin (H&E) stain vectors of the WSI of interest by using a singular value decomposition (SVD) approach applied to the non-background pixels of the input image. Second, the algorithm applies a correction to account for the intensity variations due to the original strength of the stain, staining procedure etc. Finally, the image is projected to a reference image such that after stain normalization all normalized images have similar color characteristics. The algorithm is based on the principle that the color of each pixel (RGB channels) is a linear combination of the two H&E stain vectors which are unknown and need to be estimated. Most of the processing steps are applied to the RGB color vectors converted to optical density (OD) domain. The Normalization process is based on (Marc Macenko). Here we can see the results:



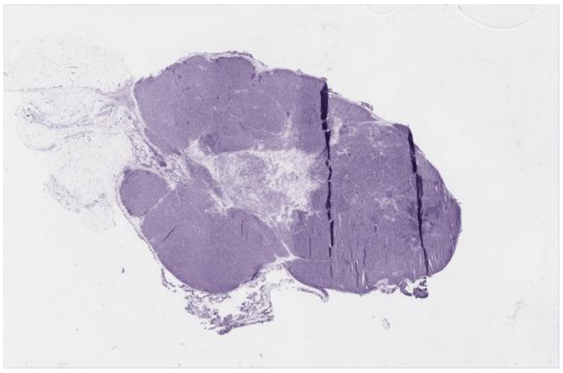
Original Image



Normalized Image



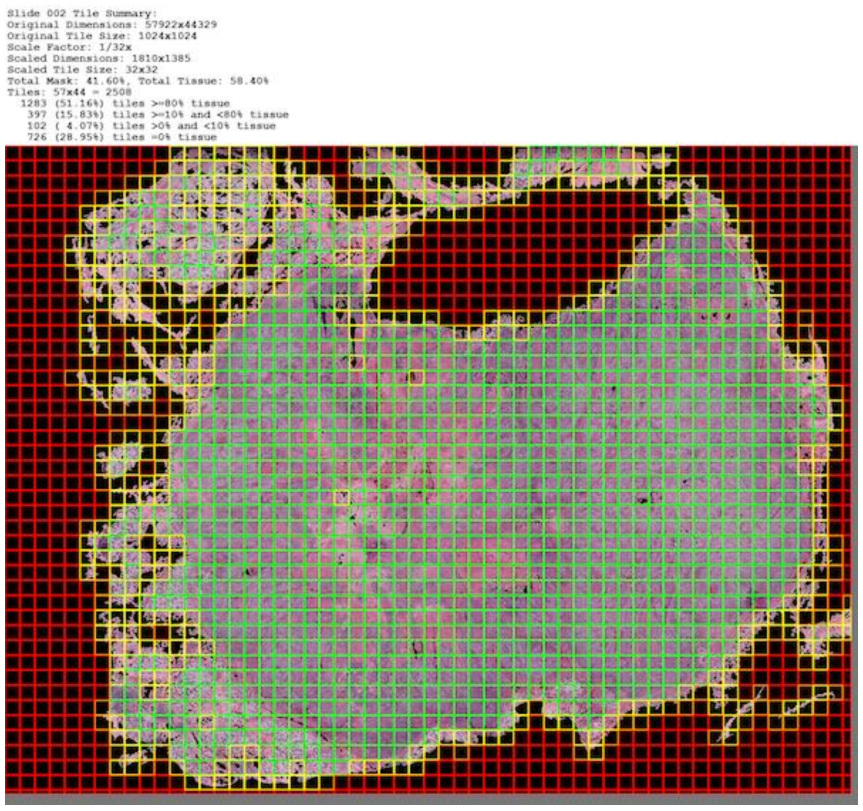
Eosin Component



Hematoxylin Component

5. Tiles

After the phases of filtering and normalization, it comes the one of patch selection and tissue sampling. We should have fairly good tissue segmentation for our dataset, where non-tissue pixels have been masked out from WSIs. In this phase we break our images into tile regions. The tissue percentage of each tile is color-coded in a similar way to a heat map. Tile with 80% or more tissue are green, between 10% and 80% yellow, less than 10% and greater than 0% are orange, and with 0% tissue are red. This values of course can be modified as desired as well as the colors. The heat map border size can be adjusted as well, while tiles sizes are specified according to the number of pixels in the original WSI files.



```
Slide 002 Tile Summary:
Original Dimensions: 57922x44329
Original Tile Size: 1024x1024
Scale Factor: 1/32x
Scaled Dimensions: 1810x1385
Scaled Tile Size: 32x32
Total Mask: 41.60%, Total Tissue: 58.40%
Tiles: 57x44 = 2508
  1283 (51.16%) tiles >=80% tissue
   397 (15.83%) tiles >=10% and <80% tissue
   102 ( 4.07%) tiles >0% and <10% tissue
   726 (28.95%) tiles =0% tissue
```

Here an example of an analyzed WSI:

Characteristic	Result
Original Dimensions	8109x5310
Original Tile Size	1,024x1,024
Scale Factor	1/32x
Scaled Dimensions	1,810x1,385
Scaled Tile Size	32x32
Total Mask	41.60%
Total Tissue	58.40%
Tiles	57x44 = 2,508
	1,283 (51.16%) tiles >=80% tissue
	397 (15.83%) tiles >=10% and <80% tissue
	102 (4.07%) tiles >0% and <10% tissue
	726 (28.95%) tiles =0% tissue

6. Metrics

The last part of the work concerns the check of the processed images with different metrics.

$$MSE = \frac{1}{m\,n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2$$

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right)\frac{?\sigma_{xy} + C_2)}{\cdot{}^2_x + \sigma_y^2 + C_2)}.$$

With the PSNR we are able to measure the differences with the compressed images. The higher the PSNR, the better quality of the compressed image. The signal in this case is the original data, and the noise is the error introduced by compression. Typical values are between 30dB and 50dB with 8 bit depth.

$$\mathrm{MDSI} = \left[\frac{1}{N}\sum_{i=1}^{N}\left|\widehat{\mathrm{GCS}}_i^{1/4} - \left(\frac{1}{N}\sum_{i=1}^{N}\widehat{\mathrm{GCS}}_i^{1/4}\right)\right|\right]^{1/4}$$

Mean Deviation Standard Index (MDSI) uses gradient magnitude to measure structural distortions and use chrominance features to measure color distortions. It shows very good compromise between prediction accuracy and model complexity, and it is efficient, effective and reliable at the same time. It also shows consistent performance for both natural and synthetic images.

$$\mathrm{FSIM}_C = \frac{\sum_{\mathbf{x}\in\Omega} S_L(\mathbf{x}) \cdot [S_C(\mathbf{x})]^\lambda \cdot PC_m(\mathbf{x})}{\sum_{\mathbf{x}\in\Omega} PC_m(\mathbf{x})}$$

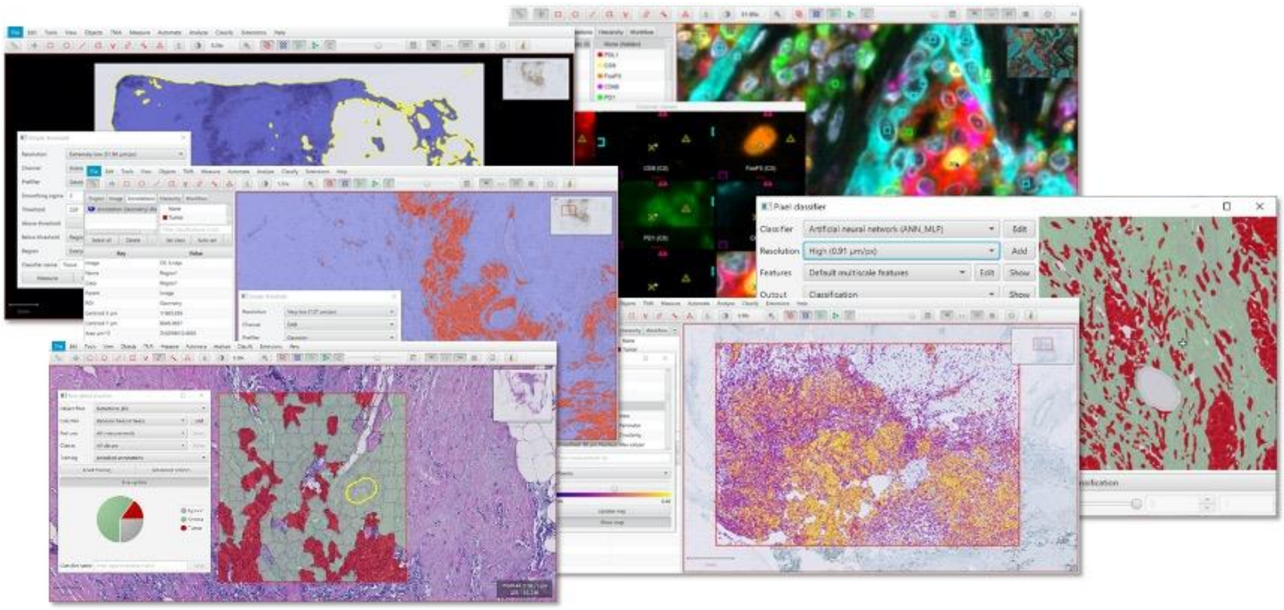
Feature Similarity Index is characterized primarily by phase congruency, which is a dimensionless measure of the significance of a local structure. As second feature, the metric is characterized by gradient magnitude, given the fact that phase congruency is contrast invariant while the contrast information does affect HVS perception of image quality. After obtaining the local quality map, we use again phase contrast as a weighting function to derive a single quality score.

For all the metrics we used PyTorch Image Quality (PIQ). The library contains a set of measures and metrics that is continually getting extended.

7. Appendix – QuPath

The initial phase of the project was dedicated to the learning and the knowledge of *QuPath*, an open-source software for bioimage analysis, often used for digital pathology applications because it offers a powerful set of tools for working with WSIs. *QuPath* is created and maintained by researchers whose focus is on developing the image processing algorithms and other computational methods that make the analysis possible -so that those who use the software can focus on analyzing their data and interpreting the results. We studied the main functionalities, so mainly:

- Open a WSI
- View image properties
- Navigate around images
- Separating stains
- Measuring areas
- Manually annotate regions
- Cell detection
- Cell classification
- Density maps
- View Measurements
- Export results
- Save & reload data



Pixel classifier

Classifier: Artificial neural network (ANN_MLP)

Resolution: High (0.91 µm/px)

Features: Default multiscale features

Output: Classification

References

A METHOD FOR NORMALIZING HISTOLOGY SLIDES FOR QUANTITATIVE ANALYSIS [Journal] / auth. Marc Macenko Marc Niethammer, J. S. Marron, David Borland, John T. Woosley, Xiaojun Guan, Charles Schmitt, and Nancy E. Thomas.

Developing image analysis pipelines of whole-slide images: Pre- and post-processing [Journal] / auth. Byron Smith Meyke Hermsen, Elizabeth Lesser, Deepak Ravichandar, Walter Kremers // Journal of Clinical and Translational Science. - [s.l.] : Cambridge University Press, August 27, 2020.

Metadata matters: access to image data in the real world [Journal] / auth. Melissa Linkert Curtis T. Rueden, Chris Allan, Jean-Marie Burel, Will Moore, Andrew Patterson // Journa Cell of Biology. - May 31, 2010.

Roche Digital Pathology Open Environment [Online] / auth. Gasuad Kimberly. - https://diagnostics.roche.com/global/en/article-listing/roche-digital-pathology-open-environment.html.

The impact of pre- and post-image processing techniques on deep learning frameworks: A comprehensive review for digital pathology image analysis [Journal] / auth. Massimo Salvi U. Rajendra Acharya, Filippo Molinari, Kristen M.Meiburger // Computers in Biology and Medicine. - January 2021. - Vol. 128.

Whole Slide Imaging (WSI) in Pathology: Current Perspectives and Future Directions [Journal] / auth. Neeta Kumar Ruchika Gupta, Sanjay Gupta // Journal of Digital Imaging. - [s.l.] : Springer Link, 2020. - Vol. 33.

Whole Slide Imaging and Its Applications to Histopathological Studies of Liver Disorders [Journal] / auth. Rossana C. N. Melo Maximilian W. D. Raas, Cinthia Palazzi, Vitor H. Neves, Kássia K. Malta and Thiago P. Silva. - January 8, 2020.

Whole slide imaging in pathology: advantages, limitations, and emerging perspectives [Journal] / auth. Farahani N Parwani A, Pantanowitz L. - 2015. - Vol. 2015:7 . - pp. 23—33.

Whole-slide image preprocessing in Python [Report] / auth. Deron Eriksson Fei Hu. - [s.l.] : IBM Developer, 2018.