CHALLENGE 5

Xabier Oyanguren Asua 1456628

In [1]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

Load Problem Images

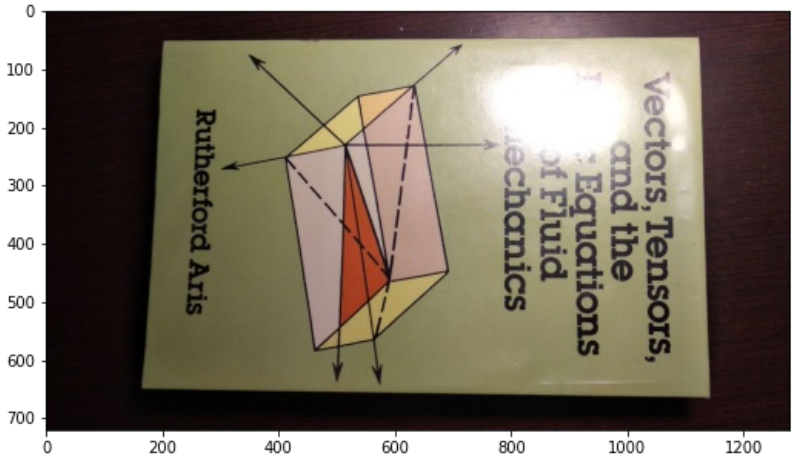
Let us first load four images from different angles, such that the light reflex appears in different points of the image. Our objective will be to have a photo in which we can see the book cover with no bright spot over it.

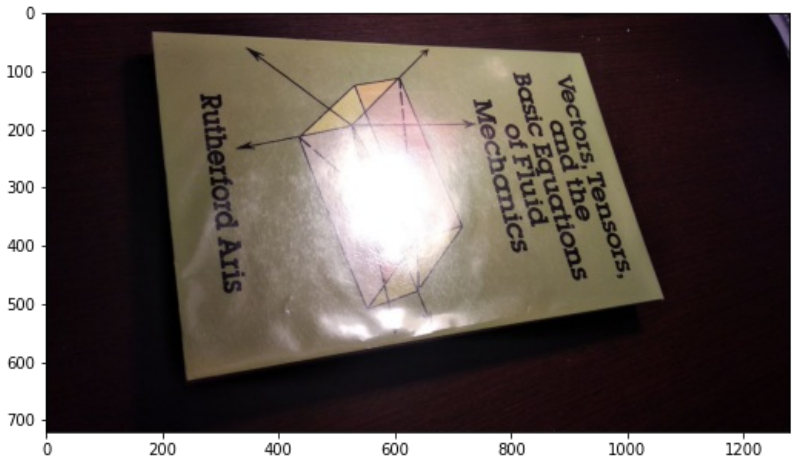
In [2]:

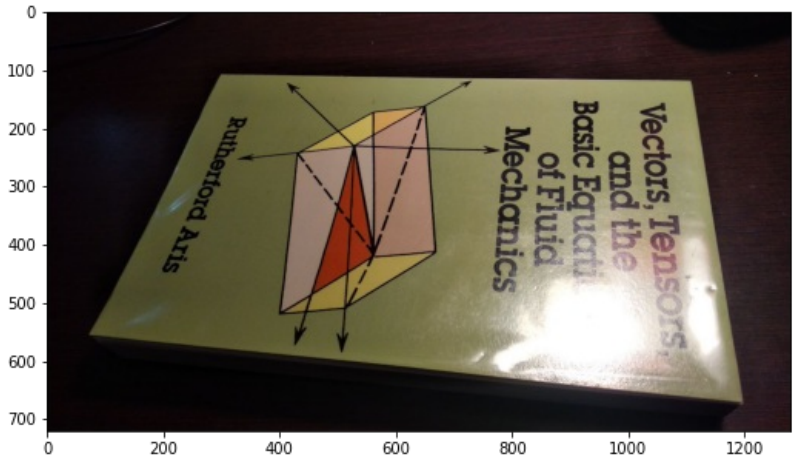
```
im1 = cv2.imread("bookA.jpeg")[:,:,::-1] # get them as RGB not BGR
im2 = cv2.imread("bookB.jpeg")[:,:,::-1]
im3 = cv2.imread("bookC.jpeg")[:,:,::-1]
im4 = cv2.imread("bookD.jpeg")[:,:,::-1]
```

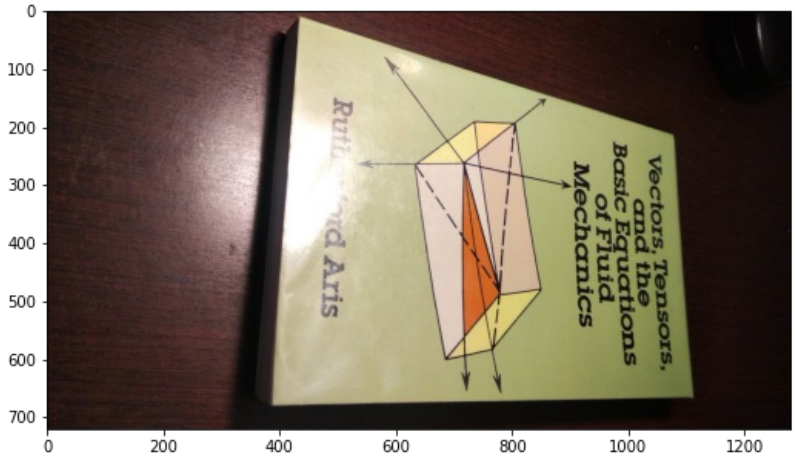
In [3]:

```
def plot(im, cmap='gray', title=None):
    %matplotlib inline
    plt.figure(figsize=(10,5))
    plt.imshow(im, cmap=cmap)
    if title is not None:
        plt.title(title)
    plt.show()
[plot(im) for im in [im1,im2,im3,im4]]
```









Out[3]:

```
[None, None, None, None]
```

General Routines we will be using

Let us now define the functions to get the homography sending pixel coordinates in the image plane of the first image (which will be our reference) to the pixel positions of the rest of image camera planes. With it, we will be able to any of the other three images as seen from the camera position and angle of the reference image.

To get this homography we will employ the Direct Linear Transform method. For that we need to manually choose some corresponding points between the first and the rest of images, so we also define functions for that.

In [4]:

```
def choose_points(image, num_points=-1):
    %matplotlib tk
    plt.imshow(image)
    plt.title("Choose the points of correspondence")
    points = plt.ginput(n=num_points, timeout=-1, show_clicks=True)
    plt.show()
    return points

def build_homography_from_m_to_mp(m_points, mp_points):
    # mp alpha Hm
    # turned into Bh=0
    B = np.zeros((len(m_points)*2, 9), dtype=np.float64)
    for k, (mp, m) in enumerate(zip(mp_points, m_points)):
        B[k*2:(k+1)*2, :] = np.array([[0,0,0,-m[0], -m[1], -1, mp[1]*m[0], mp[1]*m[1], mp[1]],
                                      [m[0], m[1], 1, 0,0,0, -mp[0]*m[0], -mp[0]*m[1], -mp[0]]])
    u,s,vt = np.linalg.svd(B)
    h = vt.T[:,-1]
    H = h.reshape(3,3)
    H = H/H[2,2]
    # st now if we take [m']=Hm we know which pixel in image of m should be correspondent with m'
    return H

def homog_ops(H, inputs): # inputs expected to be [3,N] or [ 2,N]
    if inputs.shape[0]==2:
        inputs = np.vstack((inputs, np.ones(inputs.shape[-1])))
    out = H@inputs
    return (out/out[-1, :])[:-1,:]

def image_of_mp_as_seen_from_m( H_m_to_mp, image_m, image_mp ):
    # Image of m' as seen from image m-s camera
    image_mp_as_m = np.zeros(image_m.shape, image_m.dtype)
    dest_xys = np.array(np.meshgrid(
                    np.arange(image_m.shape[1]), np.arange(image_m.shape[0]))
                       ).reshape(-1, image_m.shape[1]*image_m.shape[0])
    # [2 (x,y), image_m.shape[0]*image_m.shape[1]] both dest_xys and look_up_mps
    look_up_mps = np.round( homog_ops( H_m_to_mp, dest_xys ) ).astype(int)
    is_out_of_bounds = (look_up_mps[0]>=0) & (look_up_mps[0]<image_mp.shape[1]) & \
                       (look_up_mps[1]>=0) & (look_up_mps[1]<image_mp.shape[0])
    # [image_m.shape[0]*image_m.shape[1]]
    for dest_m, look_up_mp, is_out in zip(dest_xys.T, look_up_mps.T, is_out_of_bounds):
        if is_out:
            image_mp_as_m[dest_m[1],dest_m[0], :] = image_mp[look_up_mp[1],look_up_mp[0], :]

    return image_mp_as_m


def get_image_of_mp_as_seen_from_m_and_viceversa(image_m, image_mp, plot_them=True):
    points_m = choose_points(image_m)
    points_mp = choose_points(image_mp, len(points_m))

    print(f"Gathered {len(points_m)} points")
    H_m_to_mp = build_homography_from_m_to_mp(points_m, points_mp)
    image_mp_as_m = image_of_mp_as_seen_from_m( H_m_to_mp, image_m, image_mp )

    H_mp_to_m = build_homography_from_m_to_mp(points_mp, points_m)
    image_m_as_mp = image_of_mp_as_seen_from_m( H_mp_to_m, image_mp, image_m )
    if plot_them==True:
        plot(np.hstack((image_m, image_mp_as_m)))
        plot(np.hstack((image_mp, image_m_as_mp)))
    return image_mp_as_m, image_m_as_mp, H_mp_to_m, H_m_to_mp
```

Get the Corresponding points manually

Get the corresponding points of the four images:

In [ ]:

```
points = []
points.append(choose_points(im1))
points.append(choose_points(im2, len(points[0])))
points.append(choose_points(im3, len(points[0])))
points.append(choose_points(im4, len(points[0])))
```

In [15]:

```
print(f"Gathered {len(points[0])} points")
```

```
Gathered 18 points
```

We chose in each 18 points.

Get the Homographies from the first image to the rest

Now, we will get the homographies from the first image to the rest of images:

In [6]:

```
H_im1_to_imk = [build_homography_from_m_to_mp(points[0], points[k]) for k in range(1,4)] # m is 1 mp are the rest
imagek_as_im1 = [image_of_mp_as_seen_from_m(H_im1_to_imk[k], im1, imk) for k, imk in enumerate([im2,im3,im4])]
```

In [7]:

```
imagek_as_im1 = [im1]+imagek_as_im1
for k in range(4):
    print(f"Image im{k+1} as seen from im1's camera position and angle")
    plot(imagek_as_im1[k])
```

Image im1 as seen from im1's camera position and angle

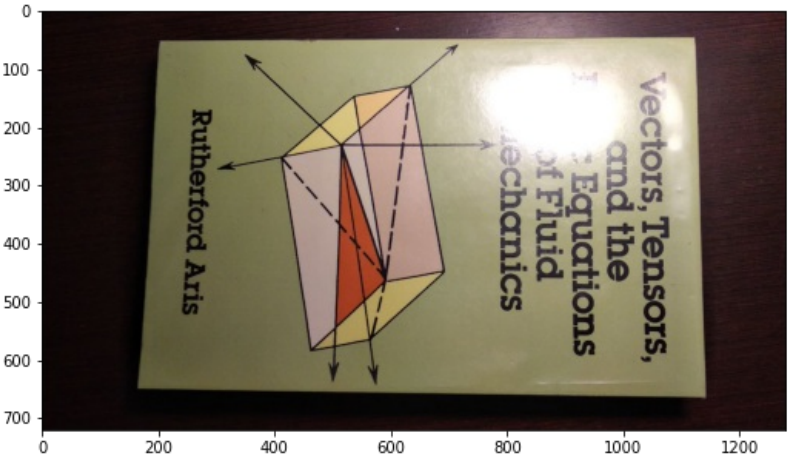


Image im2 as seen from im1's camera position and angle

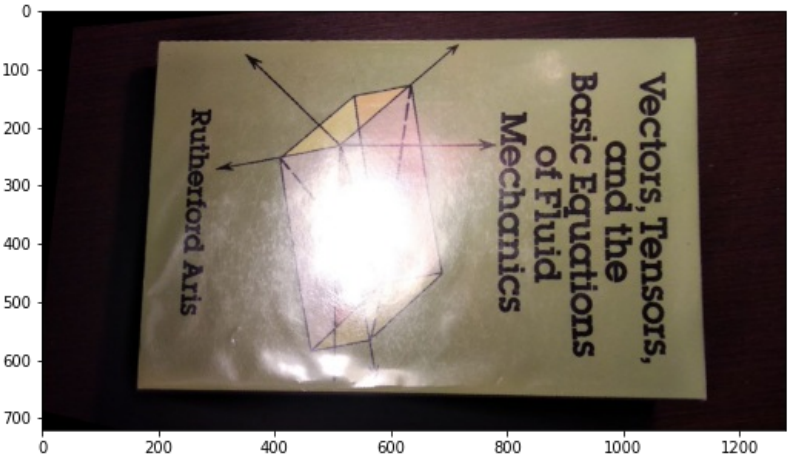


Image im3 as seen from im1's camera position and angle

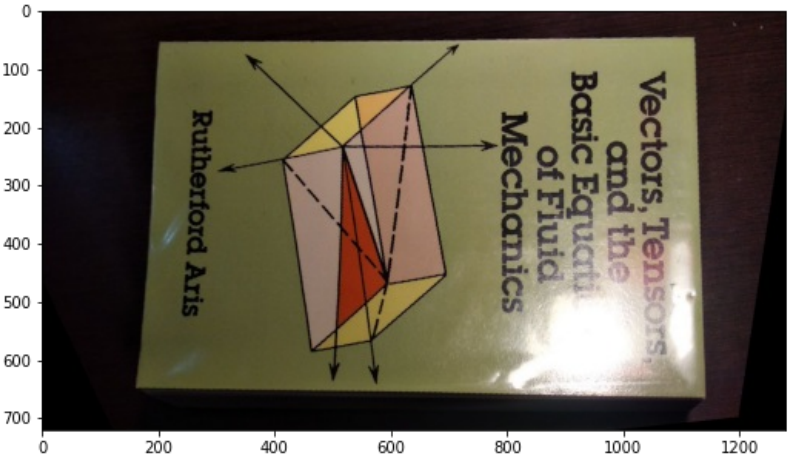
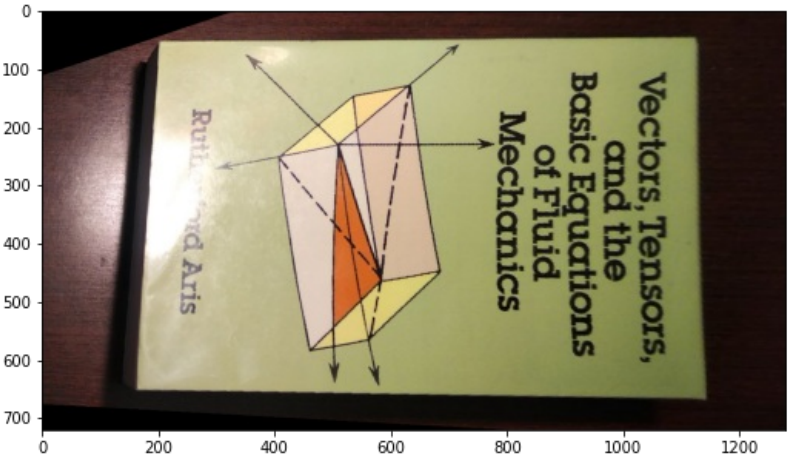


Image im4 as seen from im1's camera position and angle



Register the Different Images using Fourier Phase Correlation

Since I do not know how to do the reference frame warp indicated in the challenge guide, I will use instead image register using Fourier phase correlation.

In [8]:

```
from scipy.fft import fft2, fftshift, ifft2

def register_images(ims_to_reg): # first image will be taken as reference
    coefs = [fft2( cv2.cvtColor(im, cv2.COLOR_RGB2GRAY) ) for im in ims_to_reg]

    corr1ks = [coefs[k]*coefs[0].conj() for k in range(1,len(ims_to_reg))]
    corr1ks = [corr1k/np.abs(corr1k) for corr1k in corr1ks] # Correlation phases in Fourier Space
    corr1ks = [np.abs(fftshift(ifft2( corr1k ))) for corr1k in corr1ks] # correlation delta in space
    # Obtain the peak correlation match indices
    match_inds = np.vstack((np.array(ims_to_reg[0].shape[:-1])//2,
                np.array([np.unravel_index(np.argmax(corr1k, axis=None), corr1k.shape) for corr1k in corr1ks]))).
T
    # Select the matching window with overlaping parts within all registered channels
    h1 = np.min(match_inds[0])
    w1 = np.min(match_inds[1])
    h2 = min([ims_to_reg[k].shape[0]-match_ind for k, match_ind in enumerate(match_inds[0])])
    w2 = min([ims_to_reg[k].shape[1]-match_ind for k, match_ind in enumerate(match_inds[1])])
    # crop and align the images
    return np.stack(list((ims_to_reg[k][match_inds[0][k]-h1:match_inds[0][k]+h2,
                                    match_inds[1][k]-w1:match_inds[1][k]+w2] for k in range(len(ims_to_reg)))),
                    axis=0)
```

In [9]:

```
imagek_as_im1_registered = register_images(imagek_as_im1)
```

Merge Images to get the book with no reflection

If we now compute the average, the median or the minimum over each channel we could get the bright spots out. The mean and median should work since only one of the images has a bright spot in each region (more or less). The minimum on the other hand should work, for the bright spot is an intensity saturated point with maximum brigthness in the three channels, unlike the non-bright spot book cover (which can be seen to be yellow).

We will try the straight element-wise mean, median and minimum using each color independently, but also doing this using the grayscale images, which integrate the three channels of eahc image within a single value (color channels from different images will not be mixed this way). We will then judge which one gives us the best result.

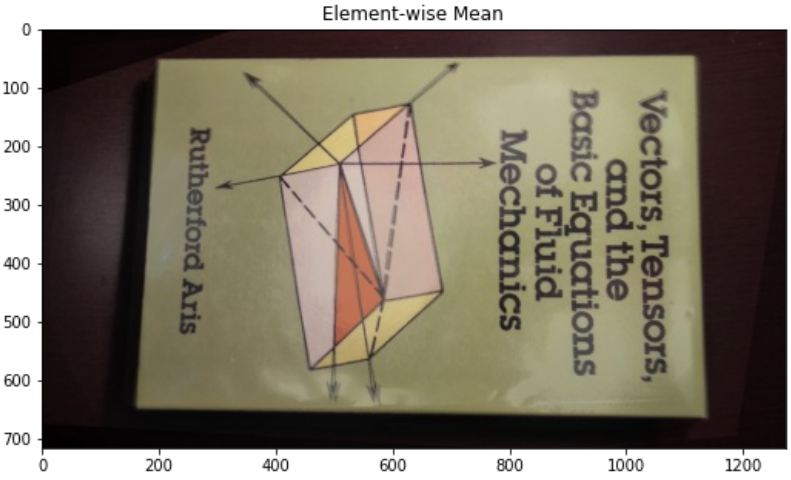
In [10]:

```
result_mean = np.mean((np.array(imagek_as_im1_registered)).astype(np.float64), axis=0).astype(np.uint8)
result_median = np.median((np.array(imagek_as_im1_registered)).astype(np.float64), axis=0).astype(np.uint8)
result_channel_wise_min = np.min((np.array(imagek_as_im1_registered)).astype(np.float64), axis=0).astype(np.uint8
)

# The grayscale element-wise methods:
result_grayscale_min=np.zeros(imagek_as_im1_registered[0].shape, imagek_as_im1_registered[0].dtype)
result_grayscale_median=np.zeros(imagek_as_im1_registered[0].shape, imagek_as_im1_registered[0].dtype)

grayed = np.array([cv2.cvtColor(im, cv2.COLOR_RGB2GRAY) for im in imagek_as_im1_registered])

args_gray_min = np.argmin(grayed, axis=0)
args_gray_median = np.argsort(grayed, axis=0)[grayed.shape[0]//2-1]

for i in range(imagek_as_im1_registered.shape[1]):
    for j in range(imagek_as_im1_registered.shape[2]):
        result_grayscale_min[i,j,:] = imagek_as_im1_registered[args_gray_min[i,j],i,j,:]
        result_grayscale_median[i,j,:] = imagek_as_im1_registered[args_gray_median[i,j],i,j,:]
```

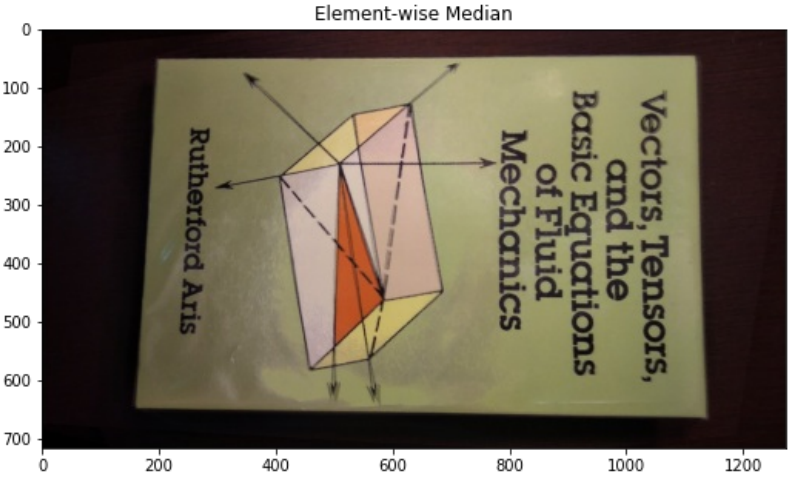
In [11]:

```
plot(result_mean, title="Element-wise Mean")
plot(result_median, title="Element-wise Median")
plot(result_grayscale_median, title="Grayscale element-wise Median")
plot(result_channel_wise_min, title="Element-wise Minimum")
plot(result_grayscale_min, title="Grayscale element-wise Minimum")
```

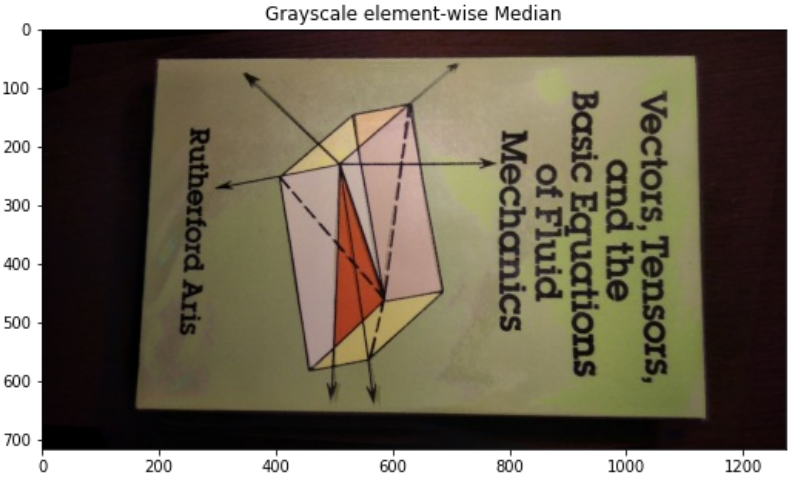
Element-wise Mean



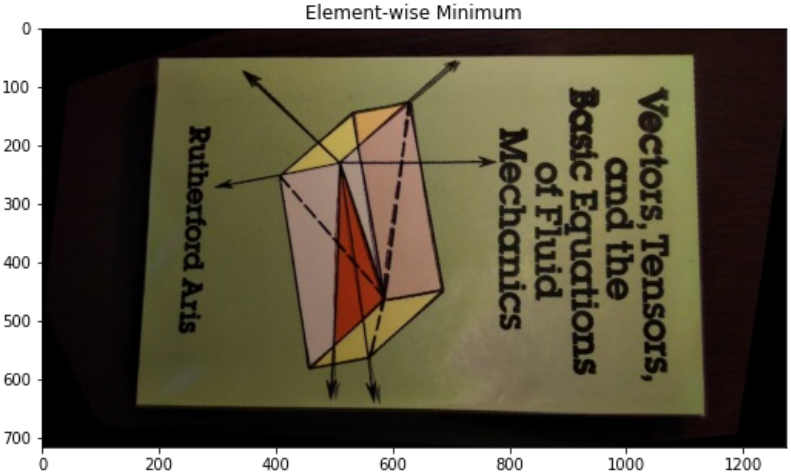
Element-wise Median

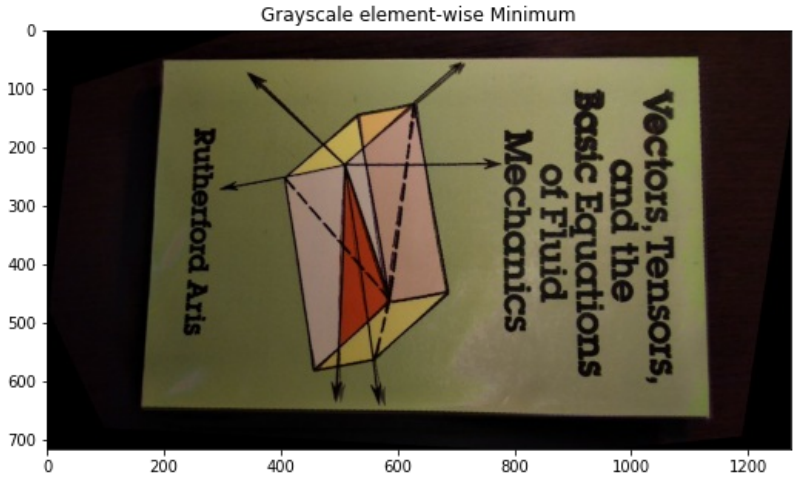


Grayscale element-wise Median



Element-wise Minimum





We will choose the "grayscale element-wise median" as the optimal one by the virtue of the highest visual definition of the image. Interestingly, since some images have different colour tones (in particular the fourth image, due to the ambient light possibly), the book is recovered with differently stained regions! This actually allows us to visually know which pixels come from the fourth image and which ones no. It is interesting to note that even if it was the image with the most cumbersome angle, where the top part of the book (right in the images) is seen as the smallest one, the fourth image still contributes considerably in the final reconstruction of that part of the image with no reflections.

In [14]:

```
plot(result_grayscale_median, title="Grayscale element-wise Median")
```