ДОДАТОК А

Реалізація алгоритму автоматичного підрахунку фракції викиду лівого шлуночка серця

А.1 – Архітектура класичної U-Net

```
class inconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(inconv, self).__init__()
        self.conv = double_conv(in_ch, out_ch)

    def forward(self, x):
        x = self.conv(x)
        return x

class double_conv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(double_conv, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        x = self.conv(x)
        return x

class down(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(down, self).__init__()
        self.mpconv = nn.Sequential(
            nn.MaxPool2d(2),
            double_conv(in_ch, out_ch)
        )
    def forward(self, x):
        x = self.mpconv(x)
        return x
class up(nn.Module):
    def __init__(self, in_ch, out_ch, bilinear=True):
        super(up, self).__init__()
        if bilinear:
            self.up = nn.Upsample(scale_factor=2,
mode='bilinear', align_corners=True)
        else:
```

```
            self.up = nn.ConvTranspose2d(in_ch//2, in_ch//2,
2, stride=2)
        self.conv = double_conv(in_ch, out_ch)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, (diffX // 2, diffX - diffX//2,
                        diffY // 2, diffY - diffY//2))
        x = torch.cat([x2, x1], dim=1)
        x = self.conv(x)
        return x
class outconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(outconv, self).__init__()
        self.conv = nn.Conv2d(in_ch, out_ch, 1)
    def forward(self, x):
        x = self.conv(x)
        return x
class UNet(nn.Module):
    def __init__(self, n_channels=1,bilinear=True):
        super(UNet, self).__init__()
        self.inc = inconv(n_channels, 16)
        self.down1 = down(16, 32)
        self.down2 = down(32, 64)
        self.down3 = down(64, 128)
        self.down4 = down(128, 128)
        self.up1 = up(256, 64,bilinear)
        self.up2 = up(128, 32,bilinear)
        self.up3 = up(64, 16,bilinear)
        self.up4 = up(32, 16,bilinear)
        self.outc = outconv(16, 1)
    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        out = self.up1(x5, x4)
        out = self.up2(out, x3)
        out = self.up3(out, x2)
        out = self.up4(out, x1)
        out = self.outc(out)
        return out
```

А.2 – Архітектура MFP-UNet

```
class inconv(nn.Module):
```

```
    def __init__(self, in_ch, out_ch):
        super(inconv, self).__init__()
        self.conv = double_conv(in_ch, out_ch)

    def forward(self, x):
        x = self.conv(x)
        return x

class double_conv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(double_conv, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        x = self.conv(x)
        return x

class down(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(down, self).__init__()
        self.mpconv = nn.Sequential(
            nn.MaxPool2d(2),
            double_conv(in_ch, out_ch)
        )

    def forward(self, x):
        x = self.mpconv(x)
        return x


class up(nn.Module):
    def __init__(self, in_ch, out_ch, bilinear=True):
        super(up, self).__init__()
        self.up = nn.ConvTranspose2d(in_ch, in_ch//2, 2)
        self.conv = double_conv(in_ch, out_ch)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, (diffX // 2, diffX - diffX//2,
                        diffY // 2, diffY - diffY//2))

        x = torch.cat([x2, x1], dim=1)
        x = self.conv(x)
```

```
        return x
class outconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(outconv, self).__init__()
        self.conv = nn.Conv2d(in_ch, out_ch, 1)

    def forward(self, x):
        x = self.conv(x)
        return x

class pyramid(nn.Module):
    def __init__(self, in_ch, out_ch=16, scale_factor=2):
        super(pyramid, self).__init__()
        self.scale_factor = scale_factor
        self.conv = nn.Conv2d(in_ch, out_ch, 3, padding=1)
        self.up = nn.Upsample(scale_factor=scale_factor,
mode='bilinear', align_corners=True)

    def forward(self, x):
        x = self.conv(x)
        if(self.scale_factor != 0):
            x = self.up(x)
        return x

class MFP_UNet(nn.Module):
    def __init__(self, n_channels=1, bilinear=True):
        super(MFP_UNet, self).__init__()

        self.inc = inconv(n_channels, 16)
        self.down1 = down(16, 32)
        self.down2 = down(32, 64)
        self.down3 = down(64, 128)
        self.down4 = down(128, 128)

        self.up1 = up(128, 64, bilinear)
        self.up2 = up(64, 32, bilinear)
        self.up3 = up(32, 16, bilinear)

        self.pyramid1 = pyramid(128, scale_factor=8)
        self.pyramid2 = pyramid(64, scale_factor=4)
        self.pyramid3 = pyramid(32, scale_factor=2)
        self.pyramid4 = pyramid(16, scale_factor=0)

        self.outconv = outconv(64, 1)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
```

```
        out1 = self.up1(x4, x3)
        out1 = self.dropout(out1)

        out2 = self.up2(out1, x2)
        out2 = self.dropout(out2)

        out3 = self.up3(out2, x1)

        layer1 = self.pyramid1(x4)
        layer2 = self.pyramid2(out1)
        layer3 = self.pyramid3(out2)
        layer4 = self.pyramid4(out3)

        out = torch.cat([layer1, layer2, layer3, layer4],
dim=1)
        out = self.outconv(out)

        return out
```

А.3 – Аугментація даних

```
class RandomShift(object):
    def __init__(self, probability=0.5):
        self.prob = probability
    def __call__(self, sample):
        image,mask = sample['image'],sample['mask']
        rnd = np.random.uniform()
        if rnd <= self.prob:
#             print("Shift")
            cols = image.shape[1]
            rows = image.shape[0]
            rand = np.random.randint(0, 21)
            type_of_shift = np.random.choice([1, 2, 3, 4], 1)
            shift_x = round(cols * rand / 100)
            shift_y = round(rows * rand / 100)
            new_width = cols - round(cols * rand / 100)
            new_height = rows - round(rows * rand / 100)
            if(type_of_shift == 1):
                image[:, 0:new_width] = image[:, shift_x:]
                image[:, new_width:] = 0.0
                mask[:, 0:new_width] = mask[:, shift_x:]
                mask[:, new_width:] = 0.0
            elif(type_of_shift == 2):
                image[:, shift_x:] = image[:, :new_width]
                image[:, 0:shift_x] = 0
                mask[:, shift_x:] = mask[:, :new_width]
                mask[:, 0:shift_x] = 0
            elif(type_of_shift == 3):
                image[0:new_height, :] = image[shift_y:, :]
                image[new_height:, :] = 0
```

```
                mask[0:new_height, :] = mask[shift_y:, :]
                mask[new_height:, :] = 0
            elif(type_of_shift == 4):
                image[shift_y:, :] = image[:new_height, :]
                image[0:shift_y, :] = 0
                mask[shift_y:, :] = mask[:new_height, :]
                mask[0:shift_y, :] = 0
        return {'image': image, 'mask': mask }
class RandomScale():
    def __init__(self, probability=0.5):
        self.prob = probability
    def __call__(self, sample):
        image,mask = sample['image'],sample['mask']
        rnd = np.random.uniform()
        if rnd <= self.prob:
            cols = image.shape[1]
            rows = image.shape[0]
            rand = np.random.randint(2, 16)
            new_width = cols - round(cols * rand / 100 / 2) *
2
            new_height = rows - round(rows * rand / 100 / 2) *
2
            shift_x = round(cols * rand / 100 / 2)
            shift_y = round(rows * rand / 100 / 2)
            type_of_scale = np.random.choice([1, 2], 1)
            if(type_of_scale == 1):
                image[shift_y:-shift_y, shift_x:-shift_x] =
cv2.resize(image, (new_width, new_height))
                image[0:shift_y, :] = 0.0
                image[new_height+shift_y:, :] = 0.0
                image[:,0:shift_x] = 0.0
                image[:, new_width+shift_x:] = 0.0
                mask[shift_y:-shift_y, shift_x:-shift_x] =
cv2.resize(mask, (new_width, new_height))
                mask[0:shift_y, :] = 0.0
                mask[new_height+shift_y:, :] = 0.0
                mask[:, 0:shift_x] = 0.0
                mask[:, new_width+shift_x:] = 0.0
            elif(type_of_scale == 2):
                image = image[shift_y:-shift_y, shift_x:-
shift_x]
                mask = mask[shift_y:-shift_y, shift_x:-
shift_x]
        return {'image': image, 'mask': mask }
class RandomRotate(object):
    def __init__(self, probability=0.5):
        self.prob = probability
    def __call__(self, sample):
        image,mask = sample['image'],sample['mask']
        rnd = np.random.uniform()
        if rnd <= self.prob:
#             print("Rotate")
            degree = np.random.randint(1, 15)
```

```
            rotate_direction = np.random.choice([1, 2], 1)
            degree = degree if rotate_direction == 1 else
degree*(-1)
            rows, cols = image.shape
            M = cv2.getRotationMatrix2D((cols/2,rows/2),
degree, 1)
            image = cv2.warpAffine(image, M, (cols, rows))
            mask = cv2.warpAffine(mask, M, (cols, rows))
        return {'image': image, 'mask': mask }
class Resize():
    def __init__(self):
        super(Resize, self).__init__()
    def __call__(self, sample):
        image,mask = sample['image'],sample['mask']
        image = cv2.resize(image, (IMG_WIDTH, IMG_HEIGHT))
        mask = cv2.resize(mask, (IMG_WIDTH, IMG_HEIGHT))
#         mask = np.transpose(mask, (2, 0, 1))
        return {'image': image, 'mask': mask }
class RandomLight(object):
    def __init__(self, probability=0.5, lower_level=0.8,
upper_level=1.8):
        self.prob = probability
        self.lower_level = lower_level
        self.upper_level = upper_level
    def __call__(self, sample):
        image,mask = sample['image'],sample['mask']
        rnd = np.random.uniform()
        if rnd <= self.prob:
            rnd_light = np.random.choice([1, 2], 1)
            if(rnd_light == 1):
                gamma = ((9 - 7) * np.random.random_sample() +
7) / 10 # get value in range [0.5, 0.9] - make image darker
            else:
                gamma = (2 - 1) * np.random.random_sample() +
1
            invGamma = 1.0 / gamma
            table = np.array([((i / 255.0) ** invGamma) * 255
                              for i in np.arange(0,
256)]).astype("uint8")
            image=cv2.LUT(image, table)
        return {'image': image, 'mask': mask }
```

А.4 – Додавання додаткових шарів координат

```
class AddCoordsLayers():
    def __init__(self):
        super(AddCoordsLayers, self).__init__()
    def __call__(self, sample):
        image,mask = sample['image'],sample['mask']
        x_dim = image.shape[0]
```

```
        y_dim = image.shape[1]
        xx_ones = np.ones([1, x_dim], dtype=np.int32)
        xx_ones = np.expand_dims(xx_ones, -1)
        xx_range = np.tile(np.expand_dims(range(y_dim), 0),
[1, 1])
        xx_range = np.expand_dims(xx_range, 1)
        xx_channel = np.matmul(xx_ones, xx_range)
        yy_ones = np.ones([1, y_dim], dtype=np.int32)
        yy_ones = np.expand_dims(yy_ones, 1)
        yy_range = np.tile(np.expand_dims(range(x_dim), 0),
[1, 1])
        yy_range = np.expand_dims(yy_range, -1)
        yy_channel = np.matmul(yy_range, yy_ones)
        x_dim *= 1.0
        y_dim *= 1.0
        xx_channel = xx_channel.astype('float32') / (x_dim -
1)
        yy_channel = yy_channel.astype('float32') / (y_dim -
1)
        xx_channel = xx_channel*2 - 1
        yy_channel = yy_channel*2 - 1
        return {'image': image, 'mask': mask, 'x_layer':
xx_channel, 'y_layer': yy_channel}
class CustomToTensor(transforms.ToTensor):
    """
    Extension of original ToTensor but applying to an image
property only
    """
    def __init__(self):
        super(CustomToTensor, self).__init__()
    def __call__(self, sample):
        image,mask,x_layer, y_layer =
sample['image'],sample['mask'],sample["x_layer"],sample["y_lay
er"]
        image_tensor = super().__call__(image)
        x_layer = torch.from_numpy(x_layer)
        y_layer = torch.from_numpy(y_layer)
        mask_tensor = torch.from_numpy(mask)  #expect to be
uint8
        return {'image': image_tensor, 'mask': mask_tensor,
'x_layer': x_layer, 'y_layer': y_layer}
```

А.5 – Функція витрат та функції оцінок

```
class BCEDiceLoss(nn.Module):
    def __init__(self):
        super(BCEDiceLoss, self).__init__()
    def forward(self, input, target):
        bce = F.binary_cross_entropy_with_logits(input,
target)
```

```
        smooth = 1e-5
        input = torch.sigmoid(input)
        num = target.size(0)
        input = input.view(num, -1)
        target = target.view(num, -1)
        intersection = (input * target)
        dice = (2. * intersection.sum(1) + smooth) /
(input.sum(1) + target.sum(1) + smooth)
        dice = 1 - dice.sum() / num
        return 0.5 * bce + dice
def iou_score(output, target):
    smooth = 1e-5
    if torch.is_tensor(output):
        output = torch.sigmoid(output).data.cpu().numpy()
    if torch.is_tensor(target):
        target = target.data.cpu().numpy()
    output_ = output > 0.5
    target_ = target > 0.5
    intersection = (output_ & target_).sum()
    union = (output_ | target_).sum()
    return (intersection + smooth) / (union + smooth)
def dice_coef(output, target):
    smooth = 1e-5
    output = torch.sigmoid(output).view(-1).data.cpu().numpy()
    target = target.view(-1).data.cpu().numpy()
    intersection = (output * target).sum()
    return (2. * intersection + smooth) / \
        (output.sum() + target.sum() + smooth)
def accuracy(output, target):
    output = torch.sigmoid(output).view(-1).data.cpu().numpy()
    output = (np.round(output)).astype('int')
    target = target.view(-1).data.cpu().numpy()
    target = (np.round(target)).astype('int')
    (output == target).sum()
    return (output == target).sum() / len(output)
```

А.6 – Аналітичний метод підрахунку

```
def to_same_zoom(a2c_ed_mask, a2c_es_mask, a4c_ed_mask,
a4c_es_mask,
                 length_ed_a2c, length_es_a2c, length_ed_a4c,
length_es_a4c):    if length_ed_a4c > length_ed_a2c:
        ed_coeff = length_ed_a4c/length_ed_a2c
        es_coeff = length_es_a4c/length_es_a2c
        scale_coeff = (ed_coeff + es_coeff)
/  2        a2c_ed_mask = cv2.resize(a2c_ed_mask, None,
fx=scale_coeff, fy=scale_coeff)
        a2c_es_mask = cv2.resize(a2c_es_mask, None,
fx=scale_coeff, fy=scale_coeff)    else:
        ed_coeff = length_ed_a2c/length_ed_a4c
        es_coeff = length_es_a2c/length_es_a4c
```

```
        scale_coeff = (ed_coeff + es_coeff)
/  2        a4c_ed_mask = cv2.resize(a4c_ed_mask, None,
fx=scale_coeff, fy=scale_coeff)
        a4c_es_mask = cv2.resize(a4c_es_mask, None,
fx=scale_coeff, fy=scale_coeff)    return a2c_ed_mask,
a2c_es_mask, a4c_ed_mask, a4c_es_maskdef getPerpCoord(aX, aY,
bX, bY, length, height):
    vX = bX-aX
    vY = bY-aY
    L = math.sqrt(vX*vX + vY*vY)
    vX = vX / L
    vY = vY / L
    temp = vX
    vX = 0-vY
    vY = temp
    cX = aX + vX * length
    cY = aY + vY * length
    dX = aX - vX * length
    dY = aY - vY * length
    return int(math.ceil(cX)), int(math.ceil(cY+height)),
int(math.ceil(dX)), int(math.ceil(dY+height))def
get_point(countour_img, center, y, direction):
    img_slice = countour_img[y-1:y]
    points = np.argwhere(img_slice == 255)[:, 1]    if not
np.any(points):
        return None    if direction == 'left':
        return np.min(points), y
    else:
        return np.max(points), ydef
find_apex_mitral_points(img):
    # gray = cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)
    img = cv2.cvtColor(img[:,:,0],cv2.COLOR_GRAY2RGB)
    edged = cv2.Canny(img, 1, 10)
    cnts = cv2.findContours(edged.copy(), cv2.RETR_LIST ,
cv2.CHAIN_APPROX_SIMPLE)    cnts =
imutils.grab_contours(cnts)    c = cnts[0]
    contours = c.squeeze()    ind_apex_point =
np.argmin(contours[:, 1])
    # Find apex point
    apex_point = contours[ind_apex_point]
    copy_img = img.copy()
    circle_img = np.zeros( img.shape[0:2] )
    center, radius = cv2.minEnclosingCircle(c)
    countour_img = edged.copy()
    circle_img = cv2.circle(circle_img, (int(center[0]),
int(center[1])), int(radius), 1)
    intersection = np.logical_and( countour_img, circle_img )
    arr = np.nonzero(intersection)
    cv2.drawContours(copy_img, c, -1, (0,255,0), 1)
    cv2.circle(copy_img, (int(center[0]), int(center[1])),
int(radius), (0,255,0), 3)
    points = []
    rng = len(arr[0])
```

```
    for i in range(rng):
        points.append([arr[1][i], arr[0][i]])
    bottom_points = []
    for i in range(len(points)):
        if points[i][1] >= center[1]:
            bottom_points.append(points[i])    left_point =
min(bottom_points, key = lambda t: t[0])
    right_point = max(bottom_points, key = lambda t:
t[0])    directions = []
    if right_point[0] < center[0]:
        directions.append('right')
    if left_point[0] > center[0]:
        directions.append('left')
    for direction in directions:
        r = int(radius + center[1] - 5)
        if direction == 'left':
            old_point = left_point
        else:
            old_point = right_point
        while True:
            r -= 1
            point = get_point(edged, center, r, direction)
            if point is None:
                continue
            if direction == 'left' and point[0] >=
old_point[0]:
                left_point = point
                break
            if direction == 'right' and point[0] <=
old_point[0]:
                right_point = point
                break
            old_point = point    cv2.line(copy_img,
(int(apex_point[0]), int(apex_point[1])),
(int(right_point[0]), int(right_point[1])), (255, 0, 0), 2)
    cv2.line(copy_img, (int(left_point[0]),
int(left_point[1])), (int(right_point[0]),
int(right_point[1])), (255, 0, 0), 2)
    cv2.line(copy_img, (int(apex_point[0]),
int(apex_point[1])), (int(left_point[0]), int(left_point[1])),
(255, 0, 0), 2)    cv2.circle(copy_img, (int(apex_point[0]),
int(apex_point[1])), 2, (0,255,0), 20)
    cv2.circle(copy_img, (int(left_point[0]),
int(left_point[1])), 2, (0,255,0), 20)
    cv2.circle(copy_img, (int(right_point[0]),
int(right_point[1])), 2, (0,255,0),
20)#     plt.imshow(copy_img)
#     plt.show()
    mitral_length = np.sqrt(np.power((left_point[0]-
right_point[0]), 2)+np.power((left_point[1]-right_point[1]),
2))
    # Find mitral point
    mitral_point = [int(left_point[0]+mitral_length/2),
```

```
int(left_point[1]-(left_point[1]-right_point[1])/2)]
    return apex_point, mitral_point, cntsdef
find_diameters_and_length(img):
    test_img =
cv2.cvtColor(img[:,:,0],cv2.COLOR_GRAY2RGB)    apex_point,
mitral_point, cnts = find_apex_mitral_points(img)
    length = int(np.sqrt(np.power((apex_point[0]-
mitral_point[0]), 2)+np.power((apex_point[1]-mitral_point[1]),
2)))
    disk_height = int(round(length / disks_num))
    diams = []    for i in range(disks_num+1):
        test_img1 = np.zeros(img.shape[0:2])
        test_img2 = np.zeros(img.shape[0:2])
        height = disk_height * (i+1)
        coords = getPerpCoord(apex_point[0], apex_point[1],
mitral_point[0], mitral_point[1], img.shape[1],
height)        cv2.drawContours(test_img1, cnts, 0, 1, 2)
        cv2.line(test_img2, (coords[0], coords[1]),
(coords[2], coords[3]), 1, 1)
        cv2.line(test_img, (coords[0], coords[1]), (coords[2],
coords[3]), 1, 1)        intersection =
np.logical_and(test_img1, test_img2)
        arr = np.nonzero(intersection)
        points = []
        for i in range(len(arr[0])):
            points.append([arr[1][i], arr[0][i]])        try:
            ind_left = np.argmin(np.array(points)[:, 0])
            ind_right = np.argmax(np.array(points)[:, 0])
            diam_arr = [points[ind_left], points[ind_right]]
            diam = np.sqrt(np.power(diam_arr[1][1] -
diam_arr[0][1], 2) + np.power(diam_arr[1][0] - diam_arr[0][0],
2))
            diams.append(diam)
        except:
            diams.append(diams[-1])        cv2.line(test_img,
(apex_point[0], apex_point[1]), (mitral_point[0],
mitral_point[1]), (0, 0, 0), 1)
        cv2.circle(test_img, (apex_point[0], apex_point[1]),
1, (0, 255, 0), 2)
        cv2.circle(test_img, (mitral_point[0],
mitral_point[1]), 1, (0, 255, 0), 2)
    fig = plt.figure(figsize=(5,5))
    plt.imshow(test_img)
    plt.show()
    return np.array(diams), length

ef_results = []
count = 0
for index, value in enumerate(a2c_es_mask_list):
    try:
        mhd_mask_file = a2c_es_mask_list[index]
        _a2c_es_mask =
getLVMask(mhd_mask_file)        mhd_mask_file =
```

```
a2c_ed_mask_list[index]
        _a2c_ed_mask =
getLVMask(mhd_mask_file)        mhd_mask_file =
a4c_es_mask_list[index]
        _a4c_es_mask =
getLVMask(mhd_mask_file)        mhd_mask_file =
a4c_ed_mask_list[index]
        _a4c_ed_mask =
getLVMask(mhd_mask_file)        diams_ed_a4c, length_ed_a4c =
find_diameters_and_length(_a4c_ed_mask)
        diams_ed_a2c, length_ed_a2c =
find_diameters_and_length(_a2c_ed_mask)
        diams_es_a4c, length_es_a4c =
find_diameters_and_length(_a4c_es_mask)
        diams_es_a2c, length_es_a2c =
find_diameters_and_length(_a2c_es_mask)        a2c_ed_mask,
a2c_es_mask, a4c_ed_mask, a4c_es_mask =
to_same_zoom(_a2c_ed_mask, _a2c_es_mask, _a4c_ed_mask,
_a4c_es_mask,
                        length_ed_a2c, length_es_a2c,
length_ed_a4c, length_es_a4c)        diams_ed_a4c,
length_ed_a4c = find_diameters_and_length(a4c_ed_mask)
        diams_ed_a2c, length_ed_a2c =
find_diameters_and_length(a2c_ed_mask)
        diams_es_a4c, length_es_a4c =
find_diameters_and_length(a4c_es_mask)
        diams_es_a2c, length_es_a2c =
find_diameters_and_length(a2c_es_mask)        length_ed =
max(length_ed_a2c, length_ed_a4c)
        length_es = max(length_es_a2c,
length_es_a4c)        volume_ed = 0
        for ind in range(disks_num):
            volume_ed += (np.pi/4) * diams_ed_a2c[ind] *
diams_ed_a4c[ind] * int(math.ceil(length_ed / disks_num))
        volume_es = 0
        for ind in range(disks_num):
            volume_es += (np.pi/4) * diams_es_a2c[ind] *
diams_es_a4c[ind] * int(math.ceil(length_es /
disks_num))        calc_ef = ((volume_ed - volume_es) /
volume_ed) * 100
        real_ef = "Unknown"
        f = open(info_list[index], "r")
        for x in f:
            if('LVef' in x):
                real_ef = x[6:].strip()
        print("Predicted EF: calc - {}   ".format(calc_ef),
"Real EF: {}".format(real_ef))
        ef_results.append([int(round(calc_ef)),
round(float(real_ef))])
    except Exception as ex:
         print("{}. Error occures {}".format(index+1, ex))
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	94 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Даниленко К.О.			Автоматичний розрахунок фракції викиду лівого шлуночка серця за допомогою моделей глибинного навчання	Шифр групи	Код напр./спец.
Перевірив		Терзіян В.Я.				СШІм-19-2	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					