基于 Python 实现的高斯混合模型(GMM)背景检测算法

本文基于 Python 环境，利用 NumPy，OpenCv-Python 等工具，按步骤复现了高斯混合模型算法，并在测试图片上进行了实验，展示了各个步骤的可视化结果。最后，本文利用封装后程序，完成了测试数据上的背景检测工作，并生成了对比视频。

1. 开发环境的搭建

1.1 远程服务器上 mini-conda 开发环境的部署

由于个人电脑配置较差，内存空间不足。本文在实验室中安装了 Ubuntu22.04 的远程服务器中搭建 Python 开发环境。为了不影响服务器系统内置 Python 环境的使用，为服务器安装 mini-conda 工具，并在 mini-conda 中建立虚拟环境运行 Python 解释器。

首先，利用 ssh 工具连接到远程服务器，打开远程服务器终端。使用 wget 工具下载 mini-conda 安装包，并运行安装程序。

```
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ sh Miniconda3-latest-Linux-x86_64.sh
```

安装程序运行结束后，更新系统环境变量，便可以在命令行中使用 conda 工具管理系统中的 Python 环境。

```
$ source ~/.bashrc
```

在 mini-conda 中新建一个 Python 开发环境，并指定 Python 版本为 3.10。

```
$ conda create -n py310 python==3.10
```

等待片刻后，新的虚拟环境便配置完成了。激活这个虚拟环境，并安装本实验所需的 Python 软件包。NumPy 包用于处理矩阵数据，OpenCv-Python 包提供了 OpenCv 图像处理工具的 Python 接口，Matplotlib 包提供了图像与曲线可视化的功能，ipykernel 包提供了 IPython 接口，方便本实验在 Jupyter Notebook 环境下进行代码的调试开发工作。

```
$ conda activate py310
$ pip3 install numpy opencv-python matplotlib ipykernel
```

等待片刻后，实验所需的软件包安装完成，mini-conda 开发环境部署完毕。

1.2 在 Visual Studio Code 中进行开发

Visual Studio Code 为远程服务器开发提供了丰富的支持。可以在远程服务器中实现代码着色，自动补全，文件跳转，调试运行等功能。在 Visual Studio Code 中搜素“Remote-SSH”插件并安装。根据指引输入远程服务器的用户名，密码，配置 SSH 密钥后，便可直接在本机的 Visual Studio Code 软件中对远程服务器上的代码项目进行开发。

图 1.2 Visual Studio Code 中的“Remote-SSH”插件

正确连接到服务器后，在终端中新建目录，安装 Jupyter Notebook 软件进行项目的调试开发。Jupyter Notebook 提供了基于 IPython 的实时交互开发环境，可以方便地对图表进行可视化，查看工作空间中的变量；同时提供了基于 Markdown 与 Latex 的文本工具，方便整理研究笔记与实验结果，被广泛应用在 Python 数据科学研究中。

$ conda install jupyter notebook

$ conda install nb_conda_kernels

$ conda activate py310

配置完毕后，VSCode 能够自动识别到远程服务器中安装 Python 环境，运行 Jupyter Notebook 文件中的代码片段。

图 1.3 Visual Studio Code 中选择需要使用的 Python 解释器

2. 混合高斯模型（GMM）算法的实现

2.1混合高斯模型与多维高斯模型

一段连续视频可以看作是多张彩色图像按照时间进行的叠加。从时间维度来看，每个像素的色彩在随着时间不断变化。而其变化似乎能够利用统计学分析其分布及规律。当不考虑色彩时，可以暂时将某个像素的亮度（灰度）其在时间上的分布规律总结为某种一维分布规律。最常见的分布便是高斯分布，可描述为$X\sim N(\mu,\sigma)$，其中μ代表该分布的均值，而σ代表该分布的方差。但单个高斯分布难以描述现实中较复杂的分布规律。此时，通过将多个均值方差不同的高斯分布线性加权累加来描述该分布规律，可描述为$X\sim \Sigma_{i=0}^{k}\alpha_i\cdot N_i(\mu_i,\sigma_i)$，表示样本 X 的分布服从多个不同的高斯分布。其中α_i代表服从某个高斯分布的概率，因此有$\Sigma_{i=0}^{k}\alpha_i=1$。如图 2.1 (a) 所示，X 的分布由三种不同的高斯分布累积而成，形成一种多峰型的分布。

而最常见的彩色图像多由三个色彩通道组成，即红色通道，绿色通道及蓝色通道。在 OpenCv 中，未经压缩的原始图像数据可表示为内存中具有特定形状的张量，形状为[h, w, c]。h 代表图像的高度，w 代表图像的宽度，而 c 代表图像的色彩通道数量。此时可认为每个像素具有 c 维的长度，需要用高维的高斯分布来表示彩色像素的分布规律。高维的高斯分布除了在每个维度上具有均值外，其各个维度的信息有时具有一定的相关性，用需要用协方

差Σ来表示，其中协方差Σ是一个 $c \times c$ 的方阵。图 2.1 (b) 展示了两个具有不同协方差的二维高斯分布在三维空间中的可视化形状。

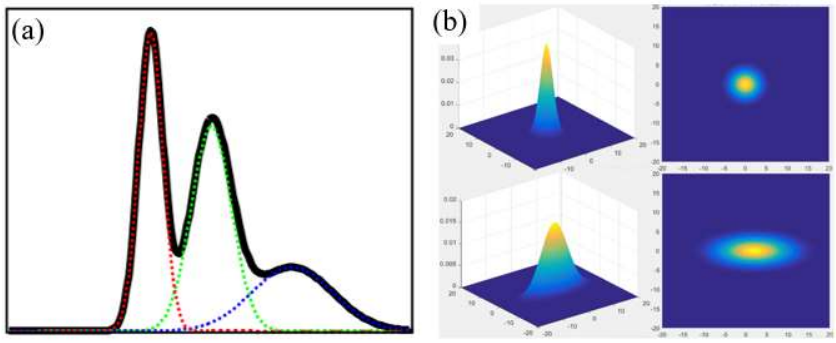

图 2.1 高斯混合分布示意图；(a) 一维混合高斯分布；(b) 高维(二维)高斯分布

连续视频中的像素则刚好满足：1、适合用多组高斯分布混合来表示其分布；2、每个像素具有多维相关信息。则适用于使用多维的混合高斯模型来描述这些彩色像素的分布规律，可以表示为：$X^c \sim \Sigma_{i=0}^{k} \alpha_i \cdot N_i(\mu_i^c, \Sigma_i)$，其中像素$X^c$及每个高斯分布的均值$\mu_i^c$都是 c 维的。

2.2 GMM 在 Python 中的建模及初始化

多维高斯混合模型在处理背景检测任务时，对每个像素的色彩分布都进行了多维建模，有 $w \times h$ 个像素，每像素具有三组色彩强度值，如图 2.2 (a) 所示。设每个像素都具有 k 组 c 维高斯分布混合，对该图像进行建模，则每个 c 维高斯分布具有$c \times c$维的协方差Σ^c，如图 2.2(b)所示，具有 C 维的均值μ^c，如图 2.2 (d) 所示；同时，混合高斯模型由多组分布加权，因此每个高斯分布都带有权重值，如图 2.2 (c)所示。

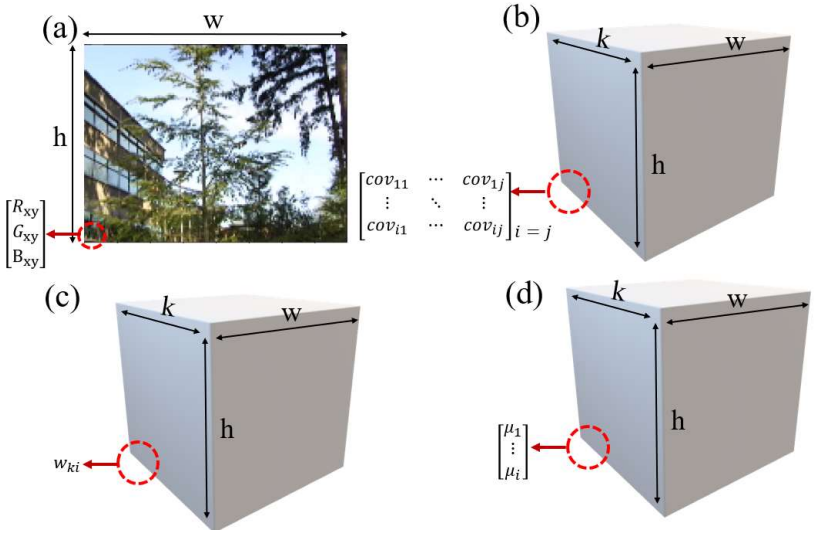

图 2.2 GMM 中的参数示意图：(a) 彩色图像；(b) 模型协方差；(c) 模型权重；(d) 模型均值

在 Python 中，OpenCv-Python 提供了处理图像常用的方法及接口，而 NumPy 模块提供了多维数组类型，适合用于处理 GMM 模型中的多维数据，并对数组进行整块运算。由于其运算功能由底层的 C 语言封装实现，Python 仅进行逻辑调用，编写得当的 Python 脚本也能具备较好的计算性能。

利用 OpenCv-Python 提供的图像读取方法将数据集的第一帧读入脚本中，查看图像的数组结构，并可视化这张图像，如图 2.3 所示。该图像的尺寸为120×160像素，具有三个通道。需要注意的是，OpenCv-Python 默认以 GBR 通道顺序读入图像，需要将其色彩通道重新排列为 RGB 顺序，才能得到色彩正常的图像。

图 2.3 读取数据集中的第一帧图像

先建立一个全为零的数组作为模型的初始均值数组。将一个3×3的初始协方差复制到每一个像素的每一个模型上，作为模型的初始协方差数组。模型的权重则利用 NumPy 的方法生成随机深度为 k 的迪利克雷分布，并复制到每一个像素上。图 2.4 展示了初始化后 GMM 模型各个参数数组的形状。进一步地，可以将均值初始化为视频的第一帧图像，加快后续模型的收敛速度。

```
shape  = img_frame_0.shape
k      = 5
lr     = 1e-2
# 使用对角阵建立协方差矩阵，此时的高斯分布是均匀，互不影响的。
cov    = 255. * np.eye(shape[2])

_mat    = get_init_sigma(cov, k, shape)
_weight = get_init_weight(k, shape)
_mu     = get_init_mu(k, shape)

print(f"协方差数组:{_mat.shape}, 均值数组:{_mu.shape}, 权重数组: {_weight.shape}")
```

✓ 0.4s

```
协方差数组:(120, 160, 5, 3, 3), 均值数组:(120, 160, 5, 3), 权重数组: (120, 160, 5)
```

图 2.4 在 Python 中初始化 GMM 的参数矩阵

2.3判断像素是否符合高斯分布

在传统一维高斯分布中，检验某样本是否服从该分布常用的方法有3σ法则，即检验该样本 X 与均值μ的距离是否在3σ以内：

$$\frac{\sqrt{(X-\mu)^2}}{\sigma} < 3$$

而在高维分布中，X 与μ均变为向量形式，其分布特征由协方差方阵 Σ 表示，此时检验方法被拓展为：

$$\sqrt{(x-\mu)^T \Sigma^{-1} (x-\mu)} < 3$$

图 2.5 展示的代码利用 NumPy 的数组计算实现了$\sqrt{(x-\mu)^T \Sigma^{-1} (x-\mu)}$的计算，该部分将图像中的每个像素与 GMM 模型中的 k 个模型均值都进行了计算，并将结果也存储为了多维数组形式。利用 NumPy 对数组进行整块的计算与变换，能够较好地提升代码的运行效能。

```python
def data_processor(img:np.ndarray, mu:np.ndarray, sigma:np.ndarray):
    """返回均值误差，被复制了K遍的图像，高斯模型数量，图像通道数

        输出参数
        ----------
        `error` : array[ h, w, k, c, 1 ]
        `img_k` : array[ h, w, k, c ]
        `k`     : int
        `c`     : int
    """
    if mu.shape[2] == sigma.shape[2]:
        k = mu.shape[2]
    else:
        raise Exception(f"检查参数的形状，它们应该具有相同的高斯模型个数\
            \n\tmu: {mu.shape[2]}, Sigma: {sigma.shape[2]}")
    if mu.shape[-1] == sigma.shape[-1] == img.shape[-1]:
        c = mu.shape[-1]
    else:
        raise Exception(f"检查参数的形状，图像的通道数应该一致\
            \n\timg: {img.shape[-1]}, mu: {mu.shape[-1]}, \
            Sigma: {sigma.shape[-1]}")
    # 把图像复制k遍（分配给每个高斯分布）
    img = img[:,:,None,:]
    img = np.repeat(img, k, 2)
    # （计算每像素与高斯均值的误差）
    erro:np.ndarray = img - mu
    erro = erro[:,:,:,None,:]
    return erro, img, k, c


def get_distance(erro:np.ndarray, sigma:np.ndarray):
    """计算图像每像素在各高斯模型上的匹配度

        输出参数
        ----------
        `d` : array[ h, w, k, 1 ]
    """
    d = np.sqrt(erro @ np.linalg.inv(sigma) @ np.transpose(erro, (0,1,2,4,3)))
    return d.squeeze(-1)
```

图 2.5 计算每个像素与各个模型均值的相对距离

根据均值的相对距离可以对像素与模型的匹配情况进行判别。如果相对距离小于某个阈值，则可以判定该像素符合该分布，否则不符合该分布。将符合分布表示为 True，不符合分布表示为 False，可以得到代表像素是否匹配的掩码矩阵。这样的矩阵对于接下来的模型预测与模型更新都有帮助，利用 NumPy 的布尔运算符重载，可以很方便的生成掩码矩阵，如图 2.6 所示。

NumPy 掩码矩阵中的布尔数在与矩阵进行运算时，会自动进行类型变换，转换为整型或浮点数组。转换后 True 会变为数值 1，False 会变为数值 0。如果使用掩码矩阵与目标矩阵做乘法，则可将目标矩阵中 False 对应的元素置零，保留 True 位置的元素。这个技巧在后续的计算中经常用到，将逻辑判断转化为矩阵运算，在简化代码的同时，还可以极大优化某些流程的运行速度。

图 2.6 利用相对距离生成匹配掩码

2.4 EM 过程迭代更新模型

在单高斯分布中，被人熟知的通过样本点拟合分布的方法是最大似然估计法。然而高斯混合模型由多组高斯分布组合，由于复杂的数学原因，混合高斯模型在最大似然估计法的某些步骤中是不可导的，无法使用最大似然估计法进行拟合。EM 方法是另一种通过迭代更新来拟合高斯混合模型的算法。

从算法的工程实现上来说，EM 方法可以分为更新与重初始化两个步骤。更新是指改变混合模型的权重分配，改变某些高斯模型的均值及协方差，从而使得高斯混合模型更加匹配样本点的分布；而重初始化是指如果某像素无法与混合模型中的任何一个分布匹配，则利用当前样本点重新初始化高斯模型的均值。实现这个过程的代码并不复杂，主要需要注意各个矩阵计算时其形状必须匹配，同时适当的改变掩码的形状来对需要更新的参数进行过滤。

更新过程，首先找出所有匹配当前像素的高斯模型，并生成掩码。增大能够匹配当前像素的模型权重，并降低不能匹配当前像素的模型权重，表示为

$$weight = weight + lr \times (mask - weight)$$

其中当模型匹配时$mask = 1$，否则$mask = 0$，lr为算法的学习率。第二步，更新均值矩阵，找出匹配当前样本的模型，并调整它的均值，让它的分布更接近当前样本，表示为：

$$\mu = \mu + mask \times \left(lr \,/\, (waight + 10^{-5})\right) \times (x - \mu)$$

分母中的10^{-5}是为了避免模型在计算的过程中出现过大的异常数值。第三步，更新协方差矩阵，表示为：

$$\Sigma = \Sigma + mask \times \left(lr/(weight + 1e - 5)\right) \times \left((x - \mu) \cdot (x - \mu)^T - \Sigma\right)$$

完整的更新过程代码实现如图 2.7 所示。

```python
erro, img_k, k, c = data_processor(img, mu, sigma)
mask = get_mask(get_distance(erro, sigma))

# -------------更新过程-----------------
#更新权重矩阵
# 如果像素匹配某分布，则增大该分布的权重，否则减少权重。
weight = weight + lr * (mask - weight)
# 将掩码和权重在通道上复制
waight_mu = np.repeat(weight[:,:,:,None], c, -1)
mask_mu   = np.repeat(mask[:,:,:,None],   c, -1)
# 更新均值矩阵
# 如果像素匹配某分布，改变该分布的均值使其更接近该点
mu = mu + mask_mu * (lr / (waight_mu + 1e-5)) * erro.squeeze(-2)
# 将掩码和权重复制为方阵
weight_sigma = np.repeat(waight_mu[:,:,:,:,None], c, -1)
mask_sigma   = np.repeat(mask_mu[:,:,:,:,None],   c, -1)
# 更新协方差矩阵
sigma = sigma + mask_sigma * (lr / (weight_sigma + 1e-5)) * \
        (np.transpose(erro, (0,1,2,4,3)) @ erro - sigma)
```

图 2.7 EM 方法中的更新过程

重初始化过程，找出没有匹配到任何分布的像素的位置，将对应位置的混合模型中，权重最低的那个高斯分布重初始化为当前像素的分布。为了实现这个过程，需要找到两个掩码：没有匹配任何高斯分布的像素掩码 *pix_unmatched_mask* 和最小权重高斯分布的位置掩码 *min_weight_mask*。它们可以用 NumPy 高效简洁地实现，如图 2.8 所示。

```python
def get_unmatched(mask:np.ndarray)->np.ndarray:
    """
        找到没有匹配任何模型的像素掩码，该像素未匹配任何模型为True

        输入参数
        ----------
        `mask` : array[ h, w, k ], dtype='bool'

        输出参数
        ----------
        `unmatched` : array[ h, w ], dtype='bool'
    """
    unmatched = np.sum(mask, -1)
    unmatched = unmatched == 0
    return unmatched
```

```python
def get_min_mask(weight):
    """
    返回权重最小掩码，权重最小的位置为True

    输入参数
    ----------
    `weight` : array[ h, w, k ]

    输出参数
    ----------
    `min_mask` : array[ h, w ], dtype='bool'
    """
    min_map  = np.argmin(weight, -1)
    # 分配内存
    min_mask = np.full(weight.shape, False, bool)
    # 将掩码在模型数量上堆叠
    for k in range(weight.shape[-1]):
        k_mask = min_map == k
        min_mask[:,:,k] = k_mask
    return min_mask
```

图 2.8 获取未匹配像素掩码及最小权重模型掩码

获取掩码后，利用与掩码相乘的特性，可以快捷对需要的元素进行处理，而不影响别的元素。以重初始化均值的算法$\mu = \mu + \left(mask \times (x - \mu)\right)$为例，当$mask = 1$时，公式化简为$\mu = x$，表示将该均值替换。当$mask = 0$时，化简为$\mu = \mu$，表示不进行任何操作。重初始化的实现代码如图 2.9 所示。

```
# -------------重初始化过程-----------------
# 获取没有匹配任何分布的像素掩码,最小权重所在位置的掩码
pix_unmatched_mask = get_unmatched(mask)
min_weight_mask = get_min_mask(weight)
pix_unmatched_mask = np.repeat(pix_unmatched_mask[:,:,None], k, -1)
reset_mask = pix_unmatched_mask * min_weight_mask
# 将未匹配像素中权值最小的像素用当前图像进行初始化
# 在通道上拓展掩码
reset_mask = np.repeat(reset_mask[:,:,:,None],   c, -1)
# 重初始化均值
mu = mu + (reset_mask * ( img_k - mu ) )
reset_mask = np.repeat(reset_mask[:,:,:,:,None],   c, -1)
init_sigma = get_init_sigma(init_cov, k, img.shape)
# 重初始化协方差
sigma = sigma + (reset_mask * ( init_sigma - sigma ) )
```

图 2.9 未匹配像素的最小权重模型的重初始化

2.5 GMM 模型的训练

实现模型的匹配及参数更新算法后，模型的训练实现较简单。首先是对图像数据按时间进行排序，并清理无关的文件。读出首张图像，根据首张图像初始化模型，生成参数矩阵。在训练的过程中逐张读入后续图像，并更新模型参数。在训练前将所有图像全部读入内存可能能带来更快的训练速度，但会占用更多的内存，需要按需选择不同的数据读取方案，这里选择逐张读入图片，如图 2.10 所示。

```
def train(num_steps:int, k:int, lr:float, data_path="./DataSet/WavingTrees/"):
    """训练GMM模型"""
    file_names = os.listdir(data_path)
    file_names.sort()
    file_names = file_names[:-2]

    frame_0 = cv2.imread(os.path.join(data_path, file_names[0]))
    shape  = frame_0.shape
    # -----------模型的初始化-----------------
    cov    = 255. * np.eye(shape[2])
    sigma  = get_init_sigma(cov, k, shape)
    weight = get_init_weight(k, shape)
    mu     = np.repeat(frame_0[:,:,None,:], k, 2)
    # -----------训练模型---------------------
    for i in tqdm(range(1, num_steps+1)):
        frame = cv2.imread(os.path.join(data_path, file_names[i]))
        mu, sigma, weight = step(mu, sigma, weight, frame, lr, cov)

    return mu, sigma, weight
```

图 2.10 GMM 模型的训练过程

选取数据集的前 150 张图片进行训练，使用 5 层的混合高斯模型，并使用 0.00001 的学习率。得益于 NumPy 对张量计算的优化，模型的训练速度十分理想，达到了 23.5 帧/秒。

```
mu, sigma, weight = train(150, 5, 1e-5)
✓ 6.5s
100% 150/150 [00:06<00:00, 23.49it/s]
```

图 2.11 模型的训练

3. 混合高斯模型（GMM）的效果验证

3.1图像的识别与渲染

将图像中的每个像素对其混合高斯分布进行匹配。能够匹配到分布的被识别为背景，不能匹配到分布的则为前景。利用上文实现的掩码算法可以轻易完成这样的识别，并输出结果掩码。在这里将背景设置为 False，方便后续通过掩码计算在可视化的过程中直接消除背景的像素值，如图 3.1 所示。

```
def forward(mu, sigma, img):
    """
    返回识别结果掩码，背景为False

    输入参数
    ----------
    `mu`    : array[ h, w, k, c ]
    `sigma` : array[ h, w, k, c, c ]
    `img`    : array[ h, w, k, c ]

    输出参数
    ----------
    `mask` : array[ h, w ], dtype='bool'

    """
    erro, _, _, _ = data_processor(img, mu, sigma)
    return get_unmatched(get_mask(get_distance(erro, sigma)))
```

图 3.1 通过高斯模型预测背景

识别结果的可视化则使用识别得到的掩码直接与原图相乘，如图 3.2 所示。由于 OpenCv 读入的图像是 BGR 格式的，需要将图片的通道变换得到正常色彩的结果。

```
def render(mu, sigma, img, BGR2RGB=False):
    """
    将图像进行可视化渲染，背景变为黑色

    输入参数
    ----------
    `mu`    : array[ h, w, k, c ]
    `sigma` : array[ h, w, k, c, c ]
    `img`   : array[ h, w, k, c ]

    输出参数
    ----------
    `img_rendered` : array[ h, w, c ], dtype='int'
    """
    mask = forward(mu, sigma, img)
    out:np.ndarray = (img * np.repeat(mask[:,:,None], img.shape[-1], -1)).astype(int)
    if BGR2RGB:
        out = out.swapaxes(0,2)
        out = out[[2,1,0]]
        out = out.swapaxes(0,2)
    return out
```

图 3.2 通过掩码可视化分割结果

读入一张图像查看分割模型的识别效果。利用 matplotlib 库在 Jupyter Notebook 的输出区域直接展示图像，发现模型将静止的背景与运动的人物和随风摇摆的树木区分而出，如图

3.3 所示。

```python
test_frame = cv2.imread(f"./DataSet/WavingTrees/b00252.bmp")
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 20))
ax[0].imshow(cv2.cvtColor(test_frame, cv2.COLOR_BGR2RGB))
ax[0].axis("off")
ax[1].imshow(render(mu, sigma, test_frame, True))
ax[1].axis("off")
```

Python

```
(-0.5, 159.5, 119.5, -0.5)
```

图 3.3 可视化分割结果

3.2 生成视频

Matplotlib 库提供了直接在 Jupyter Notebook 中打开媒体播放器，并播放视频的功能，十分便利。编写一个方便调用的函数，将一系列的图像数组拼接为视频并展示出来，如图 3.4 所示。

```python
def display_video(frames:list, framerate:int=30, dpi:int=70):
    '''
        在Jupyter Notebook页面中生成视频
    '''
    height, width, _ = frames[0].shape
    orig_backend = matplotlib.get_backend()
    matplotlib.use('Agg')
    fig, ax = plt.subplots(1, 1, figsize=(width / dpi, height / dpi))
    matplotlib.use(orig_backend)
    ax.set_axis_off()
    ax.set_aspect('equal')
    ax.set_position([0, 0, 1, 1])
    im = ax.imshow(frames[0])
    def update(frame):
      im.set_data(frame)
      return [im]
    interval = 1000/framerate
    anim = animation.FuncAnimation(fig=fig, func=update, frames=frames,
                                   interval=interval, blit=True, repeat=False)
    return HTML(anim.to_html5_video())
```

图 3.4 将图像序列生成视频

生成原图与识别结果的对比视频，可以生成一块宽度为二倍的数组，将原图与生成的结果在该数组上进行拼接。利用 NumPy 的矩阵切片替换特性可以快捷实现，如图 3.5 所示。生成的视频会直接在 Jupyter Notebook 的输出栏中展示出来，可以直接通过 IDE 或浏览器的内嵌播放器进行播放和存储，十分方便。

图 3.5 Jupyter Notebook 内嵌视频播放器

4. 总结

本实验耗时很长完成，通过这次实验本人有机会详尽学习了视频图像处理中的相关统计学原理，受益颇丰，也被传统的机器学习算法优雅而严谨的理论过程所吸引。在实现时尝试使用 NumPy 的矩阵与数组计算特性完成，也惊讶于其丰富完整的计算特性，简洁美观的代码及高效的计算性能，就算使用以运行速度慢而被诟病的 Python 语言，也能写出性能良好的算法。

附录

代码运行的文件结构如图 5.1 所示，采用相对路径放置，数据集在*./DataSet/WavingTrees/*目录下。主程序用 Jupyter Notebook 开发环境编写，保存在*./GMM_Detect.ipynb* 文件中。使用 VSCode 等支持 Python 的主流 IDE 打开后，能够看到算法的实现过程及中间运行结果。

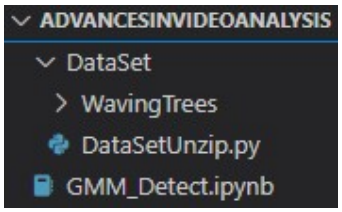

图 5.1 文件结构

算法最终输出的视频及方便阅读的 Jupyter Notebook 笔记本 pdf 打印版存储在*./media* 目录中。