Image Restoration

基于线性回归与均值的图像恢复/降噪

徐震 3180105504

May 2020

目录

1 前言 **3**

2 实验环境 **3**
- 2.1 硬件环境 . . . 3
- 2.2 软件环境 . . . 3

3 实验原理 **4**
- 3.1 添加噪声 . . . 4
  - 3.1.1 生成噪声遮罩 . . . 4
  - 3.1.2 生成噪声图片 . . . 4
  - 3.1.3 添加高斯噪声 . . . 5
- 3.2 消除噪声 . . . 5
  - 3.2.1 均值替换算法 . . . 5
  - 3.2.2 线性回归预测 . . . 6
  - 3.2.3 Haar 小波降噪 . . . 7
  - 3.2.4 对高斯噪声的处理 . . . 7

4 实验步骤 **8**
- 4.1 添加噪声 . . . 8
- 4.2 消除噪声 . . . 8
  - 4.2.1 均值替换算法 . . . 9
  - 4.2.2 线性回归预测 . . . 12
  - 4.2.3 Haar 小波降噪 . . . 13
  - 4.2.4 对高斯噪声的处理 . . . 15

5 实验结果 **16**
- 5.1 普通测试 . . . 16
  - 5.1.1 测试结果 . . . 16
  - 5.1.2 结果分析 . . . 18
- 5.2 对比测试 . . . 18
  - 5.2.1 测试结果 . . . 18
  - 5.2.2 结果分析 . . . 19

A 辅助代码与全部算法实现 **20**

B 插图，表格与列表 **37**

1 前言

图像是一种非常常见的信息载体，但是在图像的获取、传输、存储的过程中可能由于各种原因使得图像受到噪声的影响。如何去除噪声的影响，恢复图像原本的信息是计算机视觉中的重要研究问题。

常见的图像恢复算法有基于空间域的中值滤波、基于小波域的小波去噪、基于偏微分方程的非线性扩散滤波等，在本次实验中，我们要对图像添加噪声，并对添加噪声的图像进行基于模型的去噪。为了实验过程的简洁方便，我们采用一种较为特殊的噪声对图像进行加噪处理：

- 我们首先创建一个与原图大小相同的噪声遮罩。
- 遮罩的值域为 $mask \in \{0, 1\}$
- 接着我们遮罩与原图逐元素相乘，$noise\_img = img \otimes mask$
- 容易发现，遮罩中为零的区域的像素被抹除，其余像素被保留。

2 实验环境

2.1 硬件环境

CPU `Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59GHz`
`Cores: 6, Logical Processors: 12`

Memory `SODIMM 15.8GB/16GB`

2.2 软件环境

System `Microsoft Windows [Version 10.0.18363.836]`

Conda `conda 4.8.2`

Python `Python 3.7.6`

Runtime Env `ipython 7.10.2 / jupyter-notebook 6.0.2`

IDE `Pycharm / Visual Studio Code`

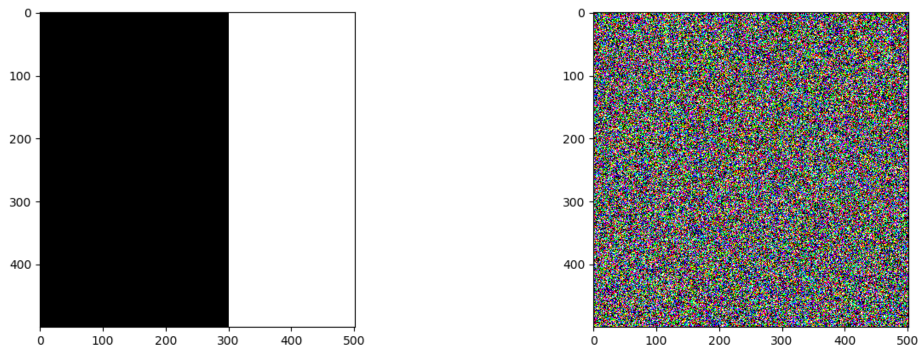
3 实验原理

3.1 添加噪声

3.1.1 生成噪声遮罩

我们通过前言中描述的方法对一张普通图片添加噪声。具体实现中，为了避免下标来回转换，我们采取了一种新的随机噪声遮罩生成函数。这种函数有着随机比例完全确定的优点，生成的噪声图片更为稳定。

- 我们首先创建与原图片等大小的全 1 噪声遮罩。
- 接着我们将每一行的前 `noise_ratio*img.shape[1]` 个像素填为 0。实际实现中我们采用 `numpy` 提供的向量化运算来加速这一过程
- 最后我们对每一通道的每一行数据调用 `shuffle` 函数，噪声遮罩如图1所示。
- 由于我们在进行随即操作前就已确定遮罩中 0 值的数目，这一随机过程是极为稳定的。

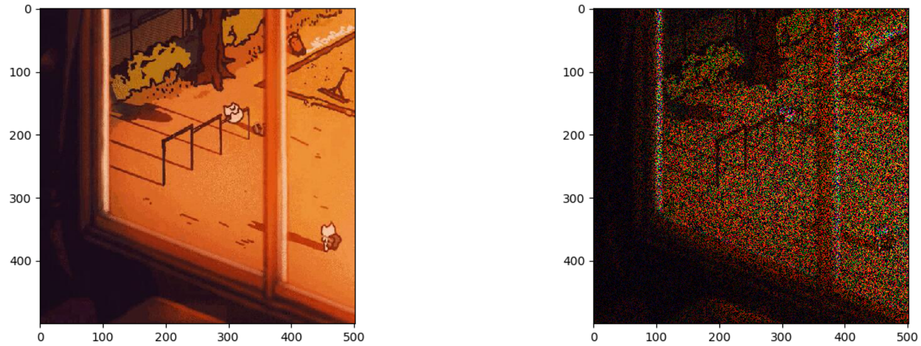


(a) 未随机化的噪声遮罩 (b) 随机化后的噪声遮罩

图 1: 噪声遮罩 ($noise\_ratio = 0.6$)

3.1.2 生成噪声图片

在生成噪声遮罩后，我们执行 $noise\_img = img \otimes mask$ 来获得噪声图像，噪声图像如图2所示。



(a) 原图像 (b) 噪声图像

图 2: 噪声图像 ($noise\_ratio = 0.6$)

3.1.3 添加高斯噪声

在实验中我们并没有止步于一定比例的椒盐噪声，我们还使用了正太分布为图片添加了高斯噪声，以进行鲁棒性的图片降噪/加噪学习。我们通过对图片的每个像素点添加一定的扰动来为图片添加高斯噪声，像素点上的扰动符合高斯分布（正态分布），因此这种噪声被称为高斯噪声。

3.2 消除噪声

如前言中所述，常见的图像恢复算法有基于空间域的中值滤波、基于小波域的小波去噪、基于偏微分方程的非线性扩散滤波等。但本次实验中的噪声有着极为鲜明的特点：**所有噪声像素点的亮度值都为 0**。基于此特点，我们甚至可以完全恢复噪声添加过程中的噪声遮罩，并基于噪声遮罩进行精细的图像恢复。

在具体实验中，均值替换算法和线性回归预测是我们实际应用的算法。*Haar* 小波降噪是为了与上述方法做出对比而实现的一种普适降噪算法。在测试过程中，我们还调用了 `OpenCV` 提供的其他普通降噪方法。

3.2.1 均值替换算法

在均值替换算法中，我们将每一个噪声点[1]的值恢复为其周围有效点的像素均值。

让我们以 P 来标记某一噪声点，x_p, y_p 为该点在图片矩阵中的坐标，如图3所示。为了简化描述过程，我们暂不考虑通道带来的影响。

[1] 即噪声遮罩中的 0 值点

Invalid Points:Red Valid Points:Green

```
array([[0.09803922, 0.        , 0.        , 0.09803922],
       [0.        , 0.        , 0.        , 0.09803922],
       [0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.09803922, 0.09803922]])
```

Target Point:Blue

图 3: 噪声点周围环境 ($radius = 2$)

- 首先我们计算出 $[x_p - radius, x_p + radius) \times [y_p - radius, y_p + radius)$ 所表示正方形中所有像素点值之和（在噪声图片上计算）。这里 $radius$ 是一个提前定义好的窗口大小。
- 由于本实验中噪声的特殊性，上述求和结果即为有效点的像素值之和[2]。
- 接着我们对图片进行噪声遮罩提取（或使用已经提取好的噪声遮罩），并用同样的步骤计算像素点值之和。
- 该和即为此窗口（$[x_p - radius, x_p + radius) \times [y_p - radius, y_p + radius)$）中有效像素点的数目[3]。
- 最终我们通过上述两和相除得到噪声点周围有效点的均值。

我们采用两次求和的原因在于，这样的算法步骤可以更大程度上利用 `numpy` 提供的向量化加速运算功能，免去不必要的循环与判断。与一般图像恢复/噪声消除算法不同，我们的算法利用了实验中噪声的特点，进行了点对点的噪声消除。

3.2.2 线性回归预测

类似于均值替换算法，在线性回归预测中我们同样只对噪声点进行恢复操作。其中噪声点的周围环境如图2所示。

- 我们首先通过下标操作取得该周围环境。
- 接着我们提取所有有效点的坐标为训练集 X 值。
- 并提取所有有效点的像素值作为训练集 Y 值。
- 接着我们通过 *LinearRegression*, *Ridge*, *Lasso*, *ElasticNet* 等线性回归模型对此小范围内的像素分布进行拟合。

[2]所有噪声点值都为零
[3]在噪声遮罩中所有噪声点值都为 0，所有有效点值都为 1

- 最后我们通过问题点的坐标对其像素值进行预测。

类似的，与一般图像恢复/噪声消除算法不同，线性回归预测算法也利用了实验中噪声的特点，进行了点对点的噪声消除。

3.2.3 Haar 小波降噪

我们实现 *Haar* 小波降噪算法的原因主要在于同上述两个点对点的图像恢复算法形成对比。

能够通过 Haar 小波变换进行降噪的主要原因在于：

- 一轮 Haar 小波变换可以很好的将图片的高频信号提取出来。
- 噪声信号往往出现在图片的高频区域，通过设置阈值将 Haar 小波变换后的高频信号进行滤波，我们可以获得降噪后的图片[4]。

在最终实现中，我们通过如下算法进行降噪操作：

- 对图片进行补边操作使得 Haar 小波变换能够正常进行。
- 对图片进行 Haar 小波变换。
- 将变换后的图片矩阵中小于一定阈值的元素置 0。
- 对置 0 调整后的图片矩阵进行 Haar 小波逆变换。
- 根据先前的补边结果对图片进行削边操作。

3.2.4 对高斯噪声的处理

同样的，与上述添加高斯噪声环节对应，我们采用了

- 均值滤波（即均值/方块模糊）。
- 中值滤波（即中指模糊）。
- 高斯模糊（或称高斯滤波）。

等方法来消除上述添加的高斯噪声。

[4]让我们假设所有高频信号都被置为 0，则进行 Haar 小波逆变换后的图像就相当于进行了一次缩放均值模糊

4 实验步骤

4.1 添加噪声

在添加噪声环节中，我们用 Python 实现了上述随机化算法，如列表1所示：

```python
def noise_mask_image(img, noise_ratio):
    # copy the original image (different memory location)
    noise_img = np.copy(img)
    # initialization
    noise_mask = np.ones_like(noise_img, dtype='double')
    # mask image according to the ratio
    noise_mask[:, :round(noise_img.shape[1] * noise_ratio), :] = 0.
    # shuffle every row in every channel
    for channel in range(noise_img.shape[2]):
        for row in range(noise_img.shape[0]):
            np.random.shuffle(noise_mask[row, :, channel])
    noise_img = noise_img * noise_mask
    return noise_img
```

Listing 1: 根据比例添加随机噪声

我们使用如下列表2所示的代码添加高斯噪声。

```python
def gaussian_noise_image(img, var=0.1, mean=0):
    # adding mean gaussian noise
    noise = np.random.normal(mean, var, img.shape)
    noise_img = img + noise
    # noise_img += -np.min(noise_img)
    noise_img /= 1. if img.dtype == np.double else 255
    noise_img = np.clip(noise_img, 0., 1.)
    return noise_img
```

Listing 2: 添加高斯噪声（noise_variance = max * 0.2）

4.2 消除噪声

为了提高噪声消除算法的模块化程度与代码复用率，我们将一些公用功能封装到了全局函数中，如列表3所示，这两个函数会处理小矩形的计算范围，免函数本体于下标越界之苦，并给予边界处同样的关照：

```python
def in_range_one(row, rows, size):
    row_beg = row - size if row - size >= 0 else 0
```

```python
    row_end = row + size if row + size < rows else rows - 1
    return row_beg, row_end

def in_range_two(row, col, rows, cols, size):
    return np.array((in_range_one(row, rows, size), in_range_one(col, cols,
    ↪  size))).flatten()
```

Listing 3: 公用函数：对边角的统一处理

为了利用现代 CPU 的多核性能，我们还在普通算法的基础上，实现了均值替换算法和线性回归预测的多核版本。我们将会在每个算法的语境下具体分析。

4.2.1 均值替换算法

核心 均值替换算法的核心部分如列表4所示。

```python
def mean(row, col, channel, rows, cols, size, noise_img, noise_mask):
    # we introduce a while(1) loop so that we can expand our search windows until one
    ↪  with valid pixel(s) is found
    while True:
        # considering the boundary, and transfer the square horizontally and
        ↪  vertically
        row_beg, row_end, col_beg, col_end = in_range_two(row, col, rows, cols, size)

        # mean values is sum(all pixels)/sum(noise mask)
        # since white point won't affect total sum and sum of noise mask indicates
        ↪  number of valid pixels
        # of course we can get number of valid positions from noise_img directly
        # but in practice, computing this from sum of noise_mask proves to be much
        ↪  faster
        number = np.sum(noise_mask[row_beg:row_end, col_beg:col_end, channel])
        # we update size and continue loop before computing "total", which saves time
        if number == 0.:
            size *= 2
            continue
        total = np.sum(noise_img[row_beg:row_end, col_beg:col_end, channel])
        return total / number
```

Listing 4: 均值计算

单核实现 我们实现了均值替换算法的单核部分，如列表5所示。

```python
def restore_by_mean(noise_img, size=1):
    # copy the original image (different memory location)
    res_img = np.copy(noise_img)
    # obtain noise_mask
    noise_mask = get_noise_mask(noise_img)
    # obtain shape of image
    rows, cols, channels = noise_img.shape
    # obtain noise points, as np.array "white"
    whites = np.argwhere(noise_mask == 0.)
    # use a progress bar to indicate progress
    with ProgressBar(max_value=len(whites)) as bar:
        for i, (row, col, channel) in enumerate(whites):
            res_img[row, col, channel] = mean(row, col, channel, rows, cols, size,
            ↪ noise_img, noise_mask)
            bar.update(i)
    return res_img
```

Listing 5: 单核均值替换降噪

多核实现 我们通过 `multiprocess` 包提供的接口实现了该算法的多核版本。值得注意的是我们在处理图片数据时采用了共享内存，这在一定程度上可以提高子进程的生成速度，并且精细的调教可以让我们免于合并多个子进程的结果[5]。经过我们的简单测试，使用共享内存可以大幅减少利用多核 CPU 时的内存占用。我们实验环境中的 CPU 有 12 个逻辑核，因此我们将需要处理的噪点坐标均分为 12 份，分给 12 个不同的子进程操作。

列表6实现了主要的进程调度算法，列表7是每个进行调用的工作函数。

```python
def restore_by_mean_multi_core(noise_img, size=1):
    noise_img_shared = Array("d", noise_img.ravel().tolist(), lock=False)
    # copy the original image (different memory location)
    res_img_shared = Array("d", np.copy(noise_img).ravel().tolist(), lock=False)
    # obtain noise_mask
    noise_mask_shared = Array("d", get_noise_mask(noise_img).ravel().tolist(),
    ↪ lock=False)
    # obtain shape of image
    rows, cols, channels = noise_img.shape
    # obtain noise points, as np.array "white"
    whites = np.argwhere(noise_img == 0.)
    # partition the whites list to 12 (number of logical cores of my CPU)
```

[5]由于我们的进程不会对其他进程涉及到的像素进行写操作，我们不需要锁来管理进程

```python
    parts = np.array_split(whites, 12)
    # contains all started jobs for future manipulation
    jobs = []
    # use a progress bar to indicate progress
    # this takes like, forever, why?
    with ProgressBar(max_value=len(parts)) as bar_start:
        for i, partial_whites in enumerate(parts):
            # we update the bar before spawning the subprocess to time it more accurately
            # and this makes the user see feedback faster, which makes them happy. At least it makes me happy
            bar_start.update(i)
            job = Process(
                target=wrapper_mean,
                args=(rows, cols, channels, size, noise_img_shared, noise_mask_shared, res_img_shared, partial_whites))
            jobs.append(job)
            job.start()
    # this takes like, forever
    with ProgressBar(max_value=len(jobs)) as bar_join:
        for i, job in enumerate(jobs):
            # same logic as above
            # if this is under job.join(), the first job takes 10s to finish,
            # the user waits 10s without seeing any feedback on screen
            bar_join.update(i)
            job.join()
    # notice that "res_img_shared" was originally a shared Array,
    # which takes some procedure to be transformed back to np.array
    return np.array(res_img_shared).reshape((rows, cols, channels))
```

Listing 6: 多核均值替换进程调度

```python
def wrapper_mean(rows, cols, channels, size, noise_img, noise_mask, res_img, partial_whites):
    # get objects we need from the shared memory, since client code requires them to have a certain shape
    noise_img = np.array(noise_img).reshape((rows, cols, channels))
    noise_mask = np.array(noise_mask).reshape((rows, cols, channels))

    # manually compute the corresponding indices, faster that iterating with np.unravel_index
```

```
    indices = partial_whites[:, 0] * cols * channels + partial_whites[:, 1] * channels + partial_whites[:, 2]

    # similar to what we do in a normal "restore by mean" function
    for i, (row, col, channel) in enumerate(partial_whites):
        # here the previously computed indices are used and we call function mean directly
        # notice that res_img is unchanged shared memory variable, which is mutable and affects the real memory
        # since different wrapper are to take care of different pixels, no lock is needed
        res_img[indices[i]] = mean(row, col, channel, rows, cols, size, noise_img, noise_mask)
```

Listing 7: 多核均值工作函数

4.2.2 线性回归预测

线性回归预测算法的整体逻辑于上述均值替换算法基本相同，在此我们列举出相关实现。

核心 线性回归预测降噪的核心算法如列表8所示。

```
def linear_regression(row, col, channel, rows, cols, size, noise_img, use_quadratic=False):
    while True:
        # considering the boundary, and transfer the square horizontally and vertically
        row_beg, row_end, col_beg, col_end = in_range_two(row, col, rows, cols, size)
        # get the "noisy" local image, flattened for better vectorized operations
        noise_img_local = noise_img[row_beg:row_end, col_beg:col_end, channel].ravel()
        # get valid positions
        x_train = np.argwhere(noise_img_local != 0.)
        if len(x_train) == 0:
            size *= 2
            continue
        y_train = noise_img_local[x_train]
        if use_quadratic:
            # quadratic linear regression
            quadratic = PolynomialFeatures(degree=3)
            x_train_quadratic = quadratic.fit_transform(x_train)
```

```
            regress_quadratic = LinearRegression()
            regress_quadratic.fit(x_train_quadratic, y_train)
            # predict
            test = quadratic.transform([[2 * size * size + size]])
            return regress_quadratic.predict(test)
        else:
            test = [[2 * size * size + size]]
            lr = Ridge().fit(x_train, y_train)
            # lr = ElasticNet().fit(x_train, y_train) # not converging
            # lr = Lasso().fit(x_train, y_train) # not converging
            # lr = LinearRegression().fit(x_train, y_train)
            # lr = RidgeCV().fit(x_train, y_train)
            # lr = Perceptron().fit(x_train, y_train)
            lr.predict(test)
```

Listing 8: 线性回归

4.2.3 Haar 小波降噪

我们按照实验原理中介绍的方式实现了 Haar 小波变换下的普适性降噪算法。

我们首先实现了简单的补边操作算法，如列表9所示。

```
def padding(img):
    # make copies in case nothing is changed
    img_v = np.copy(img)
    img_h = np.copy(img)
    if img.shape[0] % 2:
        # pad a horizontal line
        img_v = np.ndarray(shape=(img.shape[0] + 1, img.shape[1], img.shape[2]))
        img_v[:-1, :, :] = img
        img_v[-1, :, :] = img[-1, :, :]
    if img_v.shape[1] % 2:
        # pad a vertical line
        img_h = np.ndarray(shape=(img_v.shape[0], img_v.shape[1] + 1, img_v.shape[2]))
        img_h[:, :-1, :] = img_v
        img_h[:, -1, :] = img_v[:, -1, :]
    return img_h
```

Listing 9: 补边操作

接着我们实现了 Haar 小波变换与其逆变换，如列表10所示。

```python
def haar_encode(img):
    # pad the image for shape consistency
    img = padding(img)
    # compute haar info on axis=1
    img_v = np.zeros_like(img, dtype="double")
    img_v[:, :img.shape[1] // 2, :] = (img[:, ::2, :] + img[:, 1::2, :]) / 2
    img_v[:, img.shape[1] // 2:, :] = (img[:, ::2, :] - img[:, 1::2, :]) / 2
    # compute haar info on axis=0
    img_h = np.zeros_like(img, dtype="double")
    img_h[:img.shape[0] // 2, :, :] = (img_v[::2, :, :] + img_v[1::2, :, :]) / 2
    img_h[img.shape[0] // 2:, :, :] = (img_v[::2, :, :] - img_v[1::2, :, :]) / 2
    # the transformed image is returned
    return img_h

def haar_decode(img, padding_size=(0, 0)):
    # reverse haar info on axis=0
    img_v = np.zeros_like(img, dtype="double")
    img_v[::2, :, :] = img[:img.shape[0] // 2, :, :] + img[img.shape[0] // 2:, :, :]
    img_v[1::2, :, :] = img[:img.shape[0] // 2, :, :] - img[img.shape[0] // 2:, :, :]
    # reverse haar info on axis=1
    img_h = np.zeros_like(img, dtype="double")
    img_h[:, ::2, :] = img_v[:, :img.shape[1] // 2, :] + img_v[:, img.shape[1] // 2:, :]
    img_h[:, 1::2, :] = img_v[:, :img.shape[1] // 2, :] - img_v[:, img.shape[1] // 2:, :]
    # restore height and width according to padding information
    # print(img_h.shape)
    # print(padding_size, padding_size.dtype)
    if padding_size[0] != 0:
        img_h = img_h[:-padding_size[0], :, :]
    if padding_size[1] != 0:
        img_h = img_h[:, :-padding_size[1], :]
    # print(img_h.shape)
    # return the restored image
    return img_h
```

Listing 10: Haar 小波变换

最后我们实现了小波变换的降噪过程，如列表11所示。

```python
def haar_denoise(noise_img, threshold=0.1):
    # remember pading size
```

```python
    padding_size = (np.array(noise_img.shape) % 2)[:-1]
    # print(padding_size)
    # do transform
    res_img = haar_encode(noise_img)

    # throw away small value
    shape = res_img.shape
    # print(shape)
    res_img = np.ravel(res_img)
    abs_res_img = abs(res_img)
    nearly_white = np.argwhere(abs_res_img < threshold)
    res_img[nearly_white] = 0
    res_img = res_img.reshape(shape)
    # transform back
    res_img = haar_decode(res_img, padding_size)

    # return the denoised img
    return np.clip(res_img, 0., 1.)
```

Listing 11: Haar 小波变换普适降噪

4.2.4 对高斯噪声的处理

我们调用了 OpenCV 的接口以处理图片上的高斯噪声，如列表12所示。

```python
def mean_global_restore(img, kernel_size=(3, 3)):
    # we've defined three ways to reduce the noise on image, calling OpenCV library
    # you can uncomment the lines to remove gaussian noise
    blur = cv2.medianBlur((img * 255 if img.dtype == np.double else 1).astype(np.uint8),
                          kernel_size[0]) / 255 if img.dtype == np.double else 1
    # blur = cv2.GaussianBlur(img, (5, 5), 0)
    # blur = cv2.blur(img, kernel_size)
    return blur
```

Listing 12: 消除高斯噪声

5 实验结果

5.1 普通测试

5.1.1 测试结果

我们首先验证了高斯噪声的添加与消除效果，我们并以同一图片对比了不同降噪方式的效果。如图4所示。

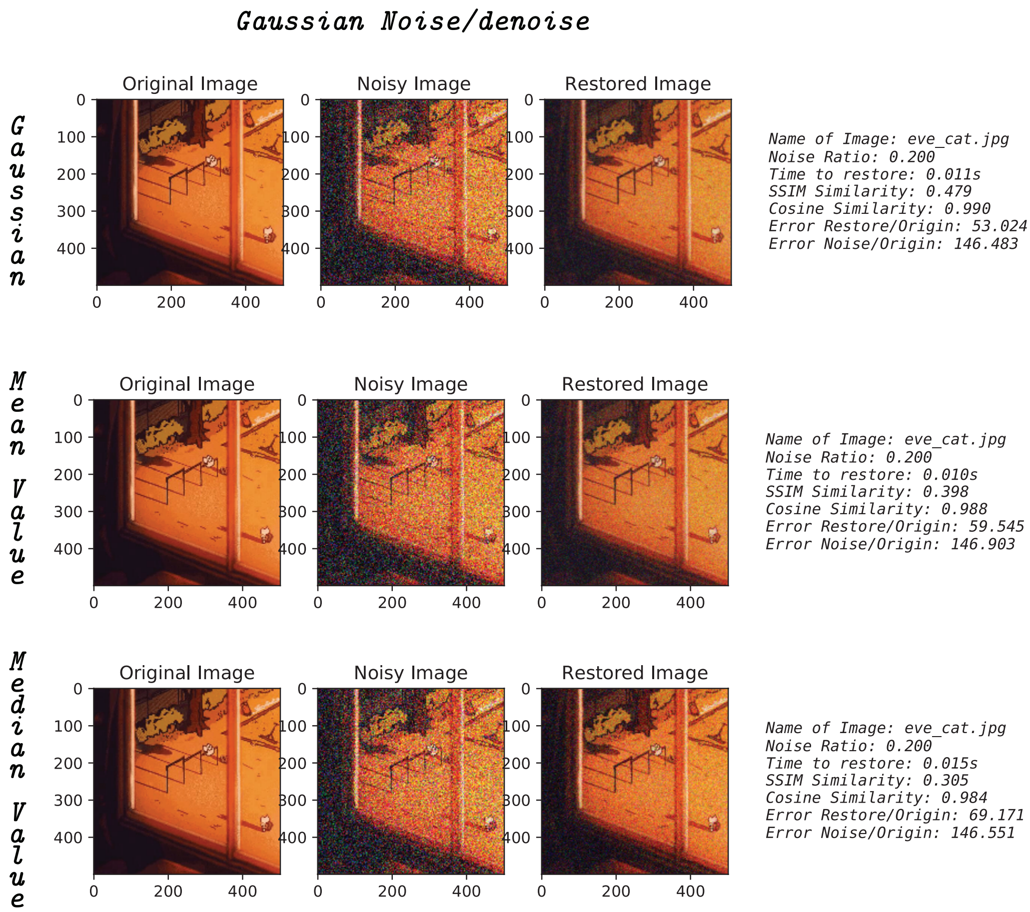
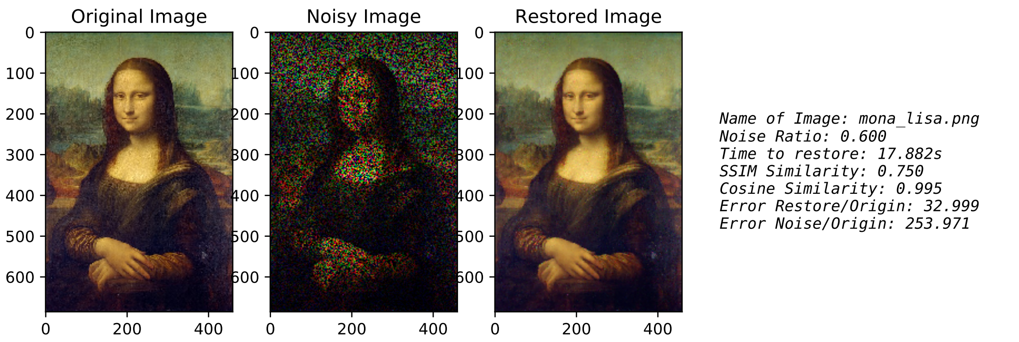
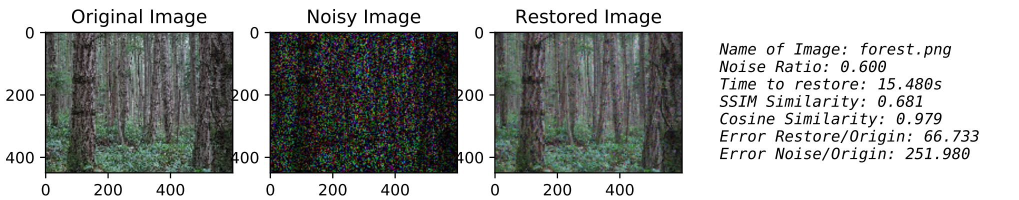


图 4: 修复高斯噪声

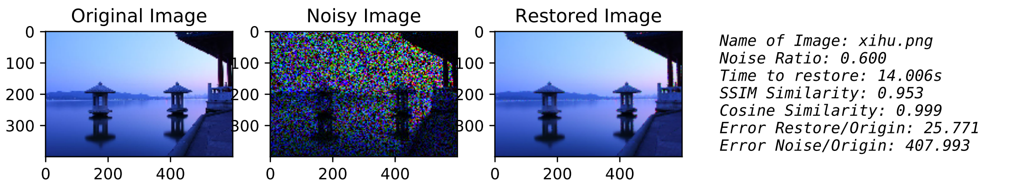
我们选取了一些图片进行算法的验证性测试（使用均值替换算法），测试结果如图5所示。



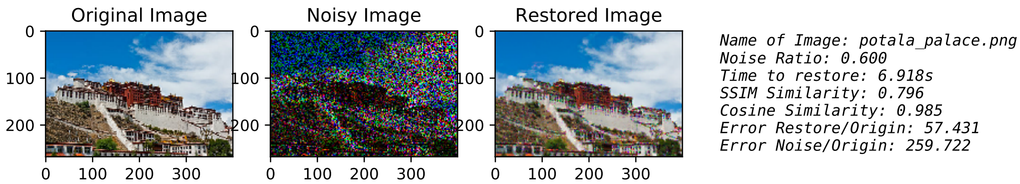
(a) Mona Lisa



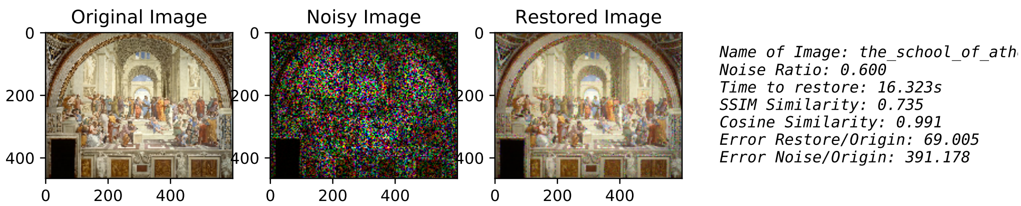
(b) Forest



(c) West Lake



(d) Potala Palace



(e) The School of Athens

图 5: 测试结果

5.1.2 结果分析

正如我们所预料的，利用了 numpy 向量化运算的加速结果较为明显，均值替换算法没有消耗过多时间。由于我们在降噪过程中充分利用了噪声的本来特点，最终降噪结果优异，能够大幅度提高噪声图片的辨识度。实验过程中我们容易发现采用线性回归预测的方法得到的结果（误差，相似度等）与使用均值替换算法几乎无异，在此就不一一列出。值得注意的是，线性回归预测算法会花费数十倍甚至更多的时间（这将在下一部分得到验证）。

5.2 对比测试

5.2.1 测试结果

我们选取了一张样例图片进行对比测试，测试结果如图6所示。

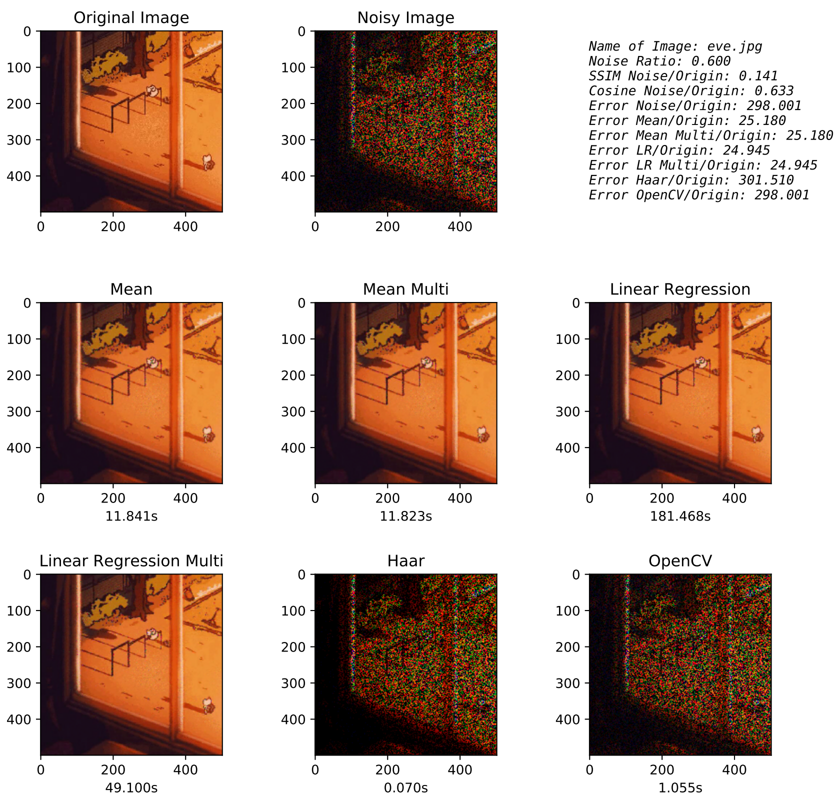


图 6: 对比测试

5.2.2 结果分析

恢复效果 正如我们所预期的，利用了噪声特点的均值替换算法，线性回归预测的图像恢复效果远优于传统普适算法如 *Haar* 小波降噪等，这一结论可以通过观察 `Error(L2 范数)` 得出。但不能忽略的是，现实世界中的图片噪声和本次实验中的相差甚远，这种情况下某些普适算法效果拔群[6]。事实上我们还用一些专业图片处理软件对此类噪声进行了测试，如 *PhotoShop*，效果远不如利用了噪声特点的算法。但在处理日常相机感光器的噪声时[7]，那些算法表现优异。

单核时间消耗 不可避免的，均值替换算法，线性回归预测耗时较普适算法更高，但值得注意的一点是，线性回归预测算法的耗时也远大于均值替换算法，这是由于在线性回归预测中每一个噪声点都需要伴随拟合一个线性回归模型，根据模型的复杂度，收敛所需的迭代次数也不近相同，但这一过程所耗时间基本都远大于普通的均值计算[8]。在样例图片中，单核 CPU 下均值算法消耗了约 12s，而线性回归预测模型消耗了 3 分钟，两者的恢复效果却相差不大。

多核优化效果 在多核版本与单核版本的对比中，我们也得到了符合预期的结果，即：时间上有所提升且恢复结果相同。值得注意的是，对均值替换算法的多核优化并没有起到太好的效果，这在程序实际运行时就能观察到：生成线程所需的时间几乎同一条线程执行完毕所需时间相差无几，也就是 CPU 整体利用率难以达到最高。

对比之下，线性回归预测算法的特性很适合多核优化。我们可以在图6中观察到，多核版本的线性回归算法只消耗了 50s 左右，比普通版本快了两分钟多。

[6]基于噪声特点的算法根本无法有效运行，因为噪声点不一定为零值点

[7]相机感光器的构造导致噪声必然会在按下快门的一刻产生，并随着相机感光度增加而上升

[8]尤其是通过向量化运算加速后的版本

附录 A 辅助代码与全部算法实现

```python
SAMPLE_DIR = "samples"
OUTPUT_DIR = "output"
RESTORE_NAME_EXTENSION = "_res.png"
NOISE_NAME_EXTENSION = "_noi.png"
ORIGIN_NAME_EXTENSION = "_ori.png"
LOG_NAME_EXTENSION = "_output.txt"
PLOT_NAME_EXTENSION = "_plot.eps"


def test_img(img_path):
    # 加载图片的路径和名称
    # img_path = 'eve.jpg'
    # img_path = 'A.png'

    # 读取原始图片
    img = read_image("/".join((SAMPLE_DIR, img_path)))

    # 展示原始图片
    # plot_image(image=img, image_title="original image")

    # 生成受损图片
    # 图像数据归一化
    nor_img = normalization(img)

    # 噪声比率
    noise_ratio = 0.6

    # 生成受损图片
    noise_img = noise_mask_image(nor_img, noise_ratio)

    # 展示受损图片
    # plot_image(image=noise_img, image_title="the noise_ratio = %s of original image"
    ↪ % noise_ratio)

    start_time = perf_counter()
    # 恢复图片
    res_img = restore_by_mean(noise_img, size=2)
    # res_img = restore_by_mean_multi_core(noise_img, size=2)
    # res_img = restore_by_linear_regression(noise_img, size=2)
    # res_img = restore_by_linear_regression_multi_core(noise_img, size=2)
```

```
    # res_img = haar_denoise(noise_img)
    # res_img =
    ↪  cv2.fastNlMeansDenoisingColored((noise_img*255).astype("uint8")).astype("double")
    ↪  / 255
    end_time = perf_counter()

    # 计算恢复图片与原始图片的误差
    ori_img_path = "/".join((OUTPUT_DIR, img_path+ORIGIN_NAME_EXTENSION))
    noi_img_path = "/".join((OUTPUT_DIR, img_path+NOISE_NAME_EXTENSION))
    res_img_path = "/".join((OUTPUT_DIR, img_path+RESTORE_NAME_EXTENSION))
    res_log_path = "/".join((OUTPUT_DIR, img_path+LOG_NAME_EXTENSION))
    os.makedirs(os.path.dirname(res_img_path), exist_ok=True)
    with open(res_log_path, "w") as f:
        log_info = "\n".join((
            "Name of Image: "+img_path,
            "Noise Ratio: {:.3f}".format(noise_ratio),
            "Time to restore: {:.3f}s".format(end_time-start_time),
            "SSIM Similarity: {:.3f}".format(calc_ssim(res_img, nor_img)),
            "Cosine Similarity: {:.3f}".format(calc_csim(res_img, nor_img)),
            "Error Restore/Origin: {:.3f}".format(compute_error(res_img, nor_img)),
            "Error Noise/Origin: {:.3f}".format(compute_error(noise_img, nor_img)),
        ))
        f.write(log_info)

    # 展示恢复图片
    # plot_image(image=res_img, image_title="restore image")

    # 保存恢复图片
    save_image(res_img_path, res_img)
    save_image(noi_img_path, noise_img)
    save_image(ori_img_path, nor_img)
    plot_img(nor_img, noise_img, res_img, log_info, img_path)


def test_all(img_path):
    ori = read_image(img_path)
    nor_img = normalization(ori)
    noise_ratio = 0.6
    noise_img = noise_mask_image(nor_img, noise_ratio)
    times = [perf_counter()]
    res_img_mean = restore_by_mean(noise_img, size=2)
    # res_img_mean = haar_denoise(noise_img)
```

```
    times += [perf_counter()]
    res_img_mean_multi = restore_by_mean_multi_core(noise_img, size=2)
    # res_img_mean_multi = haar_denoise(noise_img)
    times += [perf_counter()]
    res_img_lr = restore_by_linear_regression(noise_img, size=2)
    # res_img_lr = haar_denoise(noise_img)
    times += [perf_counter()]
    res_img_lr_multi = restore_by_linear_regression_multi_core(noise_img, size=2)
    # res_img_lr_multi = haar_denoise(noise_img)
    times += [perf_counter()]
    res_img_haar = haar_denoise(noise_img)
    times += [perf_counter()]
    res_img_cv =
    ↪   cv2.fastNlMeansDenoisingColored((noise_img*255).astype("uint8")).astype("double")
    ↪   / 255
    times += [perf_counter()]
    log_info = "\n".join((
        "Name of Image: "+img_path,
        "Noise Ratio: {:.3f}".format(noise_ratio),
        "SSIM Noise/Origin: {:.3f}".format(calc_ssim(noise_img, nor_img)),
        "Cosine Noise/Origin: {:.3f}".format(calc_csim(noise_img, nor_img)),
        "Error Noise/Origin: {:.3f}".format(compute_error(noise_img, nor_img)),
        "Error Mean/Origin: {:.3f}".format(compute_error(res_img_mean, nor_img)),
        "Error Mean Multi/Origin: {:.3f}".format(compute_error(res_img_mean_multi,
        ↪   nor_img)),
        "Error LR/Origin: {:.3f}".format(compute_error(res_img_lr, nor_img)),
        "Error LR Multi/Origin: {:.3f}".format(compute_error(res_img_lr_multi,
        ↪   nor_img)),
        "Error Haar/Origin: {:.3f}".format(compute_error(res_img_haar, nor_img)),
        "Error OpenCV/Origin: {:.3f}".format(compute_error(res_img_cv, nor_img)),
    ))
    times = np.array(times)
    times = times[1::] - times[0:-1]
    hspace = 0.5
    wspace = 0.5
    width = 10
    height = ori.shape[0]/ori.shape[1] * width * hspace/wspace
    fig = plt.figure(figsize=(width, height))
    fig.subplots_adjust(hspace=hspace, wspace=wspace)
    axi_ori = fig.add_subplot(331)
    axi_noi = fig.add_subplot(332)
    axi_log = fig.add_subplot(333)
```

```python
    axi_res_mean = fig.add_subplot(334)
    axi_res_mean_multi = fig.add_subplot(335)
    axi_res_lr = fig.add_subplot(336)
    axi_res_lr_multi = fig.add_subplot(337)
    axi_res_haar = fig.add_subplot(338)
    axi_res_cv = fig.add_subplot(339)

    axi_ori.set_title("Original Image")
    axi_ori.imshow(ori)
    axi_noi.set_title("Noisy Image")
    axi_noi.imshow(noise_img)

    axi_res_mean.set_title("Mean")
    axi_res_mean.imshow(res_img_mean)
    axi_res_mean.set_xlabel("{:.3f}s".format(times[0]))
    axi_res_mean_multi.set_title("Mean Multi")
    axi_res_mean_multi.imshow(res_img_mean_multi)
    axi_res_mean_multi.set_xlabel("{:.3f}s".format(times[1]))
    axi_res_lr.set_title("Linear Regression")
    axi_res_lr.imshow(res_img_lr)
    axi_res_lr.set_xlabel("{:.3f}s".format(times[2]))
    axi_res_lr_multi.set_title("Linear Regression Multi")
    axi_res_lr_multi.imshow(res_img_lr_multi)
    axi_res_lr_multi.set_xlabel("{:.3f}s".format(times[3]))
    axi_res_haar.set_title("Haar")
    axi_res_haar.imshow(res_img_haar)
    axi_res_haar.set_xlabel("{:.3f}s".format(times[4]))
    axi_res_cv.set_title("OpenCV")
    axi_res_cv.imshow(res_img_cv)
    axi_res_cv.set_xlabel("{:.3f}s".format(times[5]))

    axi_log.set_xlim(0, ori.shape[1])
    axi_log.set_ylim(0, ori.shape[0])
    axi_log.text(0, ori.shape[0]//2, log_info, family="monospace", style="italic",
    ↪  ha="left", va="center")
    axi_log.axis("off")
    fig.savefig("/".join((OUTPUT_DIR, img_path+PLOT_NAME_EXTENSION)))


def main():
    img_list = os.listdir(SAMPLE_DIR)
    jobs = []
```

```
    with ProgressBar(max_value=len(img_list)) as bar_start:
        for i, img_path in enumerate(img_list):
            bar_start.update(i)
            job = Process(target=test_img, args=(img_path,))
            job.start()
            jobs.append(job)
    with ProgressBar(max_value=len(jobs)) as bar_join:
        for i, job in enumerate(jobs):
            bar_join.update(i)
            job.join()


def plot_img(ori, noi, res, log, img_path):
    width = 10
    height = ori.shape[0]/ori.shape[1]/4 * width
    fig = plt.figure(figsize=(width, height))
    axi_ori = fig.add_subplot(141)
    axi_noi = fig.add_subplot(142)
    axi_res = fig.add_subplot(143)
    axi_log = fig.add_subplot(144)
    axi_ori.set_title("Original Image")
    axi_ori.imshow(ori)
    axi_noi.set_title("Noisy Image")
    axi_noi.imshow(noi)
    axi_res.set_title("Restored Image")
    axi_res.imshow(res)
    axi_log.set_xlim(0, ori.shape[1])
    axi_log.set_ylim(0, ori.shape[0])
    axi_log.text(0, ori.shape[0]//2, log, family="monospace", style="italic",
    ↪  ha="left", va="center")
    axi_log.axis("off")
    fig.savefig("/".join((OUTPUT_DIR, img_path+PLOT_NAME_EXTENSION)))


def plot_dir(dirname):
    files = os.listdir(dirname)
    files.sort()
    txts = [file_name for file_name in files if
    ↪  file_name.endswith(LOG_NAME_EXTENSION)]
    ress = [file_name for file_name in files if
    ↪  file_name.endswith(RESTORE_NAME_EXTENSION)]
```

```
    oris = [file_name for file_name in files if file_name.endswith(ORIGIN_NAME_EXTENSION)]
    nois = [file_name for file_name in files if file_name.endswith(NOISE_NAME_EXTENSION)]
    lst = np.transpose(np.array((txts, oris, nois, ress)))

    for txt, ori, noi, res in lst:
        with open(dirname+txt, "r") as f:
            txt = f.read()
        img_path = ori
        ori = read_image(ori)
        noi = read_image(noi)
        res = read_image(res)
        plot_img(ori, noi, res, txt, img_path)
```

Listing 13: 辅助代码：测试，作图等

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge, Lasso, RidgeCV, Perceptron, ElasticNet
from sklearn.preprocessing import PolynomialFeatures
from multiprocessing import Process, Array
from progressbar import ProgressBar


def restore_image(img, size=2):
    return restore_by_mean(img, size=size)
    # return restore_by_mean_multi_core(img)
    # return restore_by_linear_regression(img)
    # return restore_by_linear_regression_multi_core(img)


def noise_mask_image(img, noise_ratio):
    """
    generates the "noisy" image according to the specific problem

    :param img: img np.ndarray
    :param noise_ratio: more noise? 0.4/0.6/0.8
    :return: noise_img is the image with noise, 0-1 np.array, data type=np.double shape=(height,width,channel) channel=RGB
```

```python
    """
    # copy the original image (different memory location)
    noise_img = np.copy(img)
    # initialization
    noise_mask = np.ones_like(noise_img, dtype='double')
    # mask image according to the ratio
    noise_mask[:, :round(noise_img.shape[1] * noise_ratio), :] = 0.
    # shuffle every row in every channel
    for channel in range(noise_img.shape[2]):
        for row in range(noise_img.shape[0]):
            np.random.shuffle(noise_mask[row, :, channel])
    noise_img = noise_img * noise_mask

    return noise_img


def get_noise_mask(noise_img):
    """
    get the noise mask of noise_img, usually a np.array

    :param noise_img: image with noise
    :return: noise mask, as double, size of noise_img
    """
    # we consider every 0 point in the image as noise point
    return np.array(noise_img != 0., dtype='double')


def in_range_one(row, rows, size):
    """
    generate range based on current position and size, taking care of edge

    :param row: current row/column number
    :param rows: total number of rows/columns
    :param size: radius/size of our consideration
    :return row_beg. row_end: normally row-size, row+size, however edge is taken care of
    """
    row_beg = row - size if row - size >= 0 else 0
    row_end = row + size if row + size < rows else rows - 1
    return row_beg, row_end


```

```python
def in_range_two(row, col, rows, cols, size):
    """
    calls in_range_one twice to get 2D range for current position

    :param row: current row number
    :param col: current column number
    :param rows: total number of rows
    :param cols: total number of columns
    :param size: radius/size of our consideration
    :return: flattened 2D range
    """
    return np.array((in_range_one(row, rows, size), in_range_one(col, cols, size))).flatten()


def restore_by_mean(noise_img, size=2):
    """
    restore image by calculating RGB means of surrounding pixels.

    :param noise_img: a "noisy" image
    :param size: radius of mean values computation, defaults to 4, we compute mean from pixels in a 2*size*size square
    :return: res_img is the image restored, 0-1 np.array, data type=np.double shape=(height,width,channel) channel=RGB
    """
    # copy the original image (different memory location)
    res_img = np.copy(noise_img)
    # obtain noise_mask
    noise_mask = get_noise_mask(noise_img)
    # obtain shape of image
    rows, cols, channels = noise_img.shape
    # obtain noise points, as np.array "white"
    whites = np.argwhere(noise_mask == 0.)
    # use a progress bar to indicate progress
    with ProgressBar(max_value=len(whites)) as bar:
        for i, (row, col, channel) in enumerate(whites):
            res_img[row, col, channel] = mean(row, col, channel, rows, cols, size, noise_img, noise_mask)
            bar.update(i)
    return res_img
```

```python
def restore_by_mean_multi_core(noise_img, size=2):
    """
    restore image by mean, however, we try to utilize multi-core CPU here

    :param noise_img: same as restore_by_mean
    :param size: same as restore_by_mean
    :return: same as restore_by_mean
    """
    noise_img_shared = Array("d", noise_img.ravel().tolist(), lock=False)
    # copy the original image (different memory location)
    res_img_shared = Array("d", np.copy(noise_img).ravel().tolist(), lock=False)
    # obtain noise_mask
    noise_mask_shared = Array("d", get_noise_mask(noise_img).ravel().tolist(),
    ↪  lock=False)
    # obtain shape of image
    rows, cols, channels = noise_img.shape
    # obtain noise points, as np.array "white"
    whites = np.argwhere(noise_img == 0.)
    # partition the whites list to 12 (number of logical cores of my CPU)
    parts = np.array_split(whites, 12)
    # contains all started jobs for future manipulation
    jobs = []
    # use a progress bar to indicate progress
    # this takes like, forever, why?
    with ProgressBar(max_value=len(parts)) as bar_start:
        for i, partial_whites in enumerate(parts):
            # we update the bar before spawning the subprocess to time it more
            ↪  accurately
            # and this makes the user see feedback faster, which makes them happy. At
            ↪  least it makes me happy
            bar_start.update(i)
            job = Process(
                target=wrapper_mean,
                args=(rows, cols, channels, size, noise_img_shared, noise_mask_shared,
                ↪  res_img_shared, partial_whites))
            jobs.append(job)
            job.start()
    # this takes like, forever
    with ProgressBar(max_value=len(jobs)) as bar_join:
        for i, job in enumerate(jobs):
            # same logic as above
            # if this is under job.join(), the first job takes 10s to finish,
```

```python
            # the user waits 10s without seeing any feedback on screen
            bar_join.update(i)
            job.join()

    # notice that "res_img_shared" was originally a shared Array,
    # which takes some procedure to be transformed back to np.array
    return np.array(res_img_shared).reshape((rows, cols, channels))


def wrapper_mean(rows, cols, channels, size, noise_img, noise_mask, res_img, partial_whites):
    """
    A wrapper around the computation of mean value so that we can utilize the ability of multi-core CPU

    :param rows: img.shape[0]
    :param cols: img.shape[1]
    :param channels: img.shape[2]
    :param size: radius of mean values square
    :param noise_img: "noisy" img, shared memory between processes
    :param noise_mask: extracted noise mask, shared memory
    :param res_img: the result img to be modified, shared memory
    :param partial_whites: the white points that this wrapper should take care of
    :return: nothing, the function modifies res_img directly
    """

    # get objects we need from the shared memory, since client code requires them to have a certain shape
    noise_img = np.array(noise_img).reshape((rows, cols, channels))
    noise_mask = np.array(noise_mask).reshape((rows, cols, channels))

    # manually compute the corresponding indices, faster that iterating with np.unravel_index
    indices = partial_whites[:, 0] * cols * channels + partial_whites[:, 1] * channels + partial_whites[:, 2]

    # similar to what we do in a normal "restore by mean" function
    for i, (row, col, channel) in enumerate(partial_whites):
        # here the previously computed indices are used and we call function mean directly
        # notice that res_img is unchanged shared memory variable, which is mutable and affects the real memory
```

```python
        # since different wrapper are to take care of different pixels, no lock is needed
        res_img[indices[i]] = mean(row, col, channel, rows, cols, size, noise_img, noise_mask)


def mean(row, col, channel, rows, cols, size, noise_img, noise_mask):
    """
    separate actual atomic computation from pre-processing, pave the way for multi-threading

    :param row: current row
    :param col: current column
    :param rows: total number of rows
    :param cols: total number of columns
    :param size: radius, 2*size*size image
    :param channel: current channel
    :param noise_img: "noisy" image
    :param noise_mask: binary(as double) noise mask
    :return: the mean value for [row, col, channel]
    """

    # we introduce a while(1) loop so that we can expand our search windows until one with valid pixel(s) is found
    while True:
        # considering the boundary, and transfer the square horizontally and vertically
        row_beg, row_end, col_beg, col_end = in_range_two(row, col, rows, cols, size)

        # mean values is sum(all pixels)/sum(noise mask)
        # since white point won't affect total sum and sum of noise mask indicates number of valid pixels
        # of course we can get number of valid positions from noise_img directly
        # but in practice, computing this from sum of noise_mask proves to be much faster
        number = np.sum(noise_mask[row_beg:row_end, col_beg:col_end, channel])
        # we update size and continue loop before computing "total", which saves time
        if number == 0.:
            size *= 2
            continue
        total = np.sum(noise_img[row_beg:row_end, col_beg:col_end, channel])
        return total / number
```

```python


def restore_by_linear_regression(noise_img, size=2):
    """
    restore image by quadratic linear regression

    :param noise_img: same as restore_by_mean
    :param size: same as restore_by_mean
    :return: same as restore_by_mean
    """
    # copy the original image (different memory location)
    res_img = np.copy(noise_img)
    # obtain shape of image
    rows, cols, channels = noise_img.shape
    # obtain noise points, as np.array "white"
    whites = np.argwhere(noise_img == 0.)
    # use a progress bar to indicate progress
    with ProgressBar(max_value=len(whites)) as bar:
        for i, (row, col, channel) in enumerate(whites):
            bar.update(i)
            res_img[row, col, channel] = linear_regression(row, col, channel, rows,
            ↪ cols, size, noise_img)
    return np.clip(res_img, 0., 1.)


def linear_regression(row, col, channel, rows, cols, size, noise_img,
↪ use_quadratic=False):
    """
    separate actual atomic computation from pre-processing, pave the way for
↪   multi-threading

    :param row: current row
    :param col: current column
    :param rows: total number of rows
    :param cols: total number of columns
    :param size: radius, 2*size*size image
    :param channel: current channel
    :param noise_img: "noisy" image
    :param use_quadratic: whether to use quadratic linear regression (utilize CPU
↪   more)
    :return: the predicted value for [row, col, channel]
    """
```

```python
    while True:
        # considering the boundary, and transfer the square horizontally and vertically
        row_beg, row_end, col_beg, col_end = in_range_two(row, col, rows, cols, size)
        # get the "noisy" local image, flattened for better vectorized operations
        noise_img_local = noise_img[row_beg:row_end, col_beg:col_end, channel].ravel()
        # get valid positions
        x_train = np.argwhere(noise_img_local != 0.)
        if len(x_train) == 0:
            size *= 2
            continue
        y_train = noise_img_local[x_train]
        if use_quadratic:
            # quadratic linear regression
            quadratic = PolynomialFeatures(degree=3)
            x_train_quadratic = quadratic.fit_transform(x_train)
            regress_quadratic = LinearRegression()
            regress_quadratic.fit(x_train_quadratic, y_train)
            # predict
            test = quadratic.transform([[2 * size * size + size]])
            return regress_quadratic.predict(test)
        else:
            test = [[2 * size * size + size]]
            lr = Ridge().fit(x_train, y_train)
            # lr = ElasticNet().fit(x_train, y_train) # not converging
            # lr = Lasso().fit(x_train, y_train) # not converging
            # lr = LinearRegression().fit(x_train, y_train)
            # lr = RidgeCV().fit(x_train, y_train)
            # lr = Perceptron().fit(x_train, y_train)
            return lr.predict(test)


def restore_by_linear_regression_multi_core(noise_img, size=2):
    """
    restore image by quadratic linear regression, however, we try to utilize multi-core CPU here

    :param noise_img: same as restore_by_mean
    :param size: same as restore_by_mean
    :return: same as restore_by_mean
    """
    noise_img_shared = Array("d", noise_img.ravel().tolist(), lock=False)
```

```python
    # copy the original image (different memory location)
    res_img_shared = Array("d", np.copy(noise_img).ravel().tolist(), lock=False)
    # obtain shape of image
    rows, cols, channels = noise_img.shape
    # obtain noise points, as np.array "white"
    whites = np.argwhere(noise_img == 0.)
    # partition the whites list to 12 (number of logical cores of my CPU)
    parts = np.array_split(whites, 12)
    # contains all started jobs for future manipulation
    jobs = []
    # use a progress bar to indicate progress
    # this takes like, forever, why?
    with ProgressBar(max_value=len(parts)) as bar_start:
        for i, partial_whites in enumerate(parts):
            # we update the bar before spawning the subprocess to time it more accurately
            # and this makes the user see feedback faster, which makes them happy. At least it makes me happy
            bar_start.update(i)
            job = Process(
                target=wrapper_linear_regression,
                args=(rows, cols, channels, size, noise_img_shared, res_img_shared, partial_whites))
            jobs.append(job)
            job.start()
    # this takes like, forever
    with ProgressBar(max_value=len(jobs)) as bar_join:
        for i, job in enumerate(jobs):
            # same logic as above
            # if this is under job.join(), the first job takes 10s to finish,
            # the user waits 10s without seeing any feedback on screen
            bar_join.update(i)
            job.join()

    # notice that "res_img_shared" was originally a shared Array,
    # which takes some procedure to be transformed back to np.array
    return np.clip(np.array(res_img_shared).reshape((rows, cols, channels)), 0., 1.)


def wrapper_linear_regression(rows, cols, channels, size, noise_img, res_img, partial_whites):
    """
```

```
    A wrapper around the computation of linear regression function so that we can utilize the ability of multi-core CPU

    :param rows: img.shape[0]
    :param cols: img.shape[1]
    :param channels: img.shape[2]
    :param size: radius of mean values square
    :param noise_img: "noisy" img, shared memory between processes
    :param res_img: the result img to be modified, shared memory
    :param partial_whites: the white points that this wrapper should take care of
    :return: nothing, the function modifies res_img directly
    """
    noise_img = np.array(noise_img).reshape((rows, cols, channels))
    indices = partial_whites[:, 0] * cols * channels + partial_whites[:, 1] * channels + partial_whites[:, 2]
    for i, (row, col, channel) in enumerate(partial_whites):
        res_img[indices[i]] = linear_regression(row, col, channel, rows, cols, size, noise_img, use_quadratic=False)


def padding(img):
    """
    Pad the img so that it's divided by 2

    :param img: the img to be padded, as np.ndarray
    :return: the padded image
    """

    # make copies in case nothing is changed
    img_v = np.copy(img)
    img_h = np.copy(img)
    if img.shape[0] % 2:
        # pad a horizontal line
        img_v = np.ndarray(shape=(img.shape[0] + 1, img.shape[1], img.shape[2]))
        img_v[:-1, :, :] = img
        img_v[-1, :, :] = img[-1, :, :]
    if img_v.shape[1] % 2:
        # pad a vertical line
        img_h = np.ndarray(shape=(img_v.shape[0], img_v.shape[1] + 1, img_v.shape[2]))
        img_h[:, :-1, :] = img_v
        img_h[:, -1, :] = img_v[:, -1, :]
    return img_h
```

```python


def haar_encode(img):
    """
    compute the haar transform of a img, padding it first, so you may want to store
↪   the original shape

    :param img: the img to be transformed
    :return: the transformed img
    """

    # pad the image for shape consistency
    img = padding(img)
    # compute haar info on axis=1
    img_v = np.zeros_like(img, dtype="double")
    img_v[:, :img.shape[1] // 2, :] = (img[:, ::2, :] + img[:, 1::2, :]) / 2
    img_v[:, img.shape[1] // 2:, :] = (img[:, ::2, :] - img[:, 1::2, :]) / 2
    # compute haar info on axis=0
    img_h = np.zeros_like(img, dtype="double")
    img_h[:img.shape[0] // 2, :, :] = (img_v[::2, :, :] + img_v[1::2, :, :]) / 2
    img_h[img.shape[0] // 2:, :, :] = (img_v[::2, :, :] - img_v[1::2, :, :]) / 2
    # the transformed image is returned
    return img_h


def haar_decode(img, padding_size=(0, 0)):
    """
    reverse the change caused by haar_transform, with padding information provided

    :param img: the haar transformed image
    :param padding_size: padding add to height and width to be reversed
    :return: the "untransformed" image
    """

    # reverse haar info on axis=0
    img_v = np.zeros_like(img, dtype="double")
    img_v[::2, :, :] = img[:img.shape[0] // 2, :, :] + img[img.shape[0] // 2:, :, :]
    img_v[1::2, :, :] = img[:img.shape[0] // 2, :, :] - img[img.shape[0] // 2:, :, :]
    # reverse haar info on axis=1
    img_h = np.zeros_like(img, dtype="double")
    img_h[:, ::2, :] = img_v[:, :img.shape[1] // 2, :] + img_v[:, img.shape[1] // 2:,
    ↪  :]
```

```python
    img_h[:, 1::2, :] = img_v[:, :img.shape[1] // 2, :] - img_v[:, img.shape[1] // 2:,
    ↪  :]
    # restore height and width according to padding information
    # print(img_h.shape)
    # print(padding_size, padding_size.dtype)
    if padding_size[0] != 0:
        img_h = img_h[:-padding_size[0], :, :]
    if padding_size[1] != 0:
        img_h = img_h[:, :-padding_size[1], :]
    # print(img_h.shape)
    # return the restored image
    return img_h


def haar_denoise(noise_img, threshold=0.1):
    """
    calls haar_transform and haar_transform_back to remove noise in an img, deleting
↪   pixels under a certain threshold

    :param noise_img: "noisy" img
    :param threshold: we'd assume img to be of "double" and threshold should be in [0,
↪   1]
    :return:
    """

    # remember pading size
    padding_size = (np.array(noise_img.shape) % 2)[:-1]
    # print(padding_size)
    # do transform
    res_img = haar_encode(noise_img)

    # throw away small value
    shape = res_img.shape
    # print(shape)
    res_img = np.ravel(res_img)
    abs_res_img = abs(res_img)
    nearly_white = np.argwhere(abs_res_img < threshold)
    res_img[nearly_white] = 0
    res_img = res_img.reshape(shape)
    # transform back
    res_img = haar_decode(res_img, padding_size)

```

```python
    # return the denoised img
    return np.clip(res_img, 0., 1.)
```

Listing 14: 全部算法具体实现

附录 B 插图，表格与列表

插图

1 噪声遮罩 ($noise\_ratio = 0.6$) 4
2 噪声图像 ($noise\_ratio = 0.6$) 5
3 噪声点周围环境 ($radius = 2$) 6
4 修复高斯噪声 16
5 测试结果 17
6 对比测试 18

表格

List of Listings

1 根据比例添加随机噪声 8
2 添加高斯噪声（noise_variance = max * 0.2） 8
3 公用函数：对边角的统一处理 9
4 均值计算 9
5 单核均值替换降噪 10
6 多核均值替换进程调度 11
7 多核均值工作函数 12
8 线性回归 13
9 补边操作 13
10 Haar 小波变换 14
11 Haar 小波变换普适降噪 15
12 消除高斯噪声 15
13 辅助代码：测试，作图等 25
14 全部算法具体实现 37