《人工智能与科学计算》

实验报告

题目：	图像旋转
姓名：	胡森康
学院：	信息与电子学院
邮箱：	huhbu@outlook.com
日期：	2021 年 6 月 13 日

目 录

1 实验目的

1. 采用前向映射和线性插值实现图像旋转，串行
2. 在 1 的基础上实现 OpenMP 并行
3. 注意各种优化措施

2 实验原理

2.1 图像变换与映射

我们在进行图像处理时常常需要对图像进行变换。比如对图像进行缩放，旋转，平移等。图像变换的本质是将像素点的坐标通过某一种函数关系，映射到另外的位置。假设变换前图像为 $I(x,y)$，变换后图像为 $I'(x',y')$，则变换前后的图像之间存在下列关系:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} f(x,y) \\ g(x,y) \end{bmatrix} \tag{2-1}$$

$$I(x,y) = I'(x',y') = I\left(f(x,y), g(x,y)\right) \tag{2-2}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f^{-1}(x',y') \\ g^{-1}(x',y') \end{bmatrix} \tag{2-3}$$

$$I'(x',y') = I(x,y) = I\left(f^{-1}(x',y'), g^{-1}(x',y')\right) \tag{2-4}$$

式 (2-1) 和式 (2-2) 已知原图像到目标图像的坐标变换 $(f(x,y), g(x,y))$，因此可以知道原图像的一点在变换后在目标图像的位置，称为前向映射。

相反，式 (2-3) 和 (2-4) 中已知目标图像的一点 (x',y') 在变换前在原图像上的位置 $(f^{-1}(x,y), g^{-1}(x,y))$，称为后向映射。

一般认为图像应该绕着中心点旋转，而且图像原点在左上角，计算时首先需要将左上角的远点移动至图像中心，并且将 y 轴翻转。设图像宽为 W，高为 H，有

$$[x_1 \ y_1 \ 1] = [x_0 \ y_0 \ 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -0.5W & 0.5H & 1 \end{bmatrix} \tag{2-5}$$

由坐标映射关系知，点 (x_1, y_1) 顺时针旋转 θ 后的坐标 (x_2, y_2) 为：

$$[x_2 \ y_2 \ 1] = [x_1 \ y_1 \ 1] \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2-6}$$

若旋转后的图像宽为 W'，高为 H'，那么从笛卡尔坐标原点变换回左上角的公式为

$$[x_3\ y_3\ 1] = [x_2\ y_2\ 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0.5W' & 0.5H' & 1 \end{bmatrix} \tag{2-7}$$

则点 (x_0, y_0) 与顺时针旋转 θ 角度后的坐标 (x_3, y_3) 之间的关系为：

$$[x_3\ y_3\ 1] = [x_0\ y_0\ 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -0.5W & 0.5H & 1 \end{bmatrix}\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0.5W' & 0.5H' & 1 \end{bmatrix} \tag{2-8}$$

$$[x_0\ y_0\ 1] = [x_3\ y_3\ 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -0.5'W & 0.5H' & 1 \end{bmatrix}\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0.5W & 0.5H & 1 \end{bmatrix} \tag{2-9}$$

其中式 (2-8) 为前向映射公式，式 (2-9) 为后向映射公式。

2.2 前向映射

通常情况下，一个整数位置 (x, y) 经过图像变换后，往往处于非整数的位置，此时要采用插值技术。

如图 (2-1) 所示，输入图像上整数点坐标映射到输出图像之后，变成了非整数点坐标。因此，需要将像素按照一定的权重分配至其周围四个像素点上。对于输出图像而言，其整数点像素值周围会有很多输入图像像素的映射，每个到其周围的非整数点像素值都会分配一定的灰度值到此整数像素点，将这些分配而来的像素值叠加，就是输出图像整数点位置的像素值。由于这个分配、叠加的特性，前向映射法有时也叫做**像素移交映射**。

因此，对于前向映射而言，输出图像某一点的像素值不能直接得到，需要遍历输入图像的所有像素值，对其进行坐标变换，分配像素值到整数位置，才能得到输出图像各像素点的像素值，这是前向映射的缺点。

2.3 后向映射

相比前向映射，后向映射较为直观。在这种情况下，我们知道输出图像上整数点位置 (x', y') 在变换前位于输入图像上的位置 (x, y)，一般而言这个是非整数点的位置，利

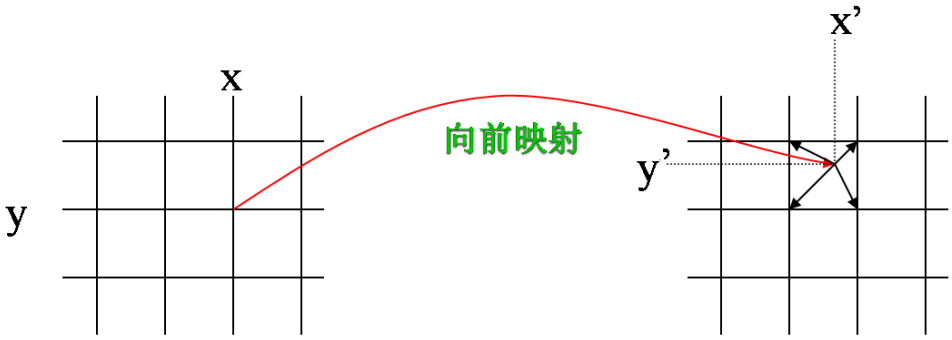

图 2-1: 前向映射示意图

用其周围整数点位置的输入图像像素值进行插值，就得到了该点的像素值。遍历输出图像，经过坐标变换、插值两步操作，就能将其像素值逐个地计算出来，因此向后映射又叫**图像填充映射**，如下图 (2-2) 所示。

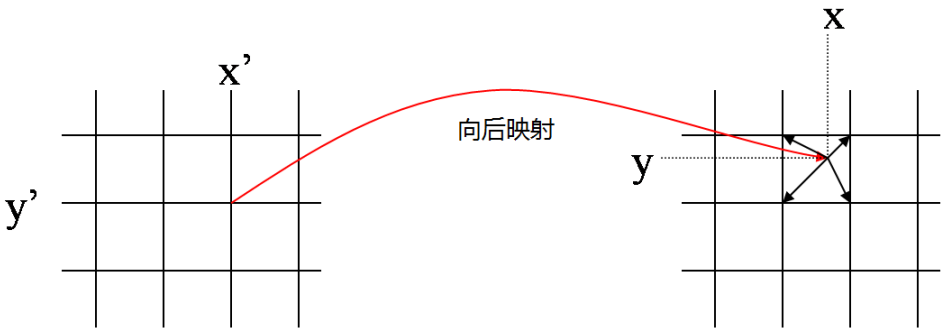

图 2-2: 后向映射示意图

2.4 插值算法

常使用的插值算法为**双线性插值**，其示意图如下图 (2-3) 所示。

对于后向插值而言，这种情况下输出图像上某点的像素值 $I'(x',y')$ 映射到 $f(x,y)$，而 $f(x,y)$ 由输入图像上四点像素值叠加而成。

$$f(x,y) = (1-x)(1-y)f(0,0) + (1-x)yf(0,1) + x(1-y)f(1,0) + xyf(1,1) \tag{2-10}$$

很容易验证四个权重系数之和为 1，插值后图像亮度不变。

对向前插值，输入图像某点 $I(x,y)$ 变换到输出图像 (x',y') 的位置，因此需要将其

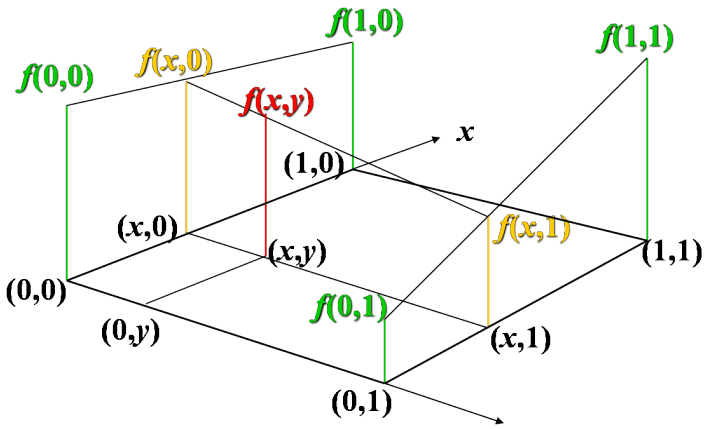

图 2-3: 双线性插值算法示意图

像素值分配到 $f(0,0)$，$f(0,1)$，$f(1,0)$，$f(1,1)$ 四个位置。分配方式为:

$$f(0,0) = (1-x)(1-y)f(x,y) \tag{2-11}$$

$$f(0,1) = (1-x)yf(x,y)) \tag{2-12}$$

$$f(1,0) = x(1-y)f(x,y) \tag{2-13}$$

$$f(1,1) = xyf(xy) \tag{2-14}$$

对于前向映射而言，虽然分配系数和为 1，但输出图像上每个点的像素值是多个分配值叠加而成的，因此不能保证多有分配到其上的权重之和为 1。因此必须记录下所有分配到其上的权重并累加起来，最后利用累加权重进行**归一化**，才可以得到正确的插值结果。

2.5 OpenMP 基本概念

OpenMP 是一种用于共享内存并行系统的多线程程序设计方案，支持的编程语言包括 C、C++ 和 Fortran。OpenMP 提供了对并行算法的高层抽象描述，适合在多核 CPU 机器上的并行程序设计。编译器根据程序中添加的 pragma 指令，自动将程序并行处理，使用 OpenMP 降低了并行编程的难度和复杂度。当编译器不支持 OpenMP 时，程序会退化成普通 (串行) 程序。程序中已有的 OpenMP 指令不会影响程序的正常编译运行。

OpenMP 采用 fork-join 的执行模式。开始的时候只存在一个主线程，当需要进行并行计算的时候，派生出若干个分支线程来执行并行任务。当并行代码执行完成之后，分支线程会合，并把控制流程交给单独的主线程。一个典型的 fork-join 执行模型的示意图如下图 (2-4) 所示：

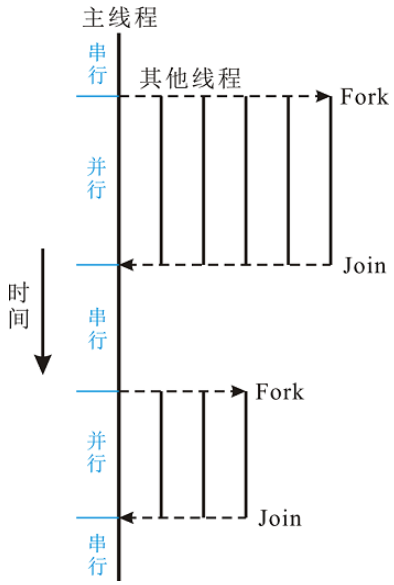

图 2-4: 典型的 fork-join 执行模型的示意图

OpenMP 编程模型以线程为基础，通过编译制导指令制导并行化，有三种编程要素可以实现并行化控制，他们分别是编译制导、API 函数集和环境变量。

2.6 OpenMP 编译指导

编译制导指令以 #pragma omp 开始，后边跟具体的功能指令，格式如：#pragma omp 指令 [子句 [, 子句] …]。

1. 常用的功能指令如下：

   - parallel：用在一个结构块之前，表示这段代码将被多个线程并行执行；
   - for：用于 for 循环语句之前，表示将循环计算任务分配到多个线程中并行执行，以实现任务分担，必须由编程人员自己保证每次循环之间无数据相关性；
   - parallel for：parallel 和 for 指令的结合，也是用在 for 循环语句之前，表示 for 循环体的代码将被多个线程并行执行，它同时具有并行域的产生和任务分担两个功能；
   - sections：用在可被并行执行的代码段之前，用于实现多个结构块语句的任务分担，可并行执行的代码段各自用 section 指令标出（注意区分 sections 和 section）；
   - parallel sections：parallel 和 sections 两个语句的结合，类似于 parallel for；
   - single：用在并行域内，表示一段只被单个线程执行的代码；
   - critical：用在一段代码临界区之前，保证每次只有一个 OpenMP 线程进入；

- flush：保证各个 OpenMP 线程的数据影像的一致性；
- barrier：用于并行域内代码的线程同步，线程执行到 barrier 时要停下等待，直到所有线程都执行到 barrier 时才继续往下执行；
- atomic：用于指定一个数据操作需要原子性地完成；
- master：用于指定一段代码由主线程执行；
- threadprivate：用于指定一个或多个变量是线程专用，后面会解释线程专有和私有的区别。

2. 相应的 OpenMP 子句为：

- private：指定一个或多个变量在每个线程中都有它自己的私有副本；
- firstprivate：指定一个或多个变量在每个线程都有它自己的私有副本，并且私有变量要在进入并行域或任务分担域时，继承主线程中的同名变量的值作为初值；
- lastprivate：是用来指定将线程中的一个或多个私有变量的值在并行处理结束后复制到主线程中的同名变量中，负责拷贝的线程是 for 或 sections 任务分担中的最后一个线程；
- reduction：用来指定一个或多个变量是私有的，并且在并行处理结束后这些变量要执行指定的归约运算，并将结果返回给主线程同名变量；
- nowait：指出并发线程可以忽略其他制导指令暗含的路障同步；
- num_threads：指定并行域内的线程的数目；
- schedule：指定 for 任务分担中的任务分配调度类型；
- shared：指定一个或多个变量为多个线程间的共享变量；
- ordered：用来指定 for 任务分担域内指定代码段需要按照串行循环次序执行；
- copyprivate：配合 single 指令，将指定线程的专有变量广播到并行域内其他线程的同名变量中；
- copyin：用来指定一个 threadprivate 类型的变量需要用主线程同名变量进行初始化；
- default：用来指定并行域内的变量的使用方式，缺省是 shared。

3. API 函数。除上述编译制导指令之外，OpenMP 还提供了一组 API 函数用于控制并发线程的某些行为，下面是一些常用的 OpenMP API 函数以及说明：

- omp_in_parallel：判断当前是否在并行域中
- omp_get_thread_num：返回线程号

- omp_set_num_threads：设置后续并行域中的线程格式
- omp_get_num_threads：返回当前并行区域中的线程数
- omp_get_max_threads：获取并行域可用的最天线程数目
- omp_get_num_procs：返回系统中处理器的个数
- omp_get_dynamic：判断是否支持动态改变线程数目
- omp_set_dynamic：启用或关闭线程数目的动态改变
- omp_get_nested：判断系统是否支持并行嵌套
- omp_set_nested：启用或关闭并行嵌套

4. 环境变量。OpenMP 中定义一些环境变量，可以通过这些环境变量控制 OpenMP 程序的行为，常用的环境变量：

- OMP_SCHEDULE：用于 for 循环并行化后的调度，它的值就是循环调度的类型；
- OMP_NUM_THREADS：用于设置并行域中的线程数；
- OMP_DYNAMIC：通过设定变量值，来确定是否允许动态设定并行域内的线程数；
- OMP_NESTED：指出是否可以并行嵌套。

3 实验过程

3.1 图像旋转

在进行实验过程中，我们首先设计前向映射的图像旋转程序，在此程序中，不进行线性插值，将映射完的浮点数类型的坐标数值直接变为整数数值。

利用此算法进行实验，将图 (3-5) 顺时针旋转 30 度，得到结果如图 (3-6) 所示。可以看到基于此算法旋转过后的图像有明显的锯齿。

图 3-5: 原始图像

图 3-6: 基于原始旋转算法旋转后图像

3.2 双线性插值

为了减少锯齿，提高图像旋转质量，编写线性插值函数，将图片 (3-5) 顺时针旋转 30 度，得到结果如图 (3-7) 所示。可以看到图像锯齿明显减少，说明此算法是有效的。

图 3-7: 基于前向映射的插值旋转图像

3.3 OpenMP 并行实现图像旋转

在上述实验的基础上对程序加入 OpenMP 并行算法，以提高程序运行的效率。根据 2.6 节中的知识，在程序前加入以下语句来实现 OpenMP 并行。

```
1 #pragma omp for schedule(static)
```

首先关闭 OpenMP 并行，运行算法，将 600×385 大小的图 (3-5) 顺时针旋转 30 度，运行 10 次，记录每次运行的时间，如下表所示。

表 3-1: 无 OpenMP 并行时算法的运行时间

次数	1	2	3	4	5	6	7	8	9	10
时间 (ms)	4.96	4.96	4.96	4.96	4.96	4.97	4.66	4.41	4.35	4.96

可见其平均时间为 4.82ms。

再开启 OpenMP 并行，将 600×385 大小的图 (3-5) 顺时针旋转 30 度，运行 10 次，记录平均运行时间，并改变并行数，得到平均运行时间如下表 (3-2) 所示。

从数据中可以看出，存在 OpenMP 并行时比串行时的速度快了 40%，说明并行的资源利用率比串行高很多。同时我们可以看到对于 OpenMP 并行，其运行速度先增加后减小，说明在并行数较小时，并行对算法处理速度有提高的作用，但在之后由于线程

表 3-2: OpenMP 并行时算法的运行时间

线程数	1	2	3	4	5	6	7	8	9	10
平均时间 (ms)	3.39	2.94	2.54	2.46	2.37	2.42	2.26	2.25	2.94	3.09
线程数	11	12	13	14	15	16				
平均时间 (ms)	3.05	2.91	2.99	2.99	2.82	2.97				

数的增加，线程同步需要的计算量增大，因此导致了速度的下降。同时我们还发现在线程数为 12 至 16 时，运行速度基本不变，没有因为增加了线程数就对运行速度有明显的影响，从而达到一个上限。

4 实验感想

在本次实验中，首先进行了相关知识的调研，了解了图像前向映射和后向映射的基本原理，并阅读了相关算法，此算法较为简单，但在离工程实际还有一些差距，因为在利用前向映射对图片进行旋转后，其坐标不是整数，如果直接粗暴取整，会导致图片的信息出现丢失，同时导致图片的分辨率下降，这是我们不希望看到的现象。

因此在图像旋转后我们对于其非整数的坐标不能粗暴取整，而是要进行一定的处理，保证图像的信息最大限度的保留，而最常用的方法就是双线性插值法 (Bilinear Interpolation)，利用双线性插值法成功解决了上述问题，达到了图像旋转后信息的最大保留。最终利用了 OpenMP 并行技术来进行图像旋转实验，实验结果表示，并行技术确实极大的提高了图像的处理速度。