# 计算机视觉实践报告（四）

## 1.实验目的

* 图像视差匹配，通过立体匹配(Stereo Matching)得到两张图像的视差图，并进行分析。

### 2.实验原理

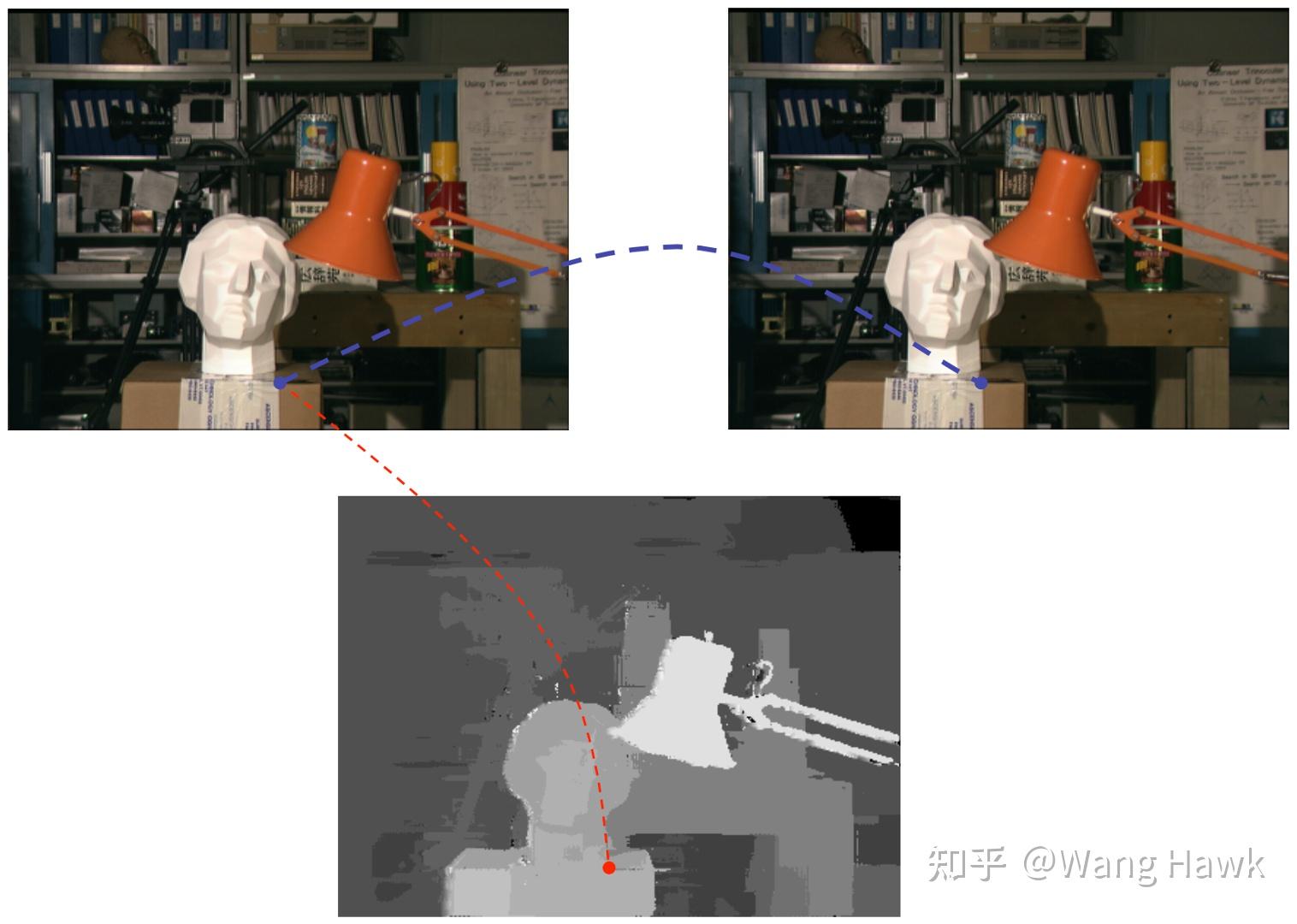
#### 2.1图像视差匹配

在计算机视觉领域中，图像视差匹配（Image Disparity Matching）是一个经典的问题，它是通过计算不同图像之间相对深度的过程来寻找两个或多个视角下的相同物体。视差匹配主要用于三维建模、视差测量等任务中。

图像视差匹配的主要思路是基于左右相邻两幅图像之间的像素点之间的位移或视差来计算相对深度。由于左图和右图存在一定的位置和角度的偏移，因此图像像素点之间存在差异。这种差异被称为视差，在我们的眼睛中会造成深度感知。

在计算机视觉中，实现图像视差匹配的方法有很多：SIFT（尺度不变特征变换）、SURF（加速稳健特征）、ORB（Oriented FAST and Rotated BRIEF）等。这些方法都是基于特征提取和匹配的技术，它们将图像表示为特征点的集合，并通过比较这些特征点之间的相似性来进行匹配和识别。

除了基于特征提取和匹配的方法，还有更为直接的像素点匹配方法，如基于灰度值的匹配、基于相关性的匹配、基于卷积神经网络的匹配等。



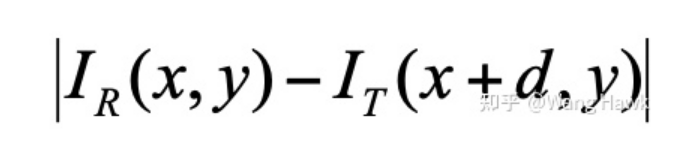
#### 2.2立体匹配

立体匹配（Stereo Matching）也称为图像视差匹配（Image Disparity Matching），它是通过比较图像中不同位置的像素之间的位移或视差来计算相对深度的过程。在成对的图像中，由于相机的左右位置不同，同一对象在两张图像中的位置会有所不同，这种不同被称为视差。立体匹配的目标是找到左右两幅图像中相应像素之间的关系，从而计算出图像中的深度信息，即三维结构。

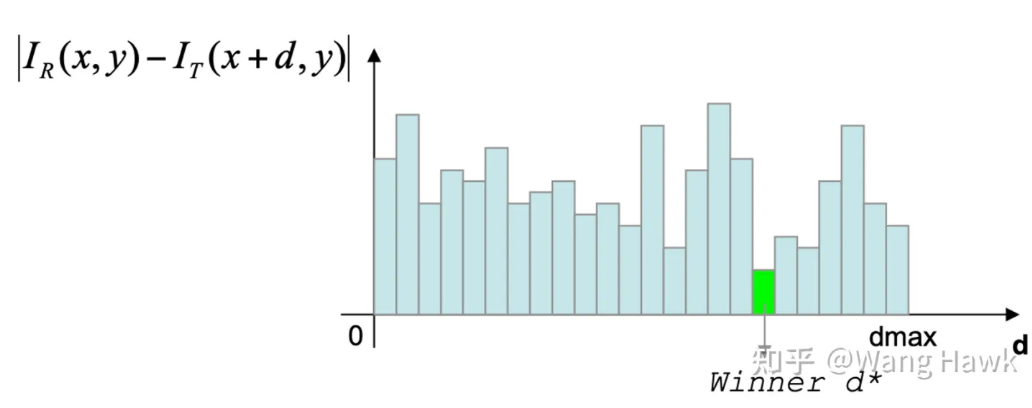
立体匹配是计算机视觉领域中的一个经典问题，其可以应用于机器人感知、三维重建、虚拟现实等多个领域。立体匹配的常用方法包括基于局部区域的匹配方法、基于全局优化的方法（如基于能量函数的方法）、基于卷积神经网络的方法等。由于立体匹配本身的复杂性和实际应用场景的挑战，仍然存在着一些未解决的问题和难点，例如性能和精度的平衡、遮挡和透明物体的处理、低纹理区域的匹配等。

一般来说，我们会定义某种匹配代价，用来衡量两个像素的相似程度。那么匹配代价最低的那个像素，就会被选为同名点。

最基本的匹配代价就是两个像素的像素值的绝对差（我们可以暂且把图像当做是单通道的灰度图像），那么匹配代价就是：

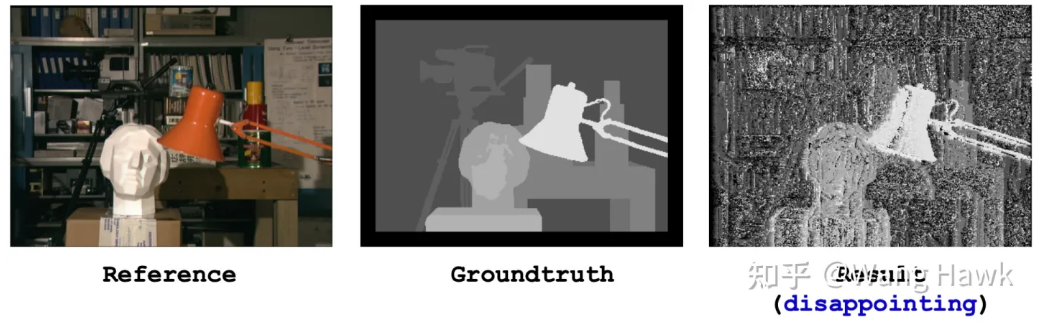


如果最小视差为0，候选的视差值d的值域范围是从0到的整数，那么可视化一下所有的匹配代价如下图所示：



从所有的候选像素中挑选匹配代价最低的作为最终的同名点，这个策略被称为Winner Takes All (WTA)，所谓的赢者通吃

那么，按照这种方法算出的视差图如何呢？我们看看下面的结果，并和Groundtruth做下对比：



肉眼判断视差图是否有错误可以通过下面几个准则来观察：

视差是近大远小的，表现到视差图上则是近亮远暗

同一个物距的物体表面的视差亮度是一致的

在物体的边缘，如果物距发生了突变，那么视差图上也应该有突变的边缘

物距渐变时，视差图上也是平滑变化的

很明显，相比理想结果，这种方法的得到的视差图充满了噪声，很多地方可以肉眼可见明显的错误。 我们上面这种简单的立体匹配算法很明显是不足的。 接下来，我会先谈谈立体匹配的困难之处，再分析一下解决这些困难的方法。

## 3立体匹配的困难之处

1. 光照不一致及噪声影响
2. 镜面反射导致信息不一致
3. 近距离拍摄时的透视形变和视角不一致，导致像素匹配时的唯一性约束失效
4. 物体表面缺乏纹理
5. 物体表面纹理重复
6. 透明物体导致寻找同名点失败
7. 遮挡和深度导致不连续

## 4. 立体匹配的基本思路和流程

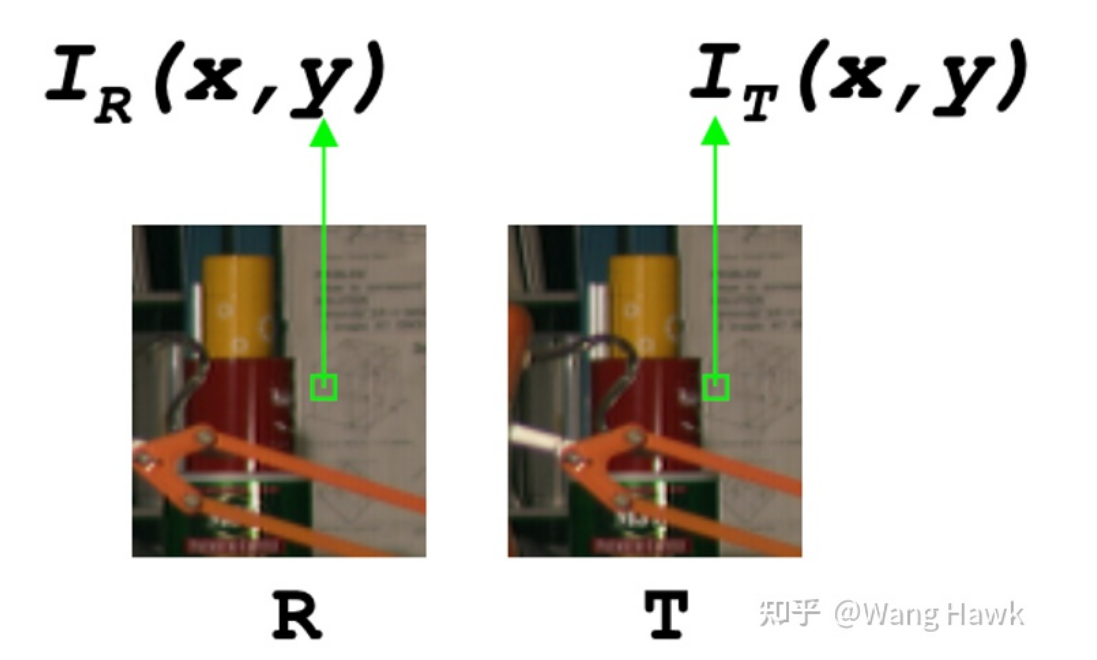
#### 4.1图像预处理

立体匹配的困难最终的反映为搜索同名点失败上，如果是因为两幅图像的亮度、噪声不一致，一般会先对图像做预处理，使得两幅图像的整体质量区域一致。

#### 4.2代价计算

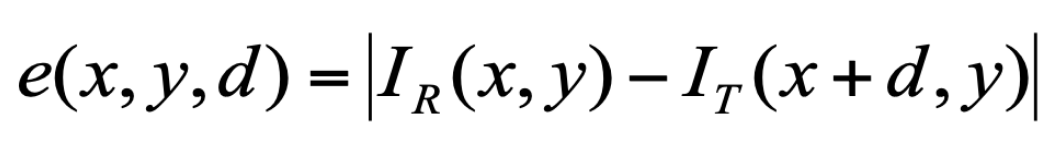
刚才计算同名点匹配代价时采用的是单个像素点的绝对值的差异，这样很容易受到噪声的干扰。规避方法有两类：

##### 4.2.1换用更鲁棒的单像素的代价函数

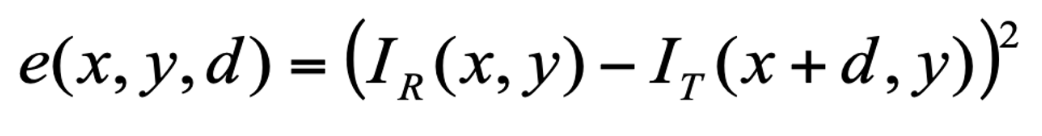


这一类方法依然是利用左右图像上单一的像素点进行代价计算。人们设计了很多不同的代价函数计算方式，分别都有自己的优缺点，这里列出如下：

像素差异绝对值 Absolute Differences

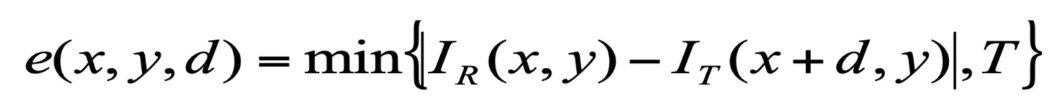


平方差 Squared Differences



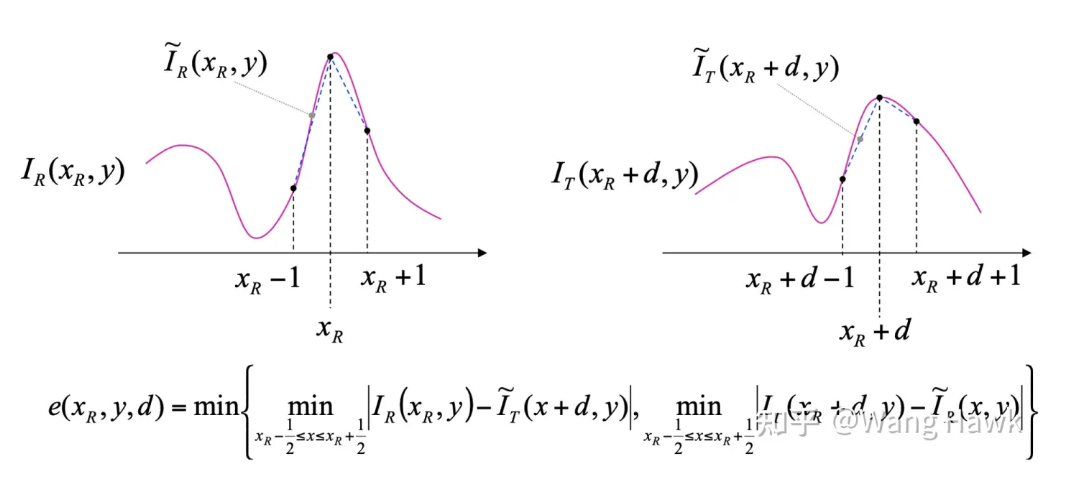
上面两种代价函数都容易受到噪声的干扰，于是就有了更加鲁棒的函数，比如：

截断绝对差 Truncated Absolute Differences (TAD）



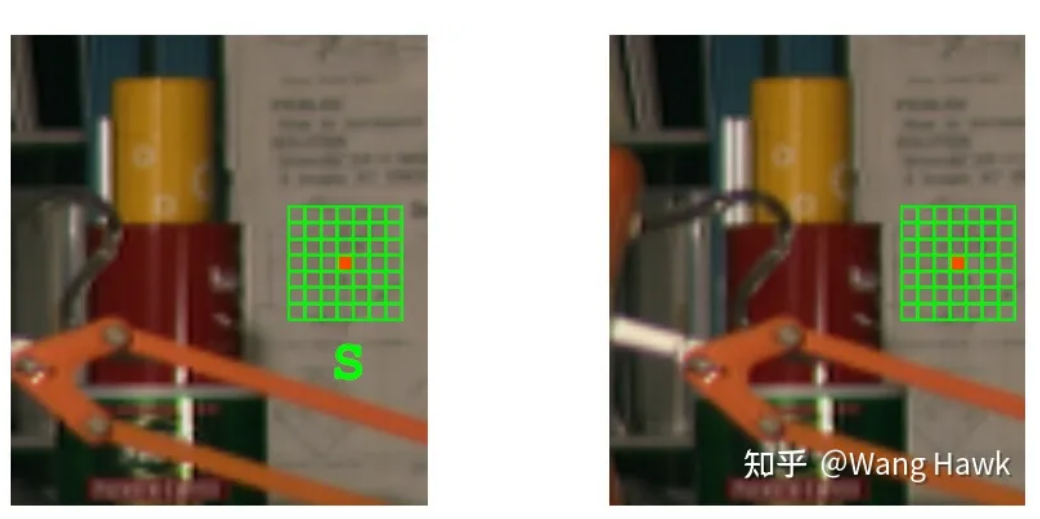
将绝对差限制在<=T的范围内，避免噪声引起过大的代价。这是一种简单的策略，其实并不比上面两种好多少，因为T很难确定

还有更复杂的，通过某种对图像内容不敏感的方式来确定两个像素的不相似度。它考虑了当前像素，以及其旁边的两个像素。



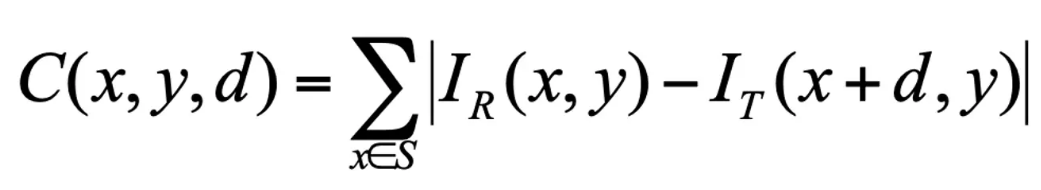
总之，仅仅考虑单个像素，还是很难得到好的结果。更好的方式是通过计算所关注的像素点的邻域的整体情况，来提升信噪比，减少噪声的影响。 我们把这个邻域范围称为“支持窗(Support Window)”，通过支持窗内所有像素来计算一个匹配代价值。

##### 4.2.2采用邻域支持窗来计算整体代价

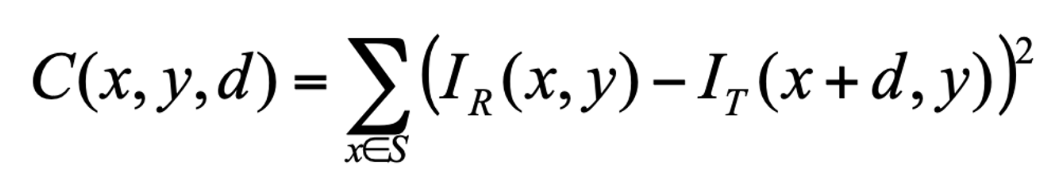


这种策略就是把单个像素的计算转换为一个支持窗内的整体计算了，比如：

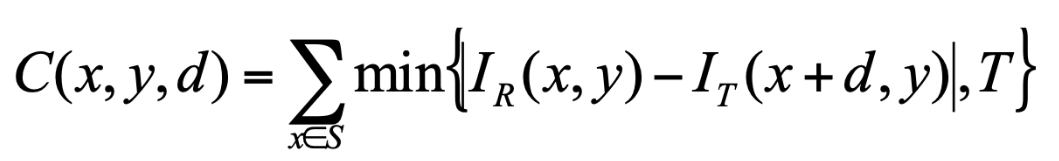
像素绝对差值和 Sum of Absolute differences (SAD)



像素差值平方和Sum of Squared differences (SSD)

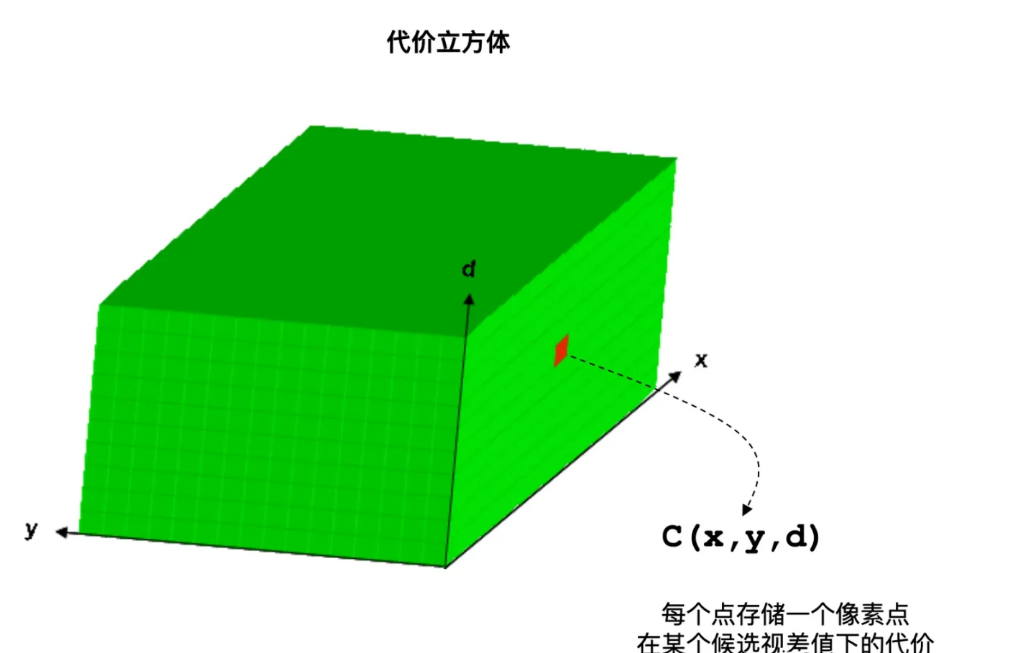


截断绝对差值和 Sum of truncated absolute differences (STAD)



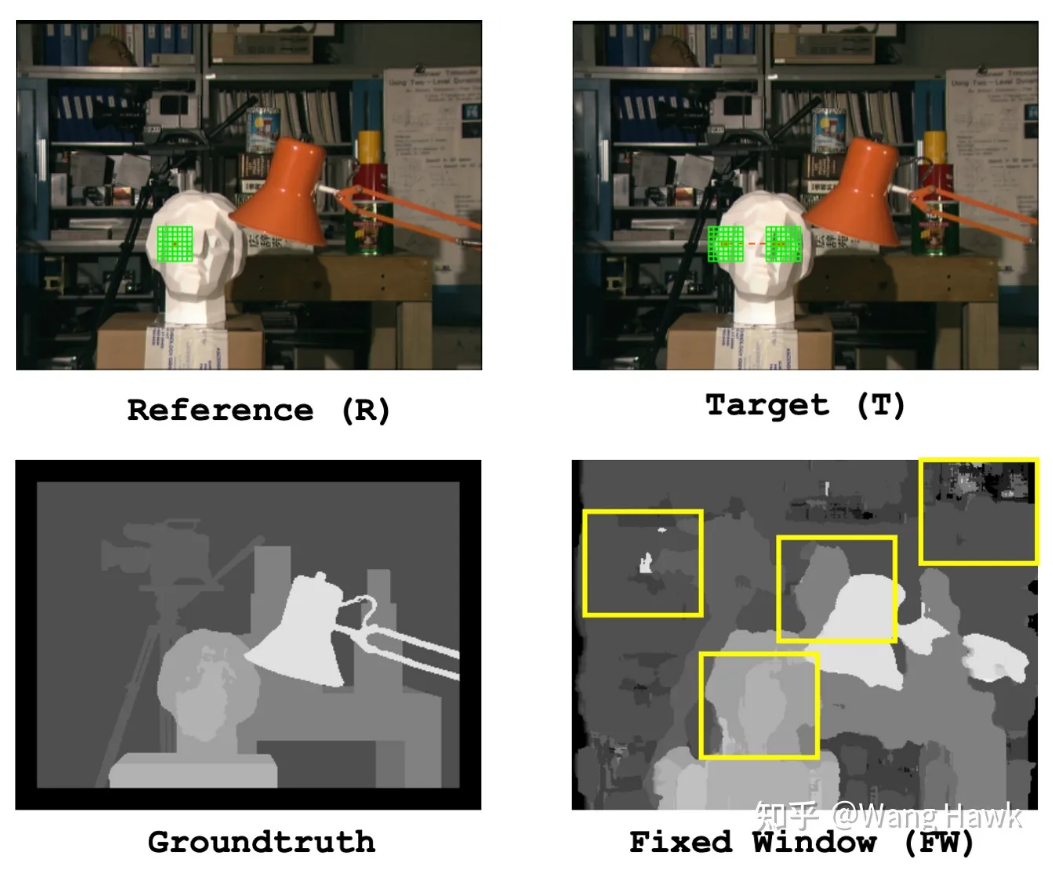
除了这些简单的代价函数，还有更多方法，比如利用两个图像的互相关信息的，利用图像梯度域信息的，或是利用一些非参数方法的，等等。

总之，我们可以为R中的每个像素点和选定的T中的像素点计算一个代价，并且这个代价还具有很高的区分度。 如前所述，我们是在一个范围[dmin, dmax]中搜索匹配点，因此对任何一个R中的像素点，可以算出dmax - dmin + 1种代价值。如果图像的宽高分别为W和H，那么我们总共会得到W x H x (dmax - dmin + 1）种代价值。 所有这些代价值可以存储到一个立方体中，这就是所谓的代价立方体，如下所示：



#### 4.3代价聚合和视差优化

通过支持窗计算代价已经对图像的噪声、光照不一致等等提升了一定的鲁棒性，但依然有很多问题遗留下来。我用一个基本的例子加以说明。让我们用一个改进的流程来处理之前给出的一对图像：计算一个简单固定尺寸方形支持窗内的截断绝对差值和STAD，然后用WTA策略计算视差。



看看上面的结果，比起最基础的方案，视差图似乎平滑了很多，没有了大片的噪声。但是很多局部是错误的，比如说灯罩边缘变得凹凸不平，背景出现异常的亮区，右上角也出现了异常的噪声，灯架断开，等等。

##### 4.3.1代价聚合

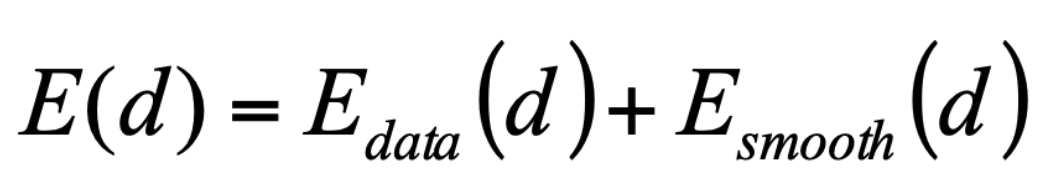
局部聚合思路是通过对代价立方体中同一视差的代价进行某种程度的聚合，来减少或消除错误代价的影响，这一步就是所谓的代价聚合(Cost Aggregation)。比如下面左图，同一个视差的窗口我们会扩大并将代价立方体中相应的代价聚合在一起。 而下面右图则说明在聚合过程中要避免混合不同视差的像素。



Stefano Mattoccia教授的讲义中介绍了各种各样代价聚合方法，它们一般来说是通过调整支持窗的位置、形状、窗内各像素的权重等等来完成聚合的。我会在下一篇文章中为你导读讲义中提到的各种各样的局部代价聚合方法。总之，通过局部的代价聚合，有可能得到非常不错的效果，比如下图中所示的一个利用了局部一致性的方案，相比FW的效果得到了很大的提升。

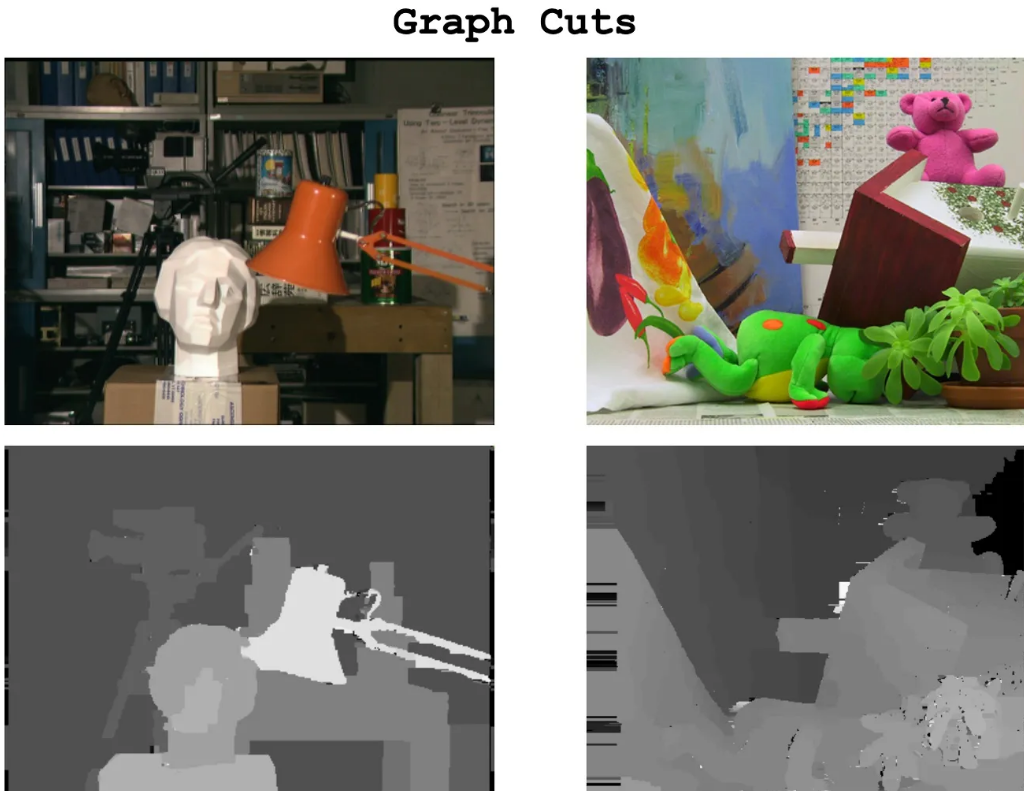
##### 4.3.2视差优化

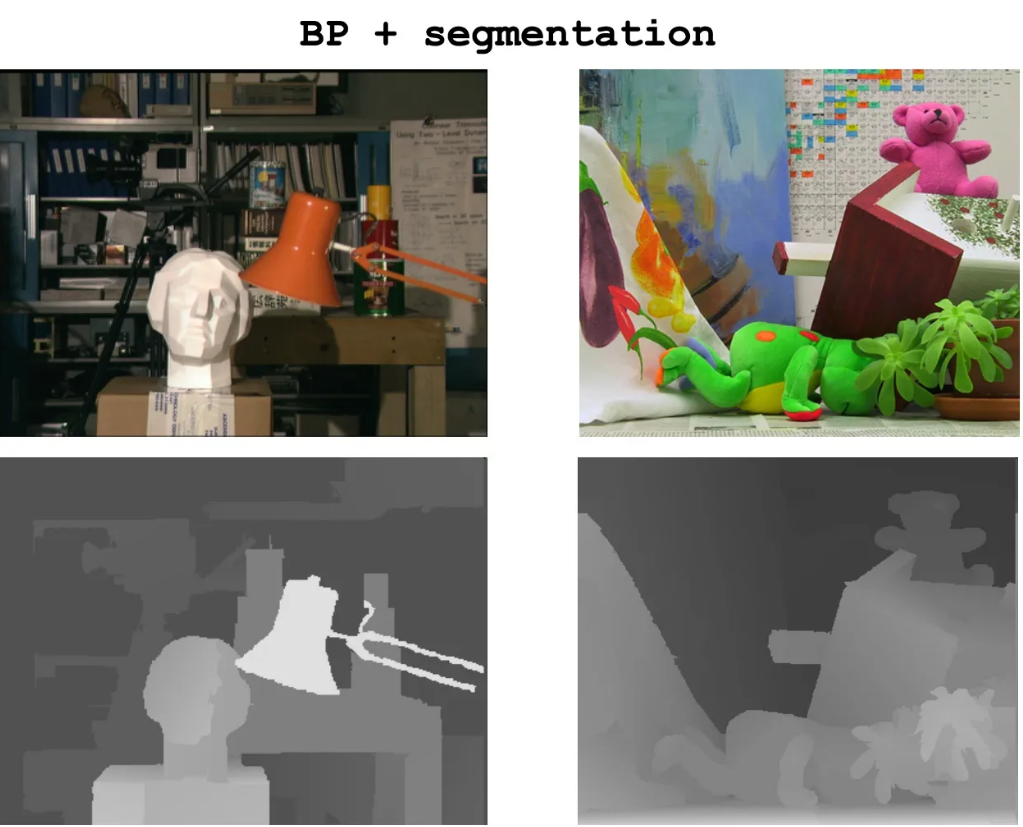
而全局优化思路，则是希望寻找到每个像素的最优视差结果，使得全局的、整体的匹配代价最小，这一步被称为视差优化(Disparity Optimization)。于是这个过程就变成了一个最优化某个能量函数的过程，该函数通常写成如下的形式：

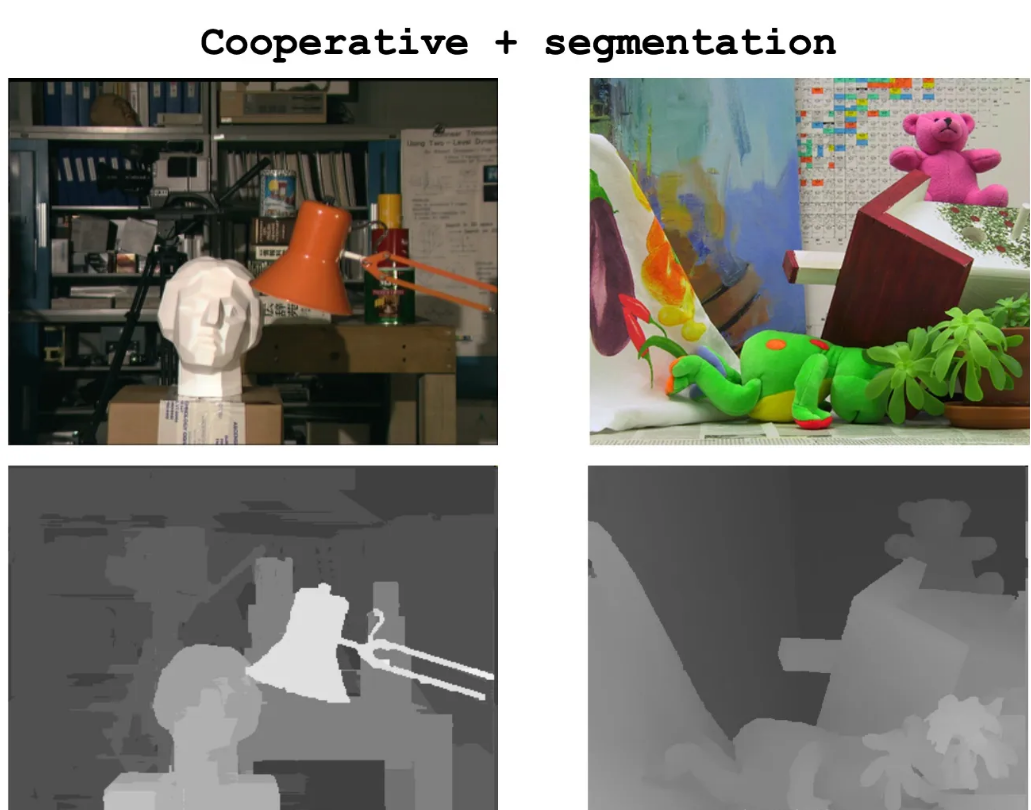


等号右边第1项是数据项，用于约束全局代价最小化。 但是代价立方体中通常含有噪声和错误，直接最小化求得的结果也会有很多问题，所以还需要第2项平滑项。 这一项一般用于给出某些额外的约束条件，比如通常假设整个图像的视差是平滑变化的。这样视差的大变化只会在场景内视差边缘处产生，一般也和图像内物体边缘高度相关。对上述能量函数的最优化过程是一个非常困难的问题，所以一般会有一些近似求解方法，还有就是用将全局能量函数的最优化转换为针对图像中的子部分的最优化，比如约束到某些扫描线方向上的最优化，然后利用动态规划或扫描线优化等方式去求解。

关于视差优化，我也会在后面专门的文章中再阐述细节。但至少当时，全局法的效果确实比起很多局部法要好，我们看下面这些例子，就很清楚了：





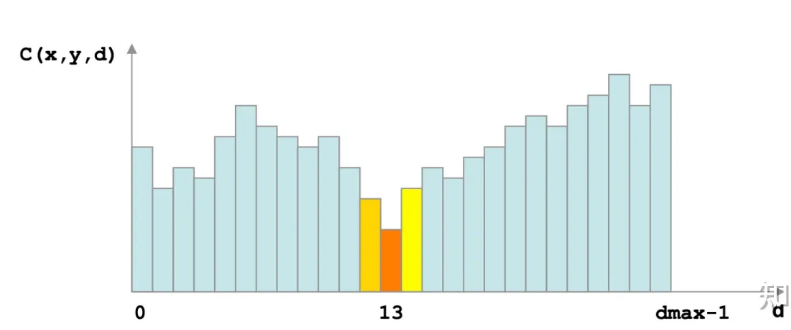


#### 4.4 视差后处理

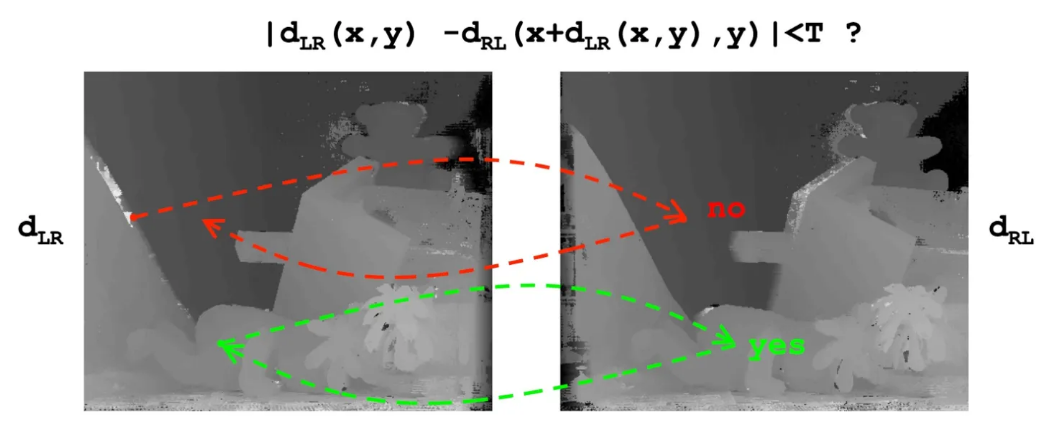
前面介绍的步骤最终将输出一张视差图，然而正如你已经看到的，即便是在上面那些受约束的场景，得到的视差图依然不是完美的，还是有很多错误。因此，还需要一个后处理的步骤，来消除其中的错误，得到更准确的视差图。

这一步需要解决哪些问题呢？这包括了：

亚像素插值：我们上面计算的视差值都是离散的整数值，但实际场景物体存在连续变化的视差，我们希望得到以浮点数表示的更精细的视差值。一般来说，会采用某种二次的抛物线插值法，得到连续的视差值，计算量也比较低，结果也不错。



噪声和错误消除：有时候会简单采用图像滤波的技术来处理视差图，也能得到不错的结果。从简单的中值滤波，到复杂的双边滤波都有人尝试。我会在之后的文章专门介绍一种强大的滤波类的处理方法。另外一个重要的技巧是双向匹配，这种方法分别以双目图像中左图和右图作为参考图像R计算两个视差图（缺点：增加了计算量)。然后它认为一对匹配点的视差值是互反的，也就是说一对正确匹配点的视差值会非常接近。如果不满足这个条件那么对应的视差值应该就是错误的。比如下面红色点的视差就计算错误了，而绿色点则是正确的。



## 5.代码使用

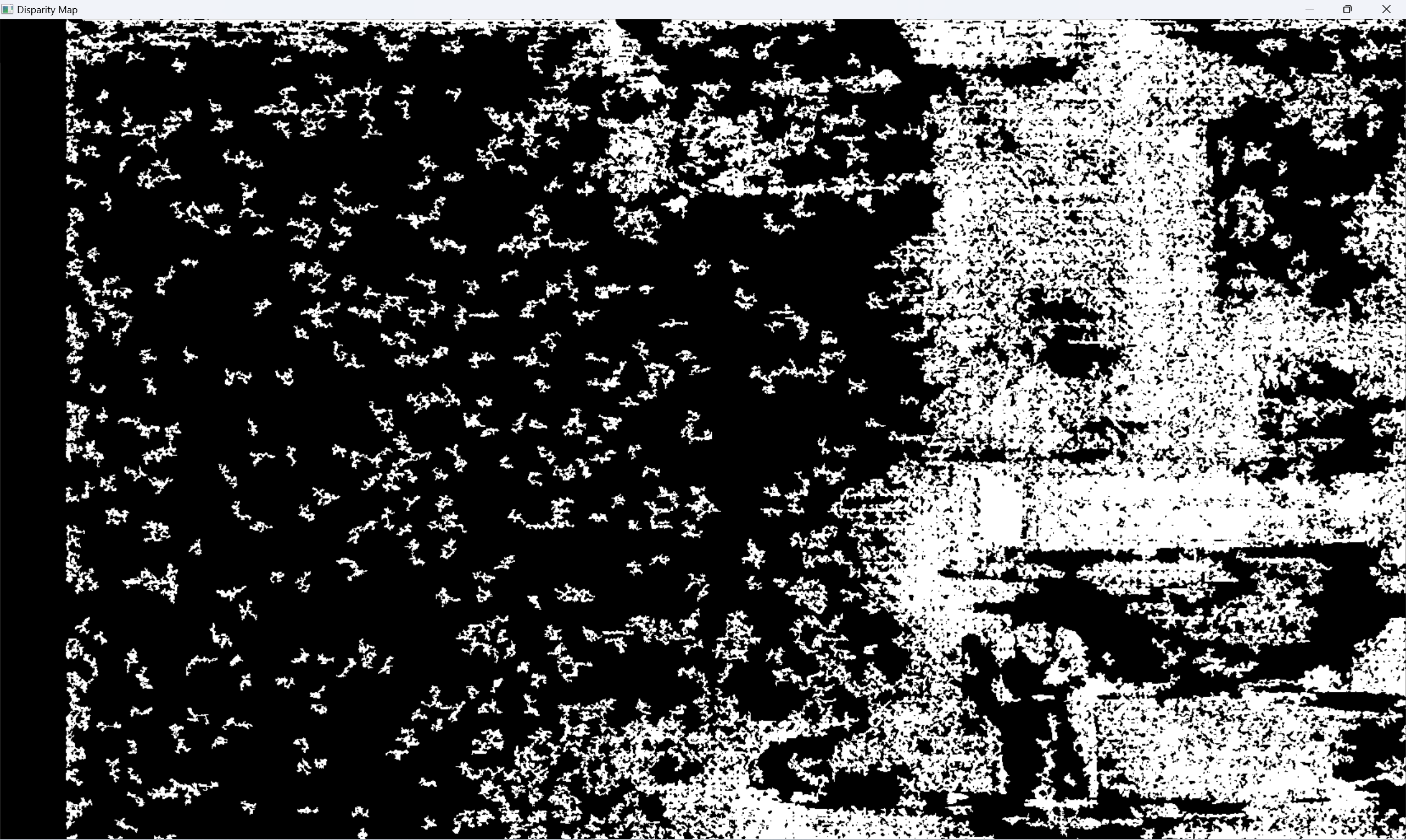
import cv2  
  
# 读取左右图像  
left\_img = cv2.imread('left.jpg', 0)  
right\_img = cv2.imread('right.jpg', 0)  
  
  
# 设置立体匹配算法参数  
window\_size = 3  
min\_disp = 0  
max\_disp = 16 \* 5  
stereo = cv2.StereoSGBM\_create(  
 minDisparity=min\_disp,  
 numDisparities=max\_disp - min\_disp,  
 blockSize=window\_size,  
 P1=8 \* 3 \* window\_size \*\* 2,  
 P2=32 \* 3 \* window\_size \*\* 2,  
 disp12MaxDiff=1,  
 uniquenessRatio=10,  
 speckleWindowSize=100,  
 speckleRange=32  
)  
  
# 计算视差图  
disp\_map = stereo.compute(left\_img, right\_img).astype('float32') / 16.0  
  
# 进行视差图归一化处理  
disp\_img = cv2.normalize(disp\_map, None, alpha=0, beta=255, norm\_type=cv2.NORM\_MINMAX)  
  
# 将视差图显示出来  
cv2.imshow('Disparity Map', disp\_img)  
cv2.waitKey(0)



left



Right



conclusion