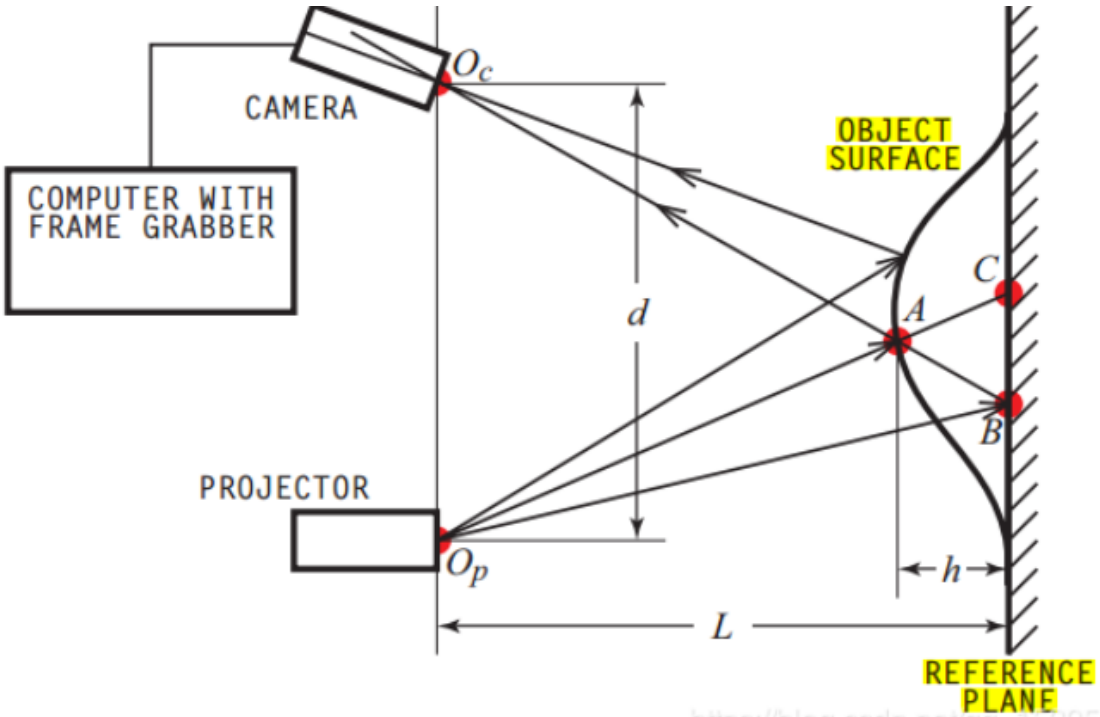
这篇文章记述了关于XXX的知识点。

深度信息求解

利用结构光技术的重建过程中，又分为两类，一类是有参考平面，一类是没有参考平面。

a: 有参考平面的系统中，只要求出参考平面和被测物体的相对相位差，利用PMP（相位测量轮廓法）方法即可求得深度信息。



$\frac{BC}{d} = \frac{h}{L-h}$

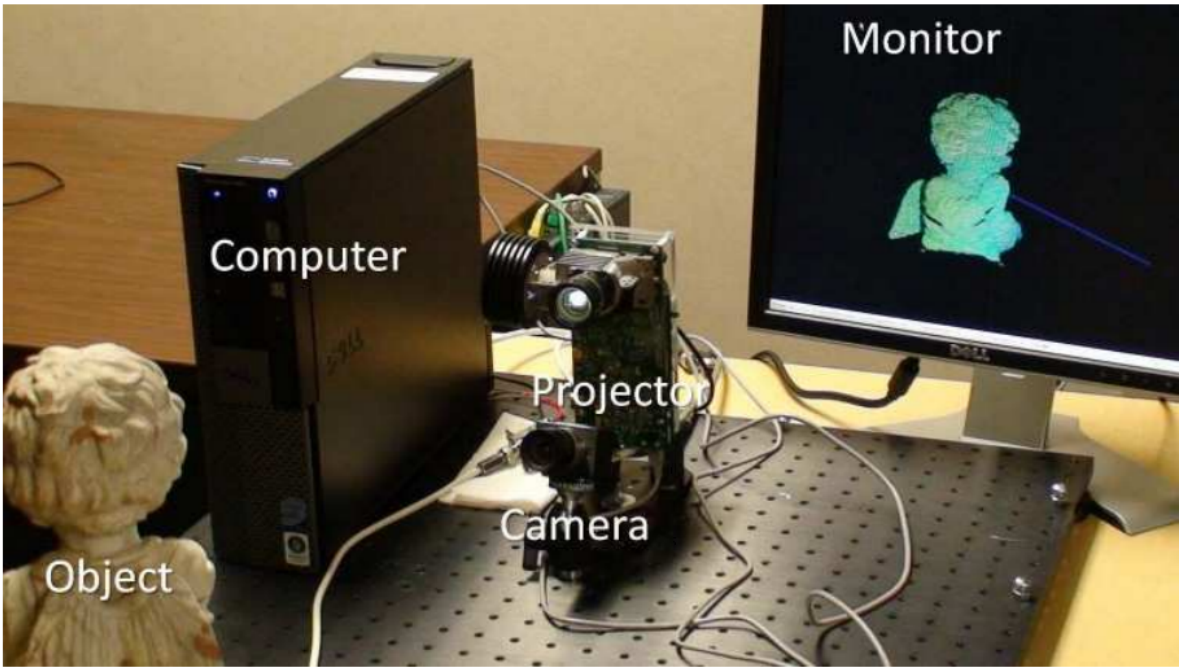
$h * d = BC * (L - h)$

$h * (d + BC) = BC * L$

$h = \frac{BC*L}{d+BC}$

$BC = \beta(\phi_C - \phi_B)$

$cos(\angle tLe) = \frac{Le^2+Lt^2-te^2}{2*Le*Lt}$



（图自 Dual-frequency pattern scheme for high-speed 3D shape measurement）

0615 上课时听到老师讲他的结构光系统里的投影仪是一台好几万的投影仪（The projector is composed of a Texas Instrument's Discovery 1100 board with ALP-1 controller and LED-OM with 225 ANSI lumens. The projector has a maximum frame rate of 150 fps at a resolution of 1024×768 and 8-bits per pixel grayscale.），为什么需要这么贵的设备呢？因为普通的商业投影仪和照相机等设备为了让用户看起来更舒服，会引入一个伽马系数为 2.2 的非线性变换（Non-linear response of optical device）,于是我就去查了一下非线性响应问题的资料。

相机的非线性亮度响应问题

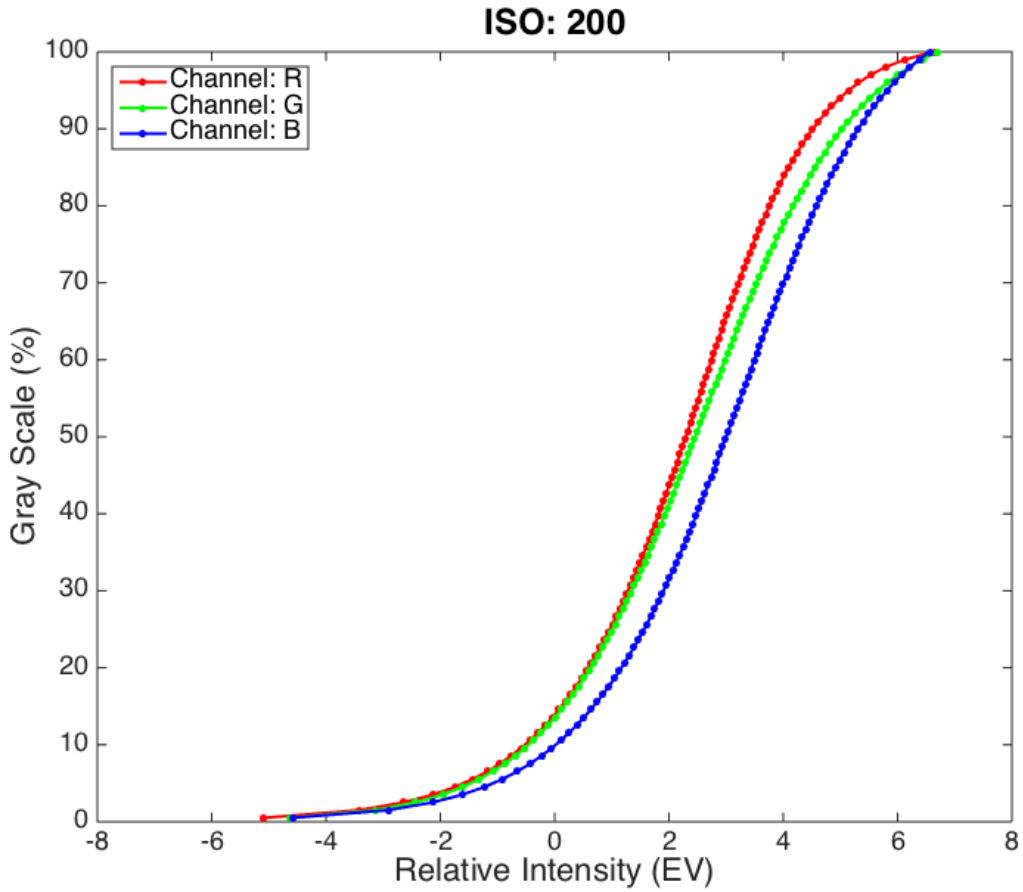
参考 https://zhuanlan.zhihu.com/p/23981690

https://www.zhihu.com/question/27467127#answer-10413243

相机的亮度响应是指拍摄场景的 真实亮度 与 成像后像素亮度 之间的关系。

最理想情况下，我们希望这个亮度响应是个线性的关系。就是说无论一个场景原始的亮度是多少，如果场景入射的光亮度变成了 2 倍，那成像后的像素值也应该变成 2 倍。

我们知道，数码相机的感光元件 CCD 或者 CMOS 对光线是非常敏感的，同时它们的线性程度非常好，在非常大范围内，数码相机感光元件的输出（电压）与亮度呈现良好的线性关系。 那么我们是不是可以认为，数码相机的亮度响应曲线就是一条直线呢？其实不然。 相机厂商为了更好地模拟人眼视觉的非线性效应，同时也为了更有效地记录很亮和很暗的场景， 会在感光元件的线性输出基础上，增加一个非线性变换，然后才输出到图像。



佳能 6D 相机的三通道响应曲线。

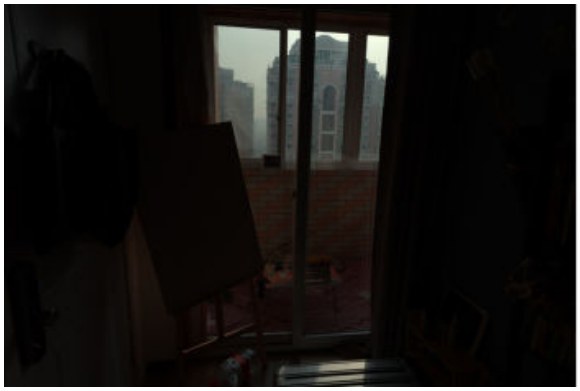
引入这样一种非线性关系，有什么好处呢？

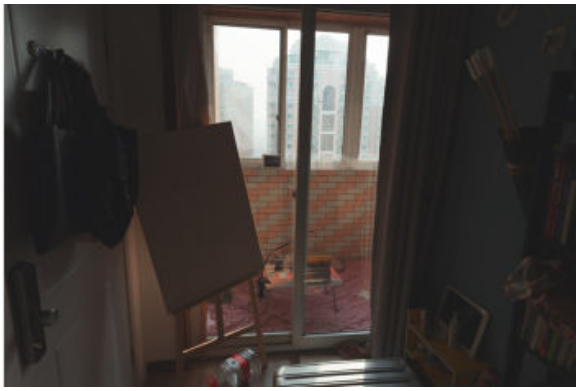
1、充分利用了人眼的视觉特性，更好地记录场景信息。

在一个固定场景中，人眼更多地注意中间亮度的情况，如果中间亮度范围内的对比度足够， 人眼就会觉得画面是通透的；反之如果中间亮度范围内对比度不足，就会觉得画面发灰。 引入这样的非线性，增加了中间亮度范围内的对比度，同时也部分保留亮部和暗部的信息不缺失。

2、让画面更接近人眼感官效果

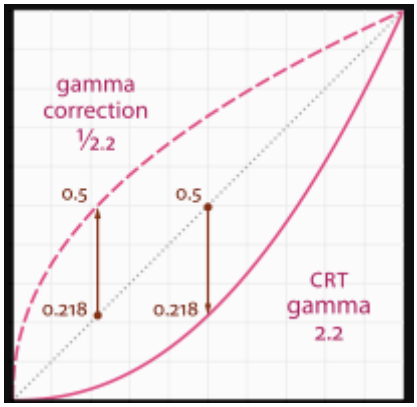
人眼对一个场景中各个不同亮度的感知是非线性的，并且人眼可以对极亮和极暗的场景做出反应。 而相对的，将照片放在显示器上或者打印出来，显示器或者纸张并不能给出太大的明暗对比， 因此需要把一个较大的明暗范围「压缩」到显示器或者打印纸张能够反应的明暗范围， 否则人眼看到会觉得缺失许多细节。 线性映射情况下，人眼看去容易产生太亮或者太暗的感觉，因此引入一定的非线性因素是有必要的。



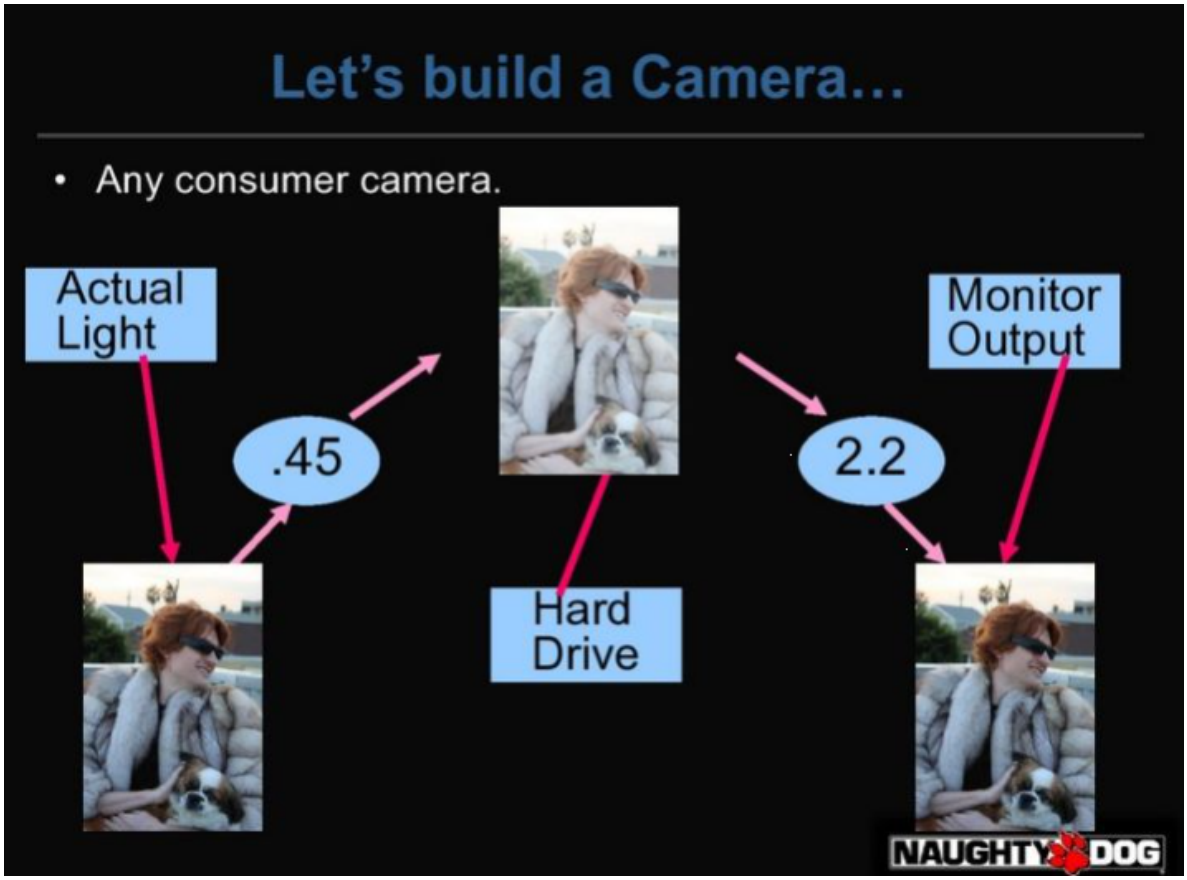


上面两张图中，左图是线性映射，右图采用类似亮度响应曲线的 S 形曲线映射。两张图片中像素的取值范围是一样的，然而人眼看来右侧的图更接近真实感受，中间亮度部分对比度足够，过亮和过暗的部分也能保留足够的细节。

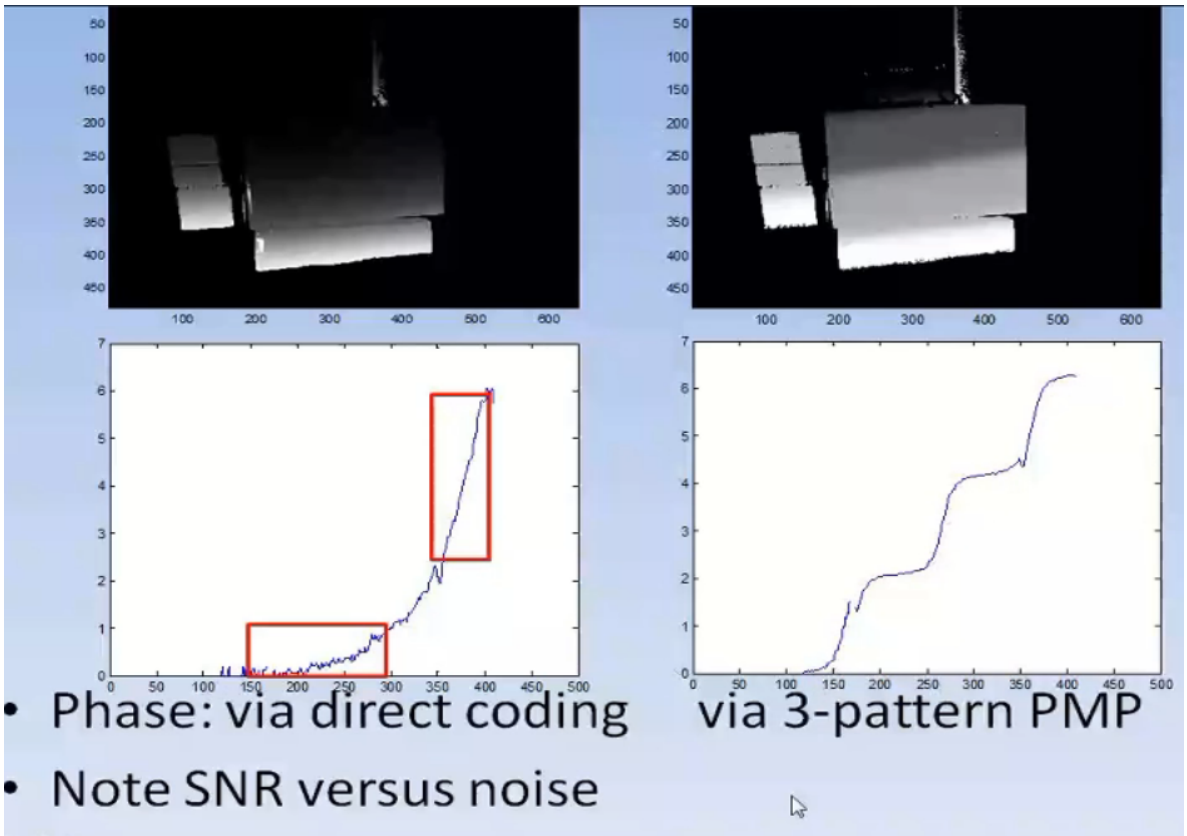
从人类的视觉特性说起。人类的视觉系统进化出了一个特性，黑暗环境下的辨识能力要强于明亮环境，这可能有助于我们及时发现黑暗中隐藏的危险。如果有兴趣，可以在计算机上绘制一条从白到黑的渐变条，你会发现明显黑色部分的色阶要多于白色部分。这并不是你的显示器问题，而是因为我们对于暗色的分辨能力远超过亮色。那么在有限的计算机颜色（民用显示器和操作系统中黑色到白色256个色阶）中，亮色和暗色均匀分布的话，那亮色部分就会精度过剩而暗色部分就会精度不足。如何解决这个问题？进行 Gamma 矫正。



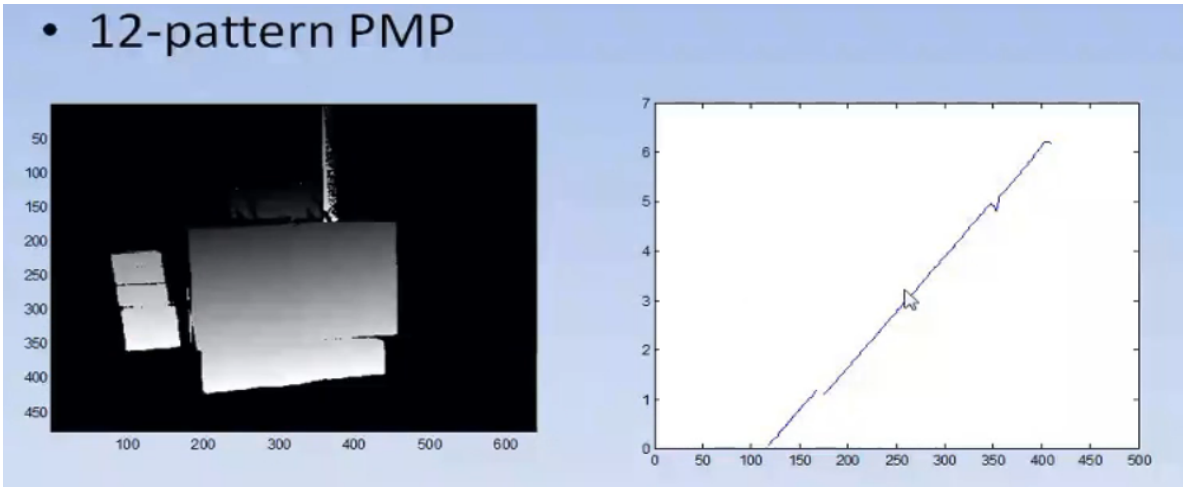
上图中的红色虚线就是图像的 Gamma 矫正曲线，一般都是2.2的倒数，0.454545..。观察矫正以后的曲线，原来0到0.5的区间被扩展到了0到~0.73，相应的大于0.5的区间被压缩到只有原来的一半左右。这样通过一次简单的计算，我们达到了不增加数据量的前提下提高可辨识精度（Perceptual precision）的目的。可能从来没有人告诉过你，存储在你硬盘上的图像，都是矫正过的，如下图，中间那个。



他后来又提及将结构光的条纹图片由三帧图片增加到四帧图片后可以显著减少非线性的影响。如下图所示：右图中的波浪形就是由非线性导致的（竖轴为$\frac{\text{采集到的光—环境光}}{\text{纯白光—环境光}}$），右图的整体上升趋势是直线，比左图要好很多，左图是直接利用$round(\frac{y_{\text{轴坐标}}}{Height}*255)$进行直接编码的，右图是利用正弦条纹进行编码的。



下面这幅图说明了只要你增加一个相位移动，它的非线性的影响就会急剧的降低：



具体原因是在 Gamma model and its analysis for phase measuring profilometry 这篇论文里面：

$$\Delta\phi = \arctan\left\{\frac{\sum_{k=1}^{\infty}[b_{kN}^{-}\sin(kN\phi)]}{1+\sum_{k=1}^{\infty}[b_{kN}^{+}\cos(kN\phi)]}\right\}. \qquad (29)$$

Obviously, the phase error $\Delta\phi$ is related to γ, ϕ, and N. It is independent of α and α_p in Eq. (7). Phase error, $\Delta\phi$, will converge to zero with increasing N because b_{kN}^{-} decreases dramatically with increasing N. The larger N, the smaller $\Delta\phi$, and the lessened effect from γ.

反正切函数的优化

> 本节参考 Dual-frequency pattern scheme for high-speed 3D shape measurement

包裹相位的计算是通过

$$\phi(x,y) = tan^{-1}\left(\frac{\sqrt{3}(I_1(x,y)-I_3(x,y))}{2I_2(x,y)-I_1(x,y)-I_3(x,y)}\right) \qquad ()$$

计算得到的。我很好奇这个公式前面的系数是从哪里来的？原来下面这个才是最原始的公式：

$$\phi(x,y) = tan^{-1}\frac{\sum_{n=0}^{N-1} I_n^c sin(\frac{2\pi n}{N})}{\sum_{n=0}^{N-1} I_n^c cos(\frac{2\pi n}{N})} \qquad ()$$

其中，N 是相位移动的总数量，n 代表是相移图片的索引。

由于我们经常使用的是三步相移，这里的 N 就等于 3，为了加快求反正切函数的速度，人为的构造了一个查找表，将n = 0,1,2 这三个值的 sin 函数值先求出来，就可以得到第一个公式了。

同理，当使用四步相移时 N = 4，构造出来的公式就如下所示：

$$\phi(x,y) = tan^{-1}[\frac{I_1^c - I_3^3}{I_0^c - I_2^c}] \qquad ()$$

极点的优化

公式推导

$I_n^p(x^p, y^p) = A^p + B^p cos(2\pi f y^p - 2\pi n/N)$

变频条纹的作用

> 本节参考 Dual-frequency pattern scheme for high-speed 3D shape measurement
>
> 概括来说，变频的作用主要是通过高频的正弦条纹来减少传感器带来的噪声影响，通时为了减少相位包裹的计算量，本文提出了一种双频的条纹公式如下：

$$I_n^p = A^p + B_1^p cos(2\pi f_h y^p - \frac{2\pi n}{N}) + B_2^p cos(2\pi f_u y^p - \frac{4\pi n}{N})$$

其中，I_n^p 是投影仪中像素的亮度，A^p, B_1^p和 B_2^p 是使8位色深投影仪的成像值保持在 0 到 255 之间的一些常量。f_h 是正弦条纹的高频，f_u 是正弦条纹的单位频率，值为1.

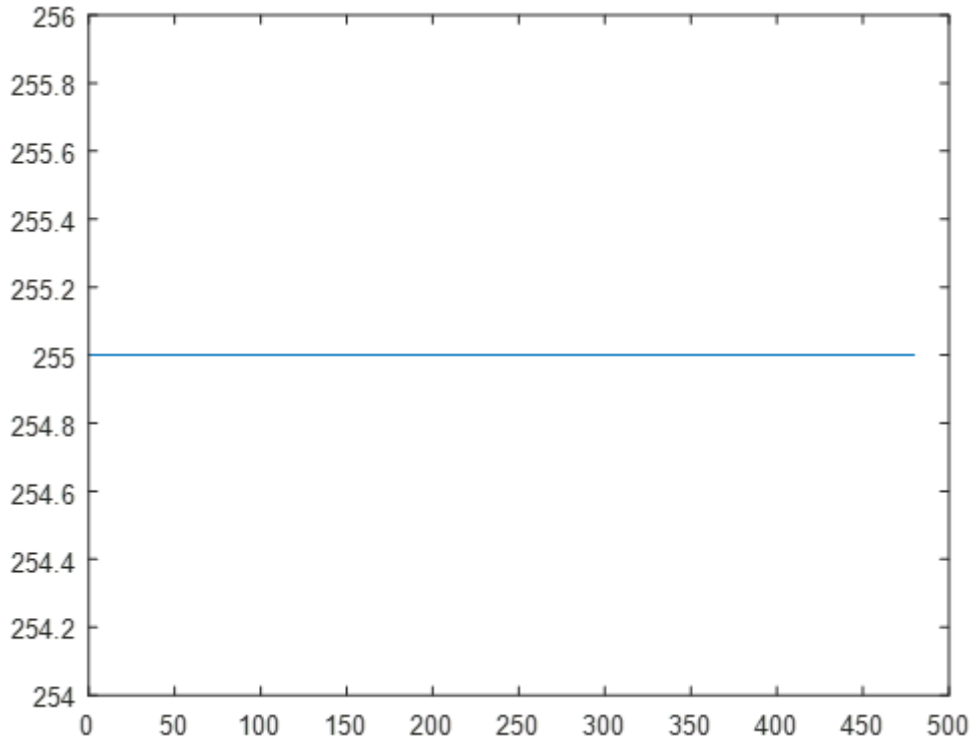
为什么$y^p = y^p / Height$？

仔细查看公式

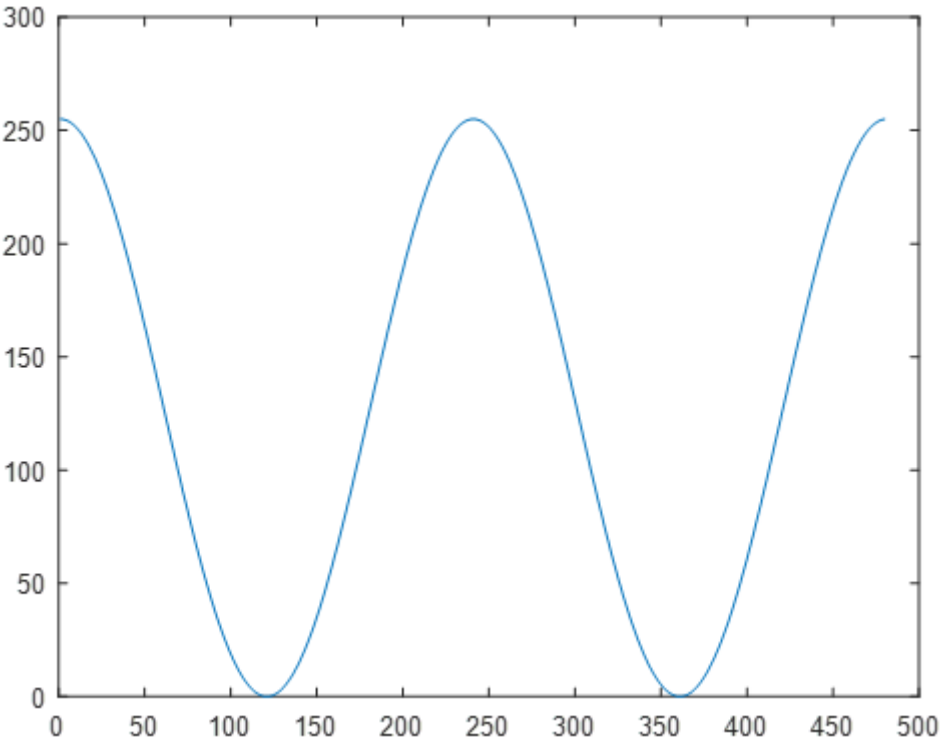
$I_n^p(x^p, y^p) = A^p + B^p cos(2\pi f y^p - 2\pi n/N)$

这里面有一个隐藏的易错点，其实$y^p = y^p / Height$ ，而不是简单的只是行索引。这是为什么呢？

如果只是简单的行索引的话，当 $n = 0$时，前面的 $2\pi f y^p$ 将会是 2π 的整数倍，由 cos 函数特性可知，2π 的整数倍的 cos 值将会是 1，所以会呈现出一条直线。



而把$y^p = y^p / Height$之后，图线变为：




```
close all;clear all;clc;
H = 480;
W = 640;

t = (0 : H - 1) / H;

vTruePhase = 2 * pi * t;

N = 4;
f = 2;

Ap = 127.5;
```

```
Bp = 255 - Ap;

vPSin = 0;
vPCos = 0;
for i = 1 : N
    n = i - 1;
    % randn(1, H) 是从标准正态分布中提取的伪随机值
    vI = (Ap + Bp * cos(2 * pi * f * t - 2 * pi * n / N)) + 1 * randn(1,
H);
    vPSin = vPSin + vI * sin(2 * pi * n / N);
    vPCos = vPCos + vI * cos(2 * pi * n / N);

end
vBc = 2 / N * sqrt(vPSin .* vPSin + vPCos .* vPCos);
vPhase = pi + atan2(-vPSin, -vPCos);%将相位提升到[0 - 2pi] 区间
figure; plot(vPhase);title('vPhase1');
vTemp = round((vTruePhase - vPhase / f) / (2 * pi / f)) ;%阶梯状
figure; plot(vTemp);title('vTemp');
vPhase = vPhase / f + vTemp * 2 * pi / f;
figure; plot(vPhase);

vErr = vPhase - vTruePhase;
vIdx = find(vErr > pi / f);
vErr(vIdx) = vErr(vIdx) - 2 * pi;
vIdx = find(vErr < -pi / f);
vErr(vIdx) = vErr(vIdx) + 2 * pi;
figure; plot(vErr);title(var(vErr));
```

通过上面的仿真程序我们可以仿真当照相机拍摄的像素存在噪声干扰时（噪声干扰是必然存在的：设想一下你投影仪投射出去的值为真值 vTruePhase，在被照相机接收时的值能完全的和vTruePhase 相等吗？肯定是会或多或少有些误差吧！于是我们采用了 randn(1, H) 正态分布伪随机值来当做这个误差）

预编译技术的内存原理

在Windows程序开发时，经常要在各个文件中包含windows.h、afx.h等标准头文件，而这些文件非常的大，在编译时就非常的慢，非常耗时。为解决这个问题，已是就有了预编译头文件的技术。

所谓头文件预编译技术，就是把一个工程(Project)中常用的一些头文件(如标准头文件Windows.h、Afxwin.h等，也可以是自己定义的头文件)包含在stdafx.h中，并对stdafx.h预先编译(在所有的.cpp文件编译之前进行编译)，得到编译结果.pch文件(默认名称为ProjectName.pch)，后期该工程在编译其它.cpp文件时不再编译stdafx.h中的内容(即使include了它)，仅仅使用预编译的结果。 其中stdafx.h叫做**预编译头文件**，stdafx名称的英文全称为：Standard Application Framework Extensions，当然你也可以自己定义预编译头文件的名称，手动重命名stdafx.h，同时将上面图2和图3中对应的名称也得改过来。ProjectName.pch叫做**预编译头**。

采用预编译头技术后，可以加快编译速度，节省编译时间。因为只需要预先编译一次就可以在所有的.cpp编译时使用，不用再次编译。这样带来的一个问题就是**每一个.cpp文件的开头都要包含预编译头文件#include “stdafx.h”。**因为预编译头技术是假定预编译头中的内容会在所有.cpp文件中使用，在编译你的 .cpp 的时候，就会将预编译头中已经编译完的部分加载到内存中。

使用预编译头文件需要注意的几个要点： \1. 你编写的任何.cpp文件都必须首先包含stdafx.h。 \2. 如果你有工程文件里的大多数.cpp文件需要的.h文件，顺便将它们加在stdafx.h(后部)上，然后预编译stdafx.cpp。 \3. 由于.pch文件具有大量的符号信息，它是你的工程文件里最大的文件。