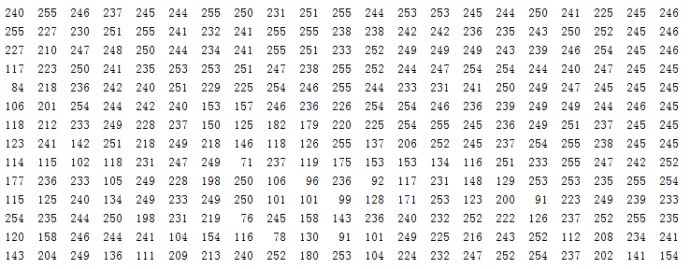
# 图像二值化

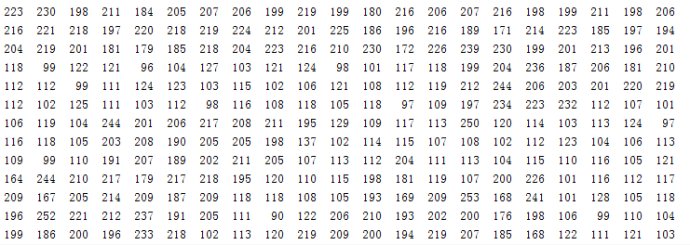
## 图像处理基本概念

像素点是最小的图像单元，一张图片由好多的像素点构成。

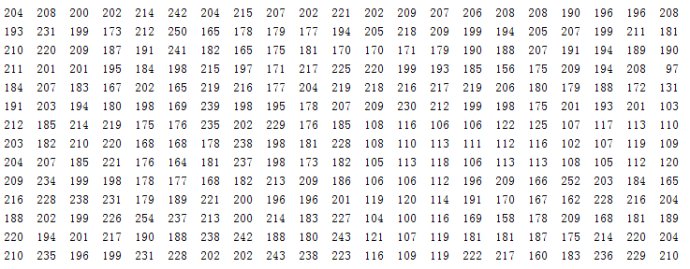
一张图片如果尺寸是800 \* 800 的，宽度是800像素，高度是800像素。也就是说这张图片是由一个800 \* 800的像素点矩阵构成的，这个矩阵是800行，800列，像素是图像的最小单元，这张图片的宽度是800个像素点的长度，高度是800个像素点的长度，共有800 \* 800 = 640000个像素点。

因为一个像素点的颜色是由RGB三个值来表现的，所以一个像素点矩阵对应三个颜色向量矩阵，分别是R矩阵，G矩阵，B矩阵，它们也都是800 \*800大小的矩阵。下面展示某图片的一部分颜色矩阵数据：

这个是R矩阵中的一部分  
[](http://photo.blog.sina.com.cn/showpic.html#blogid=13c6397540102wqtt&url=http://album.sina.com.cn/pic/005NemkAzy770eWKRak44)

这个是G矩阵中的一部分  
[](http://photo.blog.sina.com.cn/showpic.html#blogid=13c6397540102wqtt&url=http://album.sina.com.cn/pic/005NemkAzy770fs3eM323)

这个是B矩阵中的一部分

[](http://photo.blog.sina.com.cn/showpic.html#blogid=13c6397540102wqtt&url=http://album.sina.com.cn/pic/005NemkAzy770fzTLj06e)

比如每个矩阵的第一行第一列的值分别为：R：240，G：223，B：204，所以这个像素点的颜色就是（240,223,204）

存储像素值需要指定颜色空间和数据类型，彩色方式把颜色分成三种或者四种基元素，通过组合基元素可以产生所有颜色，RGB颜色空间是最常用的一种颜色空间，这归功于他也是人眼内部构成颜色的方式。他的颜色是红色，绿色，和蓝色，有时候为了表示透明颜色也会加入第四个元素alpha（A）。

### 图像中的噪声点

图像中噪声是指由于成像传感器噪声、相片颗粒噪声、图片在传输过程中的通道传输误差等因素会使图片上出现一些随机的、离散的、孤立的像素点，这就是图像噪声。图像噪声在视觉上通常与它们相邻的像素明显不同，例如黑区域中的白点、白区域中的黑点等。

## 如何改变像素点的颜色：

在理解了一张图片是由一个像素点矩阵构成之后，我们就知道我们对图像的处理就是对这个像素点矩阵的操作，想要改变某个像素点的颜色，我们只要在这个像素点矩阵中找到这个像素点的位置，比如第x行，第y列，所以这个像素点在这个像素点矩阵中的位置就可以表示成（x，y）,因为一个像素点的颜色由红、绿、蓝三个颜色变量表示，所以我们通过给这三个变量赋值，来改变这个像素点的颜色，比如改成红色（255，0，0），可以表示为（x，y，（R=255，G=0，B=0））。

### 单通道图像和三通道

（一）：单通道图，

俗称灰度图，每个像素点只能有有一个值表示颜色，它的像素值在0到255之间，0是黑色，255是白色，中间值是一些不同等级的灰色。（也有3通道的灰度图，3通道灰度图只有一个通道有值，其他两个通道的值都是零）。

（二）:三通道图，每个像素点都有3个值表示

，所以就是3通道。也有4通道的图。例如RGB图片即为三通道图片，RGB色彩模式是工业界的一种颜色标准，是通过对红(R)、绿(G)、蓝(B)三个颜色通道的变化以及它们相互之间的叠加来得到各式各样的颜色的，RGB即是代表红、绿、蓝三个通道的颜色，这个标准几乎包括了人类视力所能感知的所有颜色，是目前运用最广的颜色系统之一。总之，每一个点由三个值表示。

在我的教程 [*Mat - 基本图像容器*](http://www.opencv.org.cn/opencvdoc/2.3.2/html/doc/tutorials/core/mat%20-%20the%20basic%20image%20container/mat%20-%20the%20basic%20image%20container.html#matthebasicimagecontainer) 中，你或许已了解到，图像矩阵的大小取决于我们所用的颜色模型，确切地说，取决于所用通道数。如果是灰度图像，矩阵就会像这样：

\newcommand{\tabItG}[1] { \textcolor{black}{#1} \cellcolor[gray]{0.8}}
\begin{tabular} {ccccc}
~ & \multicolumn{1}{c}{Column 0} &   \multicolumn{1}{c}{Column 1} &   \multicolumn{1}{c}{Column ...} & \multicolumn{1}{c}{Column m}\\
Row 0 & \tabItG{0,0} & \tabItG{0,1} & \tabItG{...}  & \tabItG{0, m} \\
Row 1 & \tabItG{1,0} & \tabItG{1,1} & \tabItG{...}  & \tabItG{1, m} \\
Row ... & \tabItG{...,0} & \tabItG{...,1} & \tabItG{...} & \tabItG{..., m} \\
Row n & \tabItG{n,0} & \tabItG{n,1} & \tabItG{n,...} & \tabItG{n, m} \\
\end{tabular}

而对多通道图像来说，矩阵中的列会包含多个子列，其子列个数与通道数相等。例如，RGB颜色模型的矩阵：

\newcommand{\tabIt}[1] { \textcolor{yellow}{#1} \cellcolor{blue} &  \textcolor{black}{#1} \cellcolor{green} & \textcolor{black}{#1} \cellcolor{red}}
\begin{tabular} {ccccccccccccc}
~ & \multicolumn{3}{c}{Column 0} &   \multicolumn{3}{c}{Column 1} &   \multicolumn{3}{c}{Column ...} & \multicolumn{3}{c}{Column m}\\
Row 0 & \tabIt{0,0} & \tabIt{0,1} & \tabIt{...}  & \tabIt{0, m} \\
Row 1 & \tabIt{1,0} & \tabIt{1,1} & \tabIt{...}  & \tabIt{1, m} \\
Row ... & \tabIt{...,0} & \tabIt{...,1} & \tabIt{...} & \tabIt{..., m} \\
Row n & \tabIt{n,0} & \tabIt{n,1} & \tabIt{n,...} & \tabIt{n, m} \\
\end{tabular}

注意到，子列的通道顺序是反过来的：BGR而不是RGB。很多情况下，因为内存足够大，可实现连续存储，因此，图像中的各行就能一行一行地连接起来，形成一个长行。

## 图片的灰度化

就是让像素点矩阵中的每一个像素点都满足下面的关系：R=G=B。

### 灰度处理的方法：

一般灰度处理经常使用下面**两种计算公式**来进行处理。

方法1：

灰度化后的R=（处理前的R + 处理前的G +处理前的B）/ 3

灰度化后的G=（处理前的R + 处理前的G +处理前的B）/ 3

灰度化后的B=（处理前的R + 处理前的G +处理前的B）/ 3

方法2：

灰度化后的R = 处理前的R \* 0.3+ 处理前的G \* 0.59 +处理前的B \* 0.11

灰度化后的G = 处理前的R \* 0.3+ 处理前的G \* 0.59 +处理前的B \* 0.11

灰度化后的B = 处理前的R \* 0.3+ 处理前的G \* 0.59 +处理前的B \* 0.11

注意：第二种方法更好。

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*灰度转换函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//第一个参数image输入的彩色RGB图像；

//第二个参数imageGray是转换后输出的灰度图像；

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void ConvertRGB2GRAY(const Mat &image,Mat &imageGray)

{

if(!image.data||image.channels()!=3)

{

return ;

}

imageGray=Mat::zeros(image.size(),CV\_8UC1);

uchar \*pointImage=image.data;

uchar \*pointImageGray=imageGray.data;

int stepImage=image.step;

int stepImageGray=imageGray.step;

for(int i=0;i<imageGray.rows;i++)

{

for(int j=0;j<imageGray.cols;j++)

{

pointImageGray[i\*stepImageGray+j]=0.114\*pointImage[i\*stepImage+3\*j]+0.587\*pointImage[i\*stepImage+3\*j+1]+0.299\*pointImage[i\*stepImage+3\*j+2];

}

}

}

### 8位灰度图的解释

就是表示一个像素用了8（就是2^8=256）为二进制数，有符号最大的表示范围为-127到+128？？无符号最大表示范围为0到256，说明该图像最多有256各层次的灰度值。

举一个简单的例子，如果是2(2^2)位灰度图则最多只有0,1,2,3四种层次的灰度，如果是1(2^1)位灰度图，则只有0和1两种，即一幅图像非黑即白。

## 腐蚀，膨胀，开运算和闭运算

1. 腐蚀是一种消除边界点，使边界向内部收缩的过程。可以用来消除小且无意义的物体。

腐蚀的算法：

用3x3的结构元素，扫描图像的每一个像素用结构元素与其覆盖的二值图像做“与”操作如果都为1，结果图像的该像素为1。否则为0。结果：使二值图像减小一圈

2. 膨胀是将与物体接触的所有背景点合并到该物体中，使边界向外部扩张的过程。

可以用来填补物体中的空洞。

膨胀的算法：

用3x3的结构元素，扫描图像的每一个像素用结构元素与其覆盖的二值图像做“与”操作如果都为0，结果图像的该像素为0。否则为1结果：使二值图像扩大一圈。

3. 先腐蚀后膨胀的过程称为开运算。

用来消除小物体、在纤细点处分离物体、平滑较大物体的边界的同时并不明显改变其面积。

4. 先膨胀后腐蚀的过程称为闭运算。

用来填充物体内细小空洞、连接邻近物体、平滑其边界的同时并不明显改变其面积。

## 图片的二值化：

### 什么是阈值

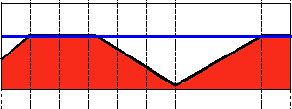
二值化就是让图像的像素点矩阵中的每个像素点的灰度值为0（黑色）或者255（白色），也就是让整个图像呈现只有黑和白的效果。

#### 截断阈值化

* 该阈值化类型如下式所示：

\texttt{dst} (x,y) =  \fork{\texttt{threshold}}{if $\texttt{src}(x,y) > \texttt{thresh}$}{\texttt{src}(x,y)}{otherwise}

* 解释：同样首先需要选定一个阈值，图像中大于该阈值的像素点被设定为该阈值，小于该阈值的保持不变。（例如：阈值选取为125，那小于125的阈值不改变，大于125的灰度值（230）的像素点就设定为该阈值）。

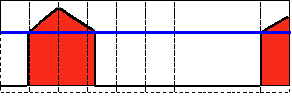


#### 阈值化为0

* 该阈值类型如下式所示：

\texttt{dst} (x,y) =  \fork{\texttt{src}(x,y)}{if $\texttt{src}(x,y) > \texttt{thresh}$}{0}{otherwise}

* 解释：先选定一个阈值，然后对图像做如下处理：1 像素点的灰度值大于该阈值的不进行任何改变；2 像素点的灰度值小于该阈值的，其灰度值全部变为0。



http://www.opencv.org.cn/opencvdoc/2.3.2/html/doc/tutorials/imgproc/threshold/threshold.html

### 二值化，灰度化区别：

在灰度化的图像中灰度值的范围为0~255，在二值化后的图像中的灰度值范围是0**或者**255。

## 二值化方法分类

可以分为： **全局二值化** 和 **局部自适应二值化**。

### Global Binariztion Method

全局二值化方法(Global Binariztion Method)对每一幅图计算一个单一的阀值。灰度级大于阈值的像素被标记为背景色，否则为前景。

### Local Adaptive Binarization Method

局部二值化方法(Local Adaptive Binarization Method)以像素的邻域的信息为基础来计算每一个像素的阈值。其中一些方法还会计算整个图像中的一个阈值面。如果图像中的一个像素(x,y)的灰度级高于在(x,y)点的阈值面的计算值，那么把像素(x,y)标记为背景，否则为前景字符。

经典的局部阈值方法包括Bernsen算法和Niblack算法。完全采用局部阈值方法进行二值化会过分夸大图像的细节而忽略了图像的整体特征易造成伪影、块效应、实现速度慢等缺点

## 常用二值化方法

### 方法一（origin）

最原始的方法，就是对RGB彩色图像灰度化以后，扫描图像的每个像素值，值小于127（相当于0~255的中数，（0+255）/2=127）的将像素值设为0(黑色)，值大于等于127的像素值设为255(白色)。该方法的优点是计算速度快。缺点也是很明显的，因为这个阀值在不同的图片中均为127，但是不同的图片，他们的颜色分布差别很大，所以用127做阀值，完全不考虑图像的像素分布情况与像素值特征。

### 方法二（imporve）

该方法则相对于第一个方法改进的地方是：计算像素点矩阵中的所有像素点的灰度值的平均值K，扫描图像的每个像素值如像素值大于K像素值设为255(白色)，值小于等于K像素值设为0(黑色)。不过该方法也是有缺点的，可能导致部分对象像素或者背景像素丢失，这使得二值化结果不能真实反映源图像信息。

### 方法三（直方图）

使用直方图方法来寻找二值化阈值，直方图是图像的重要特质，直方图方法选择二值化阈值主要是发现图像的两个最高的峰，然后在阈值取值在两个峰之间的峰谷最低处。

### 方法四：Otsu（最大类间方差法、大津算法）

matlab中的graythresh即是以该算法为原理执行的。该方法将图像分为前景和背景两部分，背景和目标之间的类间方差越大,说明构成图像的2部分的差别越大,当部分目标错分为背景或部分背景错分为目标都会导致2部分差别变小。因此,使类间方差最大的分割意味着错分概率最小。

算法步骤如下：

设图象包含L个灰度级(0,1…,L-1)，灰度值为i的的象素点数为Ni，图象总的象素点数为N=N0+N1+...+N(L-1)。灰度值为i的点的概为：

P(i)=N(i)/N.

门限t将整幅图象分为暗区c1和亮区c2两类，则类间方差σ是t的函数：

σ=a1\*a2(u1-u2)^2

式中，aj为类cj的面积与图象总面积之比，a1=sum(P(i))i->t,a2=1-a1;uj为类cj的均值，u1=sum(i\*P(i))/a10->t,

u2=sum(i\*P(i))/a2,t+1->L-1

该法选择最佳门限t^使类间方差最大，即：令Δu=u1-u2，σb=max{a1(t)\*a2(t)Δu^2}

参考：（<https://en.wikipedia.org/wiki/Otsu%27s_method>）

补充：均衡直方图二值化？？

Otsu Thresholding：

http://www.labbookpages.co.uk/software/imgProc/otsuThreshold.html

# 图像细化

骨架化：通常也将细化描述为“骨架化”、“中轴转换”和“对称轴转换”。

好的细化算法一定要满足**下面几个条件**：

1. 收敛性，保证细化后细线的连通性；
2. 保持原图的基本形状；
3. 减少笔画相交处的畸变；
4. 细化结果是原图像的中心线；
5. 细化的快速性和迭代次数少。

## 两类依据是否使用迭代运算可以分为两类：

1. **非迭代算法**：一次即产生骨架，如基于距离变换的方法，游程编码细化等。
2. **迭代算法**：即重复删除图像边缘满足一定条件的像素，最终得到单像素宽带骨架。

迭代方法依据其检查像素的方法又可以再分成**两类**：

* 串行算法：是否删除像素在每次迭代的执行中是固定顺序的，它不仅取决于前次迭代的结果，也取决于本次迭代中已处理过像素点分布情况。
* 并行算法:像素点删除与否与像素值图像中的顺序无关，仅取决于前次迭代的结果。

常用的迭代算法包括：Hilditch、Pavlidis、Rosenfeld细化算法以及基于索引表查询的细化算法等等。

1. Hilditch算法**使用于二值图像**，该算法是并行串行结合的算法。
2. Pavlidis算法通过并行和串行混合处理来实现，用位运算进行特定模式的匹配，所得的骨架是8连接的（没有理解），用于0－1二值图像。
3. Rosenfeld算法是一种并行细化算法，所得的骨架形态是8－连接的，使用于0－1二值图像。

### Hilditch细化算法的步骤为：

对图像从左向右从上向下迭代每个像素，是为一个迭代周期。在每个迭代周期中，对于每一个像素p，如果它同时满足6个条件，则标记它。在当前迭代周期结束时，则把所有标记的像素的值设为背景值。如果某次迭代周期中不存在标记点（即满足6个条件的像素），则算法结束。假设背景值为0，前景值为1，则：

6个条件为：

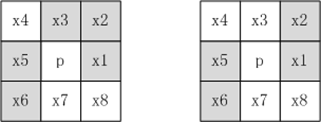
（I）：p为1，即p不是背景；

（2）：x1,x3,x5,x7不全部为1（否则把p标记删除，图像空心了）；

（3）：x1~x8中，至少有2个为1（若只有1个为1，则是线段的端点。若没有为1的，则为孤立点）；

（4）：p的8连通联结数为1；

联结数指在像素p的3\*3邻域中，和p连接的图形分量的个数：



4连通联结数计算公式是：

[image](http://images.cnblogs.com/cnblogs_com/xiaotie/WindowsLiveWriter/HilditchC_2CEF/image_12.png)

8连通联结数计算公式是：

[image](http://images.cnblogs.com/cnblogs_com/xiaotie/WindowsLiveWriter/HilditchC_2CEF/image_14.png)

其中，

[image](http://images.cnblogs.com/cnblogs_com/xiaotie/WindowsLiveWriter/HilditchC_2CEF/image_16.png)

（5）假设x3已经标记删除，那么当x3为0时，p的8联通联结数为1；

（6）假设x5已经标记删除，那么当x5为0时，p的8联通联结数为1。

### Burning算法使用迭代的方法去除图像的边界,可使用扫描线法来获取边界。设想在同一时刻，将目标的边缘线都点燃，火的前沿以匀速向内部蔓延，当前沿相交时火焰熄灭，火焰熄灭点的结合就是骨架。

### Zhang并行快速细化算法（看网上大多基于此方法）

算法介绍（算法看不懂，老师如果了解的话可以请教下）

Zhang-Suen细化算法通常是一个迭代算法，整个迭代过程分为两步：

1. 第一步：循环所有前景像素点，对符合如下条件的像素点标记为删除：

1.2<=N(p1)<=6

2.S(P1)=1

3.P2\*P4\*P6=0

4.P4\*P6\*P8=0

其中N(p1)表示跟P1相邻的8个像素点中，为前景像素点的个数

S(P1)表示从P2～P9～P2像素中出现0～1的累计次数，其中0表示背景，1表示前景

完整的P1～P9的像素位置与举例如下：

其中N(p1)=4,S(P1)=3,P2\*P4\*P6=0\*0\*0=0,P4\*P6\*P8=0\*0\*1=0,不符合条件，无需标记为删除。

1. 第二步：跟第一步很类似，条件1、2完全一致，只是条件3、4稍微不同，满足如下条件的像素P1则标记为删除，条件如下：

1.2<=N(p1)<=6

2.S(P1)=1

3.P2\*P4\*P8=0

4.P2\*P6\*P8=0

循环上述两步骤，直到两步中都没有像素被标记为删除为止，输出的结果即为二值图像细化后的骨架。

### 基于索引表的细化方法

**对于边缘点P，它的八连通域是指P点的周围的八个点，我们按如下顺序标识：**

|  |  |  |
| --- | --- | --- |
| **P0** | **P1** | **P2** |
| **P7** | **P** | **P3** |
| **P6** | **P5** | **P4** |

八邻域的数字一共有2^8次方种情况。

首先把256种状态和能不能删除的标记已经对应起来。

从p0开始，顺时针旋转一直到p7组成的二进制数

如：

000

1p1

111

组成的二进制数就是：0001111，则对应的十进制数就是(二进制转十进制):248。然后找到二进制和对应的删除标记是0还是1。

对于这八个点的不同取值情况，按如下顺序表示的二进制的值也将有不同的取值

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P7 | P6 | P5 | P4 | P3 | P2 | P1 | P0 | Sum | DeleteEnable |
|  |  |  |  |  |  |  |  |  |  |

其中Sum的取值从0~255，DeleteEnable为1表示可以删除，为0表示不能删除，这样我们就建立了一个索引表deletemark[256]，

1. deletemark[256]={
2. 0,0,0,0,0,0,0,1,0,0,1,1,0,0,1,1,
3. 0,0,0,0,0,0,0,0,0,0,1,1,1,0,1,1,
4. 0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,1,
5. 0,0,0,0,0,0,0,0,1,0,1,1,1,0,1,1,
6. 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
7. 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
8. 0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,1,
9. 1,0,0,0,0,0,0,0,1,0,1,1,1,0,1,1,
10. 0,0,1,1,0,0,1,1,0,0,0,1,0,0,1,1,
11. 0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,1,
12. 1,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,
13. 1,1,0,1,0,0,0,1,1,1,0,0,1,0,0,0,
14. 0,1,1,1,0,0,1,1,0,0,0,1,0,0,1,1,
15. 0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,
16. 1,1,1,1,0,0,1,1,1,1,0,0,1,1,0,0,
17. 1,1,1,1,0,0,1,1,1,1,0,0,1,1,0,0
18. };//索引表

# 轮廓提取

1. 基于边缘的方法

分为：基于边缘检测和基于边缘分组

基于边缘检测：边缘检测算子+去除杂点、冗余边缘、修复边缘（噪声敏感）（基于梯度的方法可以归纳到这里）

基于边缘分组：线逼近算法去除噪声+边缘点逐步组合合并成（噪声敏感）

1. Opencv中提供了很多关于图像轮廓处理的函数

这里我用cvFindContours函数来提取轮廓，首先将图像二值化，因为这个图像必须是一个二值图。得到二值图的方法有很多，这里我采用的是Opencv中的cvThreshold函数，就不需要自己写二值化的相关函数，通过设置阀值来得到相应的二值图。最后用cvDrawContours函数将提取的轮廓画出来。函数cvFindContours的第一个参数就是要进行提取轮廓的目标图像。

## Canny

John Canny研究了最优边缘检测方法所需的特性，给出了评价边缘检测性能优劣的3个指标：

* 1. 好的信噪比，即将非边缘点判定为边缘点的概率要低，将边缘点判为非边缘点的概率要低；
  2. 高的定位性能，即检测出的边缘点要尽可能在实际边缘的中心；
  3. 对单一边缘仅有唯一响应，即单个边缘产生多个响应的概率要低，并且虚假响应边缘应该得到最大抑制；

Canny算法就是基于满足这3个指标的最优解实现的，在对图像中物体边缘敏感性的同时，也可以抑制或消除噪声的影响。

#### Canny算子边缘检测的具体步骤如下：

#### http://blog.csdn.net/dcrmg/article/details/52344902

1. 用高斯滤波器平滑图像
2. 用Sobel等梯度算子计算梯度幅值和方向
3. 对梯度幅值进行非极大值抑制
4. 用双阈值算法检测和连接边缘

# 高斯滤波器

高斯滤波是一种线性平滑滤波，适用于消除高斯噪声，特别是对抑制或消除服从正态分布的噪声非常有效。滤波可以消除或降低图像中噪声的影响，使用高斯滤波器主要是基于在滤波降噪的同时也可以最大限度保留边缘信息的考虑。

# 二维贝塞尔曲线

动态绘制贝塞尔曲线的[在线演示](http://myst729.github.io/bezier-curve).

http://www.html-js.com/article/1628

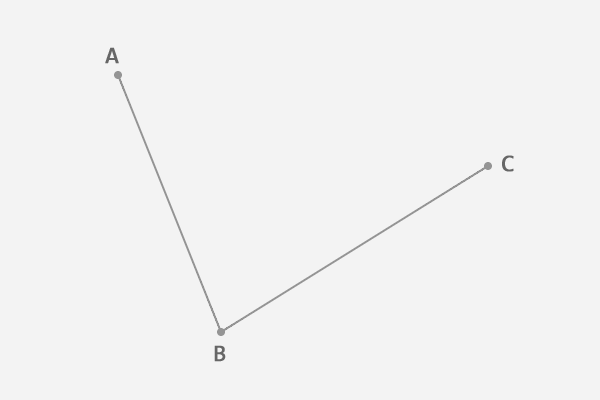
贝塞尔曲线的数学基础是早在 1912 年就广为人知的伯恩斯坦多项式。但直到 1959 年，当时就职于雪铁龙的法国数学家 Paul de Casteljau 才开始对它进行图形化应用的尝试，并提出了一种数值稳定的 de Casteljau 算法。然而贝塞尔曲线的得名，却是由于 1962 年另一位就职于雷诺的法国工程师 Pierre Bézier 的广泛宣传。

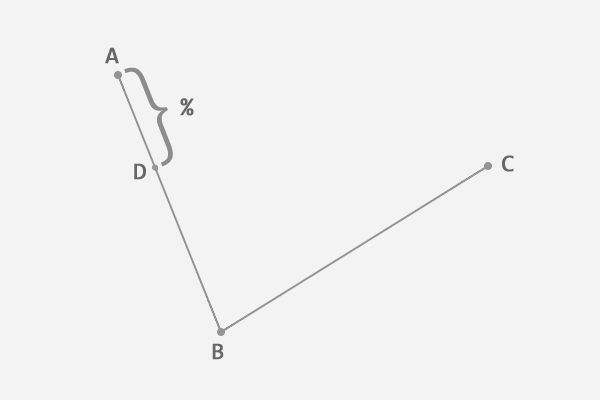
正是因为控制简便却具有极强的描述能力，贝塞尔曲线在工业设计领域迅速得到了广泛的应用。不仅如此，在计算机图形学领域，尤其是矢量图形学，贝塞尔曲线也占有重要的地位。今天我们最常见的一些矢量绘图软件，如 Flash、Illustrator、CorelDraw 等，无一例外都提供了绘制贝塞尔曲线的功能。Photoshop 这样的位图编辑软件，也把贝塞尔曲线作为仅有的矢量绘制工具（钢笔工具）包含其中。

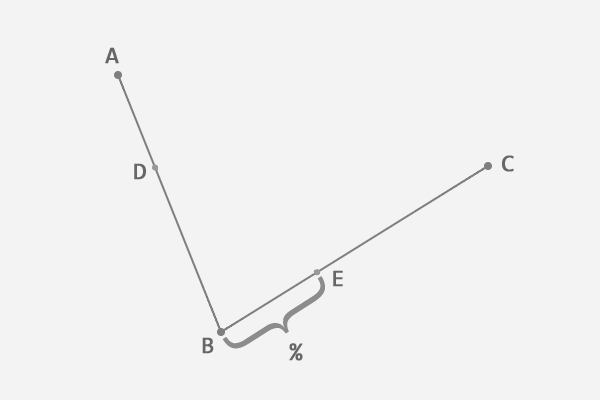
贝塞尔曲线在 web 开发领域同样占有一席之地。CSS3 新增了 transition-timing-function 属性，它的取值就可以设置为一个三次贝塞尔曲线方程。在此之前，也有不少 JavaScript 动画库使用贝塞尔曲线来实现美观逼真的缓动效果。

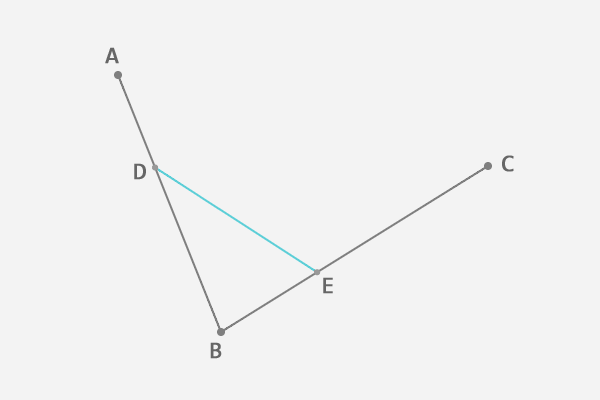
下面我们就通过例子来了解一下如何用 de Casteljau 算法绘制一条贝塞尔曲线。

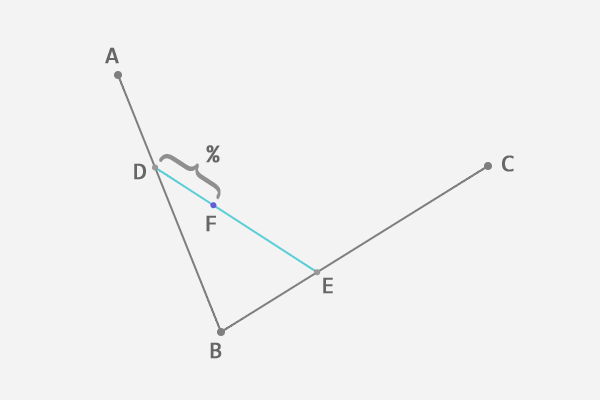
在平面内任选 3 个不共线的点，依次用线段连接。

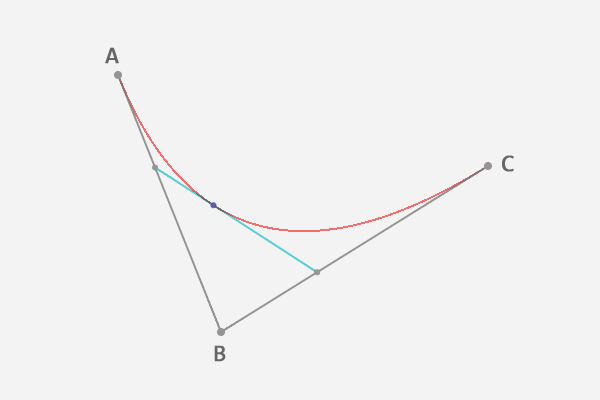


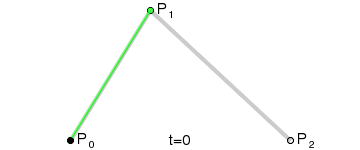
在第一条线段上任选一个点 D。计算该点到线段起点的距离 AD，与该线段总长 AB 的比例。

根据上一步得到的比例，从第二条线段上找出对应的点 E，使得 AD:AB = BE:BC。

连接这两点 DE。

从新的线段 DE 上再次找出相同比例的点 F，使得 DF:DE = AD:AB = BE:BC。

到这里，我们就确定了贝塞尔曲线上的一个点 F。接下来，请稍微回想一下中学所学的极限知识，让选取的点 D 在第一条线段上从起点 A 移动到终点 B，找出所有的贝塞尔曲线上的点 F。所有的点找出来之后，我们也得到了这条贝塞尔曲线。

如果你实在想象不出这个过程，没关系，看动画！

回过头来看这条贝塞尔曲线，为了确定曲线上的一个点，需要进行两轮取点的操作，因此我们称得到的贝塞尔曲线为二次曲线（这样记忆很直观，但曲线的次数其实是由前面提到的伯恩斯坦多项式决定的）。

# 如何遍历图像中的每一个像素？

目前，OpenCV主要有三种逐像素遍历图像的方法。我们将分别用这三种方法扫描图像，并将它们所用时间输出到屏幕上。我想这样的对比应该很有意思。