GÖRÜNTÜ İŞLEME - (7.Hafta)

KENAR BULMA ALGORİTMALARI

Bu konuda bir çok algoritma olmasına rağmen en yaygın kullanılan ve etkili olan Sobel algoritması burada anlatılacaktır.

Sobel KenarBulma Algoritması

Görüntüyü siyah beyaza çevirdikten sonra eğer kenar bulma algoritmalarını kullanmak isterseniz bir kaç seçenekten en popüleri sobel kenar bulma filtresidir. Aşağıdaki çekirdek matrisler (konvolüsyon matrisleri) dikey, yatay ve köşegen şeklindeki kenarları bulmak için kullanılır. Sobel operatörü bir resmin kenarlarına karşılık gelen alansal yüksek frekans bölgelerini (keskin kenarları) ortaya çıkarır. Teorik olarak, operatör aşağıda gösterildiği gibi 3×3 konvolüsyon matrisinden oluşur.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Şekil. Sobel konvolüsyon matrisi

Bu matrisler, yatay ve dikey olarak görünen kenarlar için ayrı ayrı olacak şekilde düzenlenmiştir. Matrisler giriş görüntüsüne birbirinden bağımsız uygulanabilir. Böylece her bir yön için pikselin değeri ayrı ayrı ölçülmüş olur. Daha sonra çapraz duran kenarların mutlak değerini bulmak ve yönünü (açısını) bulmak için bu değerler aşağıdaki formüllerle birleştirilebilir. Piksel mutlak değeri şu şekilde hesaplanabilir.

$$|G| = \sqrt{Gx^2 + Gy^2}$$

Piksel değerini daha hızlı hesaplama için şu formülde kullanılabilir.

$$|G| = |Gx| + |Gy|$$

Ortaaya çıkan kenarın yön açısı (piksel ızgarasına göre) x ve y yönlerindeki değerlerine bakarak şu şekilde bulunur.

$$\theta = \arctan\left(\frac{|Gx|}{|Gy|}\right)$$

(Gx dikey çizgiyi bulur, Gy yatay çizgiyi bulur. Tan karşı kenarın komşu kenara oranı olduğu için formül bu şekilde yazıldı)

Bu durumda 0 derece yatay duran çizgileri, 90 derece ise dikey duran çizgileri gösterecektir. Eğik duran çizgilerde diğer açıları oluşturacaktır. Burada sıfır derece çizgide alt kısım siyahtan, üst kısım beyaza doğru geçişi gösterir. 0 derecenin bir benzeri olan yine yatay duran çizgi 180 derecede ise üst kısım siyah bölgeden alt kısım beyaz bölgeye doğru bir geçişi gösterecektir. Açılar saatin tersi yönüne göre ölçülür.

Burada Gx konvolüsyon matrisi tek başına kullanılırsa yatayda renk değişimini bulacağından, ortaya çıkan çizgileri dikey olarak görürüz. Benzer şekilde Gy konvolüsyon matrisi de tek başına kullanılırsa aşağıdan yukarıya doğru renk geçişini (siyahtan-beyaza) göstereceğinden ortaya çıkan çizgiler yatay olacaktır. Resim üzerindeki çizgileri yatay yada dikey görmek yerine kendi doğal duruşlarını görmek istersek, her iki konvolüsyon matrisini yukarıda formülü verilen hesaplama ile toplayabiliriz. Benzer şekilde yukarıda verilen karaköklü formülü de kullanabiliriz. Eğer resim üzerinde belli açılardaki çizgileri ortaya çıkarmak istersek verilen açı formülü ile hesaplama yaparak bu çizgileri de belirleyebiliriz.

İki matrisi aynı anda kullanmak için ve matris üzerindeki noktaları aşağıdaki şekilde temsil etmek için hesaplama yaparak deneyelim.

P_1	P_2	P_3
P_4	P_5	P_6
P_7	P_8	P_9

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Buradaki her bir çekirdek matrisin hesabı şu şekilde hesaplanır.

$$|Gx| = |-P_1 + P_3 - 2P_4 + 2P_6 - P_7 + P_9|$$

$$|Gy| = |P_1 + 2P_2 + P_3 - P_7 - 2P_8 - P_9|$$

Bu çekirdek matris kullanılarak pikselin sobel değeri yaklaşık formül kullanılarak şu şekilde hesaplanır:

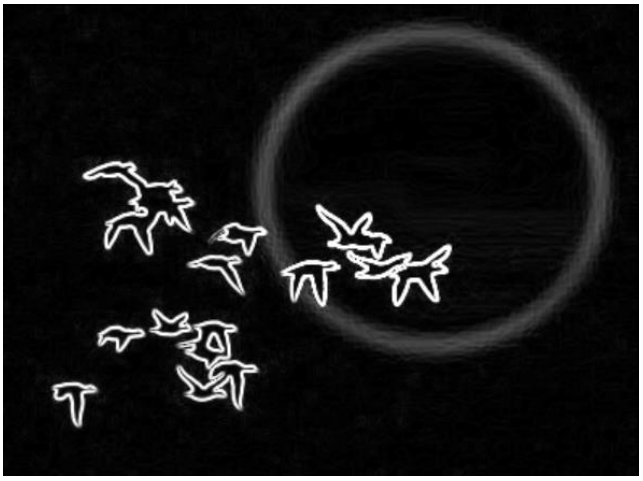
$$|G| = |Gx| + |Gy|$$

Diğer formül kullanıldığında ise şu şekilde olacaktır.

$$|G| = \sqrt{Gx^2 + Gy^2}$$

Burada renkli resimde renk değerleri 3 kanal olduğunda tek kanal üzerinden işlem yapmak için Gri renk değerleri üzerinden işlem yapmak gerekir. Aşağıdaki programda Gri renk için 3 kanalın ortalması alınmıştır. Gerekirse Gri renk formülleri de kullanılabilir (daha önceki konularda geçti).

Programlama (Sobel Filtresi)



1. Kodlama

```
private void mnuSobel_Click(object sender, EventArgs e)
{
    Bitmap GirisResmi, CikisResmiXY, CikisResmiX, CikisResmiY;
    GirisResmi = new Bitmap(pictureBox1.Image);

    int ResimGenisligi = GirisResmi.Width;
    int ResimYuksekligi = GirisResmi.Height;

    CikisResmiX = new Bitmap(ResimGenisligi, ResimYuksekligi);
    CikisResmiY = new Bitmap(ResimGenisligi, ResimYuksekligi);
    CikisResmiXY = new Bitmap(ResimGenisligi, ResimYuksekligi);
```

```
    int SablonBoyutu = 3;
    int ElemanSayisi = SablonBoyutu * SablonBoyutu;

    int x, y;

    Color Renk;
    int P1, P2, P3, P4, P5, P6, P7, P8, P9;

    for (x = (SablonBoyutu - 1) / 2; x < ResimGenisligi - (SablonBoyutu - 1) / 2; x++) //Resmi
taramaya şablonun yarısı kadar dış kenarlardan içeride başlayacak ve bitirecek.
    {
        for (y = (SablonBoyutu - 1) / 2; y < ResimYuksekligi - (SablonBoyutu - 1) / 2; y++)
        {

            Renk = GirisResmi.GetPixel(x - 1, y - 1);
            P1 = (Renk.R + Renk.G + Renk.B) / 3;

            Renk = GirisResmi.GetPixel(x, y - 1);
            P2 = (Renk.R + Renk.G + Renk.B) / 3;

            Renk = GirisResmi.GetPixel(x + 1, y - 1);
            P3 = (Renk.R + Renk.G + Renk.B) / 3;

            Renk = GirisResmi.GetPixel(x - 1, y);
            P4 = (Renk.R + Renk.G + Renk.B) / 3;

            Renk = GirisResmi.GetPixel(x, y);
            P5 = (Renk.R + Renk.G + Renk.B) / 3;

            Renk = GirisResmi.GetPixel(x + 1, y);
            P6 = (Renk.R + Renk.G + Renk.B) / 3;

            Renk = GirisResmi.GetPixel(x - 1, y + 1);
            P7 = (Renk.R + Renk.G + Renk.B) / 3;

            Renk = GirisResmi.GetPixel(x, y + 1);
            P8 = (Renk.R + Renk.G + Renk.B) / 3;

            Renk = GirisResmi.GetPixel(x + 1, y + 1);
            P9 = (Renk.R + Renk.G + Renk.B) / 3;

            //Hesaplamayı yapan Sobel Temsili matrisi ve formülü.
            int Gx = Math.Abs(-P1 + P3 - 2 * P4 + 2 * P6 - P7 + P9); //Dikey çizgiler
            int Gy = Math.Abs(P1 + 2 * P2 + P3 - P7 - 2 * P8 - P9);  //Yatay Çizgiler

            //if (Gx > 100)
            //    Gx = 255;
            //else
            //    Gx = 0;

            //if (Gy > 100)
            //    Gy = 255;
            //else
            //    Gy = 0;

            int Gxy = Gx + Gy;

            //if (Gxy > Esikleme)
            //    Gxy = 255;
            //else
            //    Gxy = 0;

            //Renkler sınırların dışına çıktıysa, sınır değer alınacak. Negatif olamaz,
formüllerde mutlak değer vardır.
            if (Gx > 255) Gx = 255;
            if (Gy > 255) Gy = 255;
```

```
            if (Gxy > 255) Gxy = 255;

            //int TetaRadyan = 0;
            //if (Gy != 0)
            //    TetaRadyan = Convert.ToInt32(Math.Atan(Gx / Gy));
            //else
            //    TetaRadyan = Convert.ToInt32(Math.Atan(Gx));


            //int TetaDerece = Convert.ToInt32((TetaRadyan * 360) / (2 * Math.PI));

            //if (TetaDerece >= 0 && TetaDerece < 45)
            //    CikisResmiXY.SetPixel(x, y, Color.FromArgb(0, 0, 0));

            //if (TetaDerece >= 45 && TetaDerece < 90)
            //    CikisResmiXY.SetPixel(x, y, Color.FromArgb(0, 255, 0));

            //if (TetaDerece >= 90 && TetaDerece < 135)
            //    CikisResmiXY.SetPixel(x, y, Color.FromArgb(0, 0, 255));

            //if (TetaDerece >= 135 && TetaDerece < 180)
            //    CikisResmiXY.SetPixel(x, y, Color.FromArgb(255, 255, 0));


            CikisResmiX.SetPixel(x, y, Color.FromArgb(Gx, Gx, Gx));
            CikisResmiY.SetPixel(x, y, Color.FromArgb(Gy, Gy, Gy));

            CikisResmiXY.SetPixel(x, y, Color.FromArgb(Gxy, Gxy, Gxy));

        }
    }
    pictureBox2.Image = CikisResmiXY;
    pictureBox3.Image = CikisResmiX;
    pictureBox4.Image = CikisResmiY;
}
```

2. Kodlama

```
private void SOBEL_KENAR2_Click(object sender, EventArgs e)
{
    Color OkunanRenk;
    Bitmap GirisResmi, CikisResmiX, CikisResmiY, CikisResmiXY;
    GirisResmi = new Bitmap(pictureBox1.Image);

    int ResimGenisligi = GirisResmi.Width;
    int ResimYuksekligi = GirisResmi.Height;

    CikisResmiX = new Bitmap(ResimGenisligi, ResimYuksekligi);
    CikisResmiY = new Bitmap(ResimGenisligi, ResimYuksekligi);
    CikisResmiXY = new Bitmap(ResimGenisligi, ResimYuksekligi);

    int SablonBoyutu = 3;
    int ElemanSayisi = SablonBoyutu * SablonBoyutu;


    int x, y, i, j;
    int Gri = 0;

    int[] MatrisX = { -1, 0, 1, -2, 0, 2, -1, 0, 1 };

    int[] MatrisY = { 1, 2, 1, 0, 0, 0, -1, -2, -1 };

    int RenkX, RenkY, RenkXY;
```

```
    for (x = (SablonBoyutu - 1) / 2; x < ResimGenisligi - (SablonBoyutu - 1) / 2; x++) //Resmi
taramaya şablonun yarısı kadar dış kenarlardan içeride başlayacak ve bitirecek.
    {
        for (y = (SablonBoyutu - 1) / 2; y < ResimYuksekligi - (SablonBoyutu - 1) / 2; y++)
        {
            int toplamGriX = 0, toplamGriY = 0;

            //Şablon bölgesi (çekirdek matris) içindeki pikselleri tarıyor.
            int k = 0; //matris içindeki elemanları sırayla okurken kullanılacak.

            for (i = -((SablonBoyutu - 1) / 2); i <= (SablonBoyutu - 1) / 2; i++)
            {
                for (j = -((SablonBoyutu - 1) / 2); j <= (SablonBoyutu - 1) / 2; j++)
                {
                    OkunanRenk = GirisResmi.GetPixel(x + i, y + j);

                    Gri = (OkunanRenk.R + OkunanRenk.G + OkunanRenk.B) / 3;

                    toplamGriX = toplamGriX + Gri * MatrisX[k];

                    toplamGriY = toplamGriY + Gri * MatrisY[k];

                    k++;
                }

            }

            RenkX = Math.Abs(toplamGriX);
            RenkY = Math.Abs(toplamGriY);
            RenkXY = Math.Abs(toplamGriX) + Math.Abs(toplamGriY);

            //============================================================
            //Renkler sınırların dışına çıktıysa, sınır değer alınacak.
            if (RenkX > 255) RenkX = 255;
            if (RenkY > 255) RenkY = 255;
            if (RenkXY > 255) RenkXY = 255;


            if (RenkX < 0) RenkX = 0;
            if (RenkY < 0) RenkY = 0;
            if (RenkXY < 0) RenkXY = 0;

            //============================================================

            CikisResmiX.SetPixel(x, y, Color.FromArgb(RenkX, RenkX, RenkX));
            CikisResmiY.SetPixel(x, y, Color.FromArgb(RenkY, RenkY, RenkY));
            CikisResmiXY.SetPixel(x, y, Color.FromArgb(RenkXY, RenkXY, RenkXY));
        }
    }
    pictureBox2.Image = CikisResmiX;
    pictureBox3.Image = CikisResmiY;
    pictureBox4.Image = CikisResmiXY;
}
```

Prewitt Kenar Bulma Algoritması

Bu algoritma Sobel'e benzer ve aşağıdaki gibi biraz farklı çekirdek matris kullanır. Elde edilen sonuçlar resmin her yerinde aynı değildir.

-1	0	+1
-1	0	+1
-1	0	+1

Gx

+1	+1	+1
0	0	0
-1	-1	-1

Gy

Prewitt Kenar Bulma algoritmasının çekirdek matrisleri

Ödev: Bu algoritmalar Gri renk için ortalama formülü kullanılmıştır. Gri tonlama için daha gelişmiş olan formülleri kullanarak deneyin.

$$|G| = |Gx| + |Gy|$$

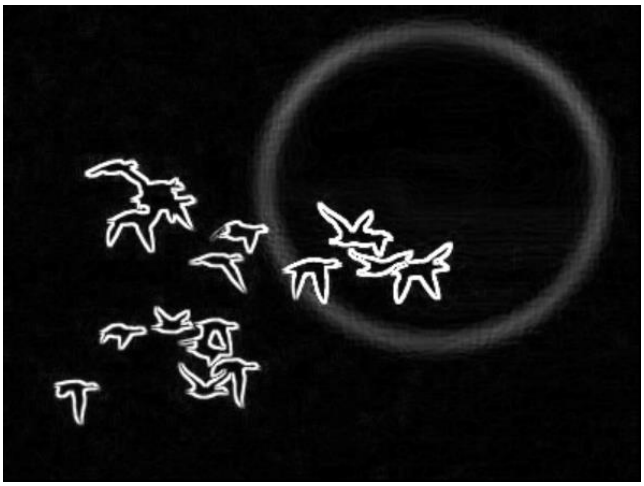
Piksel değerini daha hızlı hesaplama için şu formülde kullanılabilir.

$$|G| = \sqrt{Gx^2 + Gy^2}$$

Ortaaya çıkan kenarın yön açısı (piksel ızgarasına göre) x ve y yönlerindeki değerlerine bakarak şu şekilde bulunur.

$$\theta = \arctan\left(\frac{|Gx|}{|Gy|}\right)$$

Programlama (Prewitt Algoritması)



```
private void mnuPrewitt_Click(object sender, EventArgs e)
   {
   Bitmap GirisResmi, CikisResmi;
   GirisResmi = new Bitmap(pictureBox1.Image);

   int ResimGenisligi = GirisResmi.Width;
   int ResimYuksekligi = GirisResmi.Height;

   CikisResmi = new Bitmap(ResimGenisligi, ResimYuksekligi);
```

```
    int SablonBoyutu = 3;
    int ElemanSayisi = SablonBoyutu * SablonBoyutu;

    int x, y;

    Color Renk;
    int P1, P2, P3, P4, P5, P6, P7, P8, P9;

    for (x = (SablonBoyutu - 1) / 2; x < ResimGenisligi - (SablonBoyutu - 1) / 2; x++) //Resmi
taramaya şablonun yarısı kadar dış kenarlardan içeride başlayacak ve bitirecek.
    {
    for (y = (SablonBoyutu - 1) / 2; y < ResimYuksekligi - (SablonBoyutu - 1) / 2; y++)
    {

    Renk = GirisResmi.GetPixel(x - 1, y - 1);
    P1 = (Renk.R + Renk.G + Renk.B) / 3;

    Renk = GirisResmi.GetPixel(x, y - 1);
    P2 = (Renk.R + Renk.G + Renk.B) / 3;

    Renk = GirisResmi.GetPixel(x + 1, y - 1);
    P3 = (Renk.R + Renk.G + Renk.B) / 3;

    Renk = GirisResmi.GetPixel(x - 1, y);
    P4 = (Renk.R + Renk.G + Renk.B) / 3;

    Renk = GirisResmi.GetPixel(x, y);
    P5 = (Renk.R + Renk.G + Renk.B) / 3;

    Renk = GirisResmi.GetPixel(x + 1, y);
    P6 = (Renk.R + Renk.G + Renk.B) / 3;

    Renk = GirisResmi.GetPixel(x - 1, y + 1);
    P7 = (Renk.R + Renk.G + Renk.B) / 3;

    Renk = GirisResmi.GetPixel(x, y + 1);
    P8 = (Renk.R + Renk.G + Renk.B) / 3;

    Renk = GirisResmi.GetPixel(x + 1, y + 1);
    P9 = (Renk.R + Renk.G + Renk.B) / 3;

    int Gx = Math.Abs(-P1 + P3 - P4 + P6 - P7 + P9); //Dikey çizgileri Bulur
    int Gy = Math.Abs(P1 + P2 + P3 - P7 - P8 - P9);  //Yatay Çizgileri Bulur.

    int PrewittDegeri = 0;
    PrewittDegeri = Gx;
    PrewittDegeri = Gy;

    PrewittDegeri = Gx + Gy;  //1. Formül
    //PrewittDegeri = Convert.ToInt16(Math.Sqrt(Gx * Gx + Gy * Gy));  //2.Formül


    //Renkler sınırların dışına çıktıysa, sınır değer alınacak.
    if (PrewittDegeri > 255) PrewittDegeri = 255;

    //Eşikleme: Örnek olarak 100 değeri kullanıldı.
    //if (PrewittDegeri > 100)
    //PrewittDegeri = 255;
    //else
    //PrewittDegeri = 0;

    CikisResmi.SetPixel(x, y, Color.FromArgb(PrewittDegeri, PrewittDegeri, PrewittDegeri));

    }
    }
    pictureBox2.Image = CikisResmi;
```

```
}
```

Robert Cross Kenar Bulma Algoritması

Basit ve hızlı bir algoritmadır. Resim üzerindeki 2 boyutlu geçişleri ölçer. Keskin kenarları ortaya çıkarır. Gri resim üzerinde işlem yapar. 2x2 lik matris kullandığından çok bulanık resimlerde kenarları bulamaz. Bulmuş olduğu kenarlarıda çok ince olarak gösterir.

Aşağıdaki 2x2 lik aralarında 90 açı bulunan iki matrisle işlemleri yürütür. Kullanım olarak Sobel'e benzer.

+1	0
0	-1

Gx

0	+1
-1	0

Gy

P_1	P_2
P_3	P_4

Robert Cross konvolisyon matrisleri

Sobel'de matrislerin yönleri x ve y eksenleri yönünde idi. Burada ise Resmin ızgarasına 45 açıyla durmaktadır. Kullanılacak formüller burada yine Gx ve Gy şeklinde gösterilmiştir fakat bu yönlerin ızgaraya 45 ve 135 derece ile durduğunu kabul edelim.

Bunun için aşağıdaki iki formülü kullanabiliriz. İkinci formül daha hızlı çalışır.

$$|Gx| = |P_1 - P_4|$$
$$|Gy| = |P_2 - P_3|$$

Olmak üzere

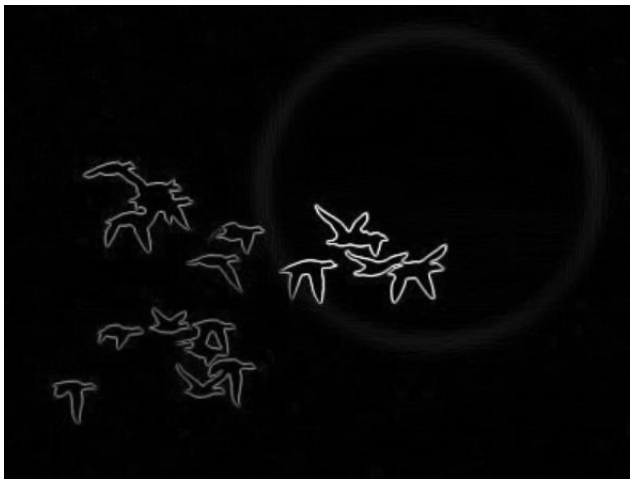
$$|G| = |Gx| + |Gy|$$
$$|G| = \sqrt{Gx^2 + Gy^2}$$

Formülleri kullanılabilir. Oluşan kenarın resim ızgarasına olan açısı aşağıdaki formülle bulunabilir (Dikkat formülü deneyerek teyid edin!).

$$\theta = \arctan\left(\frac{Gy}{Gx}\right) + 45$$

Bu operatörde 4 piksel olduğundan çıkış pikselin hangisine denk geldiği belirsizdir. Aslında yarım piksellik bir kaymadan söz edilebilir. Programlarken referans pikseli sol üst köşedeki ilk piksel alınmıştır.

Programlama (Robert Cross)



Dikkat bu uygulamada zeminle öndeki nesne arasındaki fark arttıkça kenarlar daha belirgin olmaktadır. Sebebini yorumlamaya çalışın.

```
private void mnuRobertCross_Click(object sender, EventArgs e)
{
    Bitmap GirisResmi, CikisResmi;
    GirisResmi = new Bitmap(pictureBox1.Image);

    int ResimGenisligi = GirisResmi.Width;
    int ResimYuksekligi = GirisResmi.Height;

    CikisResmi = new Bitmap(ResimGenisligi, ResimYuksekligi);

    int x, y;

    Color Renk;
    int P1, P2, P3, P4;

    for (x = 0; x < ResimGenisligi - 1; x++) //Resmi taramaya şablonun yarısı kadar dış
kenarlardan içeride başlayacak ve bitirecek.
    {
      for (y = 0; y < ResimYuksekligi - 1; y++)
      {

         Renk = GirisResmi.GetPixel(x , y);
         P1 = (Renk.R + Renk.G + Renk.B) / 3;

         Renk = GirisResmi.GetPixel(x +1, y);
         P2 = (Renk.R + Renk.G + Renk.B) / 3;

         Renk = GirisResmi.GetPixel(x , y + 1);
         P3 = (Renk.R + Renk.G + Renk.B) / 3;

         Renk = GirisResmi.GetPixel(x + 1, y + 1);
         P4 = (Renk.R + Renk.G + Renk.B) / 3;

         int Gx = Math.Abs(P1 - P4 ); //45 derece açı ile duran çizgileri bulur.
         int Gy = Math.Abs(P2 - P3); //135 derece açı ile duran çizgileri bulur.

         int RobertCrossDegeri = 0;
         RobertCrossDegeri = Gx;
         RobertCrossDegeri = Gy;

         RobertCrossDegeri = Gx + Gy;  //1. Formül
         //RobertCrossDegeri = Convert.ToInt16(Math.Sqrt(Gx * Gx + Gy * Gy));  //2.Formül

         //Renkler sınırların dışına çıktıysa, sınır değer alınacak.
```

```
            if (RobertCrossDegeri > 255) RobertCrossDegeri = 255; //Mutlak değer kullanıldığı için
negatif değerler oluşmaz.

            //Eşikleme
            //if (RobertCrossDegeri > 50)
            //    RobertCrossDegeri = 255;
            //else
            //    RobertCrossDegeri = 0;

            CikisResmi.SetPixel(x, y, Color.FromArgb(RobertCrossDegeri, RobertCrossDegeri,
RobertCrossDegeri));

        }
    }
    pictureBox2.Image = CikisResmi;

}
```

Compass Kenar Bulma Algoritması

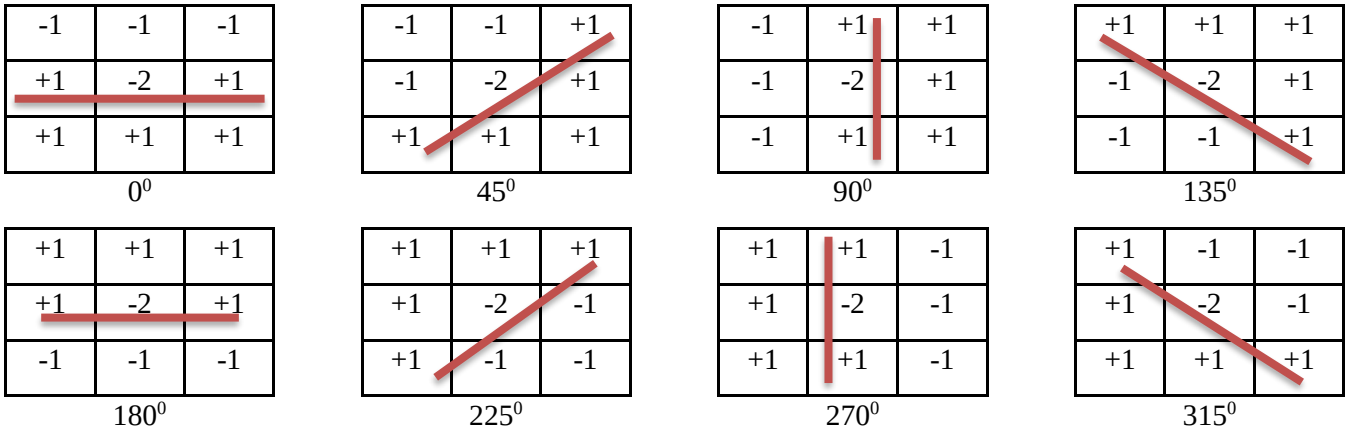
Compass algoritması da Sobel ve Robert Cross gibi kenarların her iki tarafındaki renk geçiş farkını kullanan alternatif bir algoritmadır.

Bu algoritma 8 farklı yönü verecek şekilde çekirdek matrisi döndürüp en yüksek piksel değerini veren matris ile o pikselin değerini oluşturur. Böylece kenarın ilerleme yönünü en iyi bulan matrisi deneyerek kullanmış olmaktadır. Hangi açıdaki matrisin kullanıldığı ise o kenara ait ilerleme yönünü de vermiş olmaktadır.

Önceki algoritmalar Gx ve Gy şeklinde birbirine dik iki yönü kullanırken ve ara açıları bu iki yönün birleşimi ile bulurken, burada 8 adet yönün kullanılması (G_0, G_{45}, $G9_0$, G_{135}, G_{180}, G_{225}, G_{270}, G_{315}) kenarların yönünü daha hassas olarak vermekte ve o yöndeki kenarlar daha belirgin gösterilmektedir. Önceki Gx ve Gy algoritmaları bir 360 derecelik daireyi ancak dik kenarların geçtiği kısımlarda daha belirgin gösterirken, bu algoritma 45 derecelik açılar halinde yönleri bulduğu için daire tüm yönlerde çok daha belirgin olarak gözükecektir. Algoritmanın işlem süresi 8 adet matrisi resim üzerinde gezdirdiği için daha yavaştır. Algoritmanın matematiksel anlamış şu şekilde gösterilebilir.

$$|G| = \max\left(|G_i| : i = 1 \; to \; n\right)$$

Burada n adet çekirdek matris için hesaplanan $|G_i|$ değerleri içinde en büyük değeri veren matris o pikselin değerini oluşturacaktır. Kullanılan 8 adet 45 açılarla duran matrisler şu şekilde belirlenebilir.



Buradaki kullanılan Compass algoritması olarak kullanılan çekirdek matrisin haricinde aşağıda 2 açıdaki örnekleri verilen algoritmalarda Compass mantığı ile kullanılabilir. Hatırlanırsa Sobel algoritması yukarıda 90^0 ve 0^0 derece olarak kullanılmıştı. Burada verilen 135^0 açı değerinde ve yukarıdaki örnek uygulamalarda nasıl yapıldığı

incelenerek diğer 8 adet açıdaki matrisleri oluşturulabilir. Bunları kendiniz oluşturunuz. Ayrıca bu örneklerden Kirsch ve Robinson matrislerini yukarıda Sobel konusunda verildiği şekilde Gx ve Gy mantığı ile kenar bulma işlemleri için deneyin. Sobel, Kirsch ve Robinson algoritmalarını iki eksende kullanıldığında hangisi daha iyi sonuç vermektedir gözlemleyin.

	0^0				135^0		
Sobel	-1	0	1		0	1	2
	-2	0	2		-1	0	1
	-1	0	1		-2	-1	0
Kirsch	-3	-3	5		-3	5	5
	-3	0	5		-3	0	5
	-3	-3	5		-3	-3	-3
Robinson	-1	0	1		0	1	1
	-1	0	1		-1	0	1
	-1	0	1		-1	-1	0

Dikkat! bu algoritmanın programlaması Ödevlerde istendiğinden buraya konulmamıştır. Diğer programlama örneklerine bakarak benzer şekilde programlanabilir. Fakat burada 8 tane matris olduğundan her matrisi yukarıdaki örneklerde verilen şekilde programlamak kodları uzatır. O nedenle matris değerlerini çift boyutlu bir dizide tutarak programlamak gerekir. Örneğin C# daki List (Listeler dizisi) kullanarak yapabilirsiniz. Yada 2 boyutlu bir dizi tutarak da olabilir.

Aşındırma ve Genişletme Yöntemi ile Kenar Belirleme İşlemi

Bu yöntem bir sonraki notlarda Aşındırma ve Genişletme yöntemi anlatıldığından, oradaki notların sonuna konulmuştur.

Orijinal Resim ile Bulanık Resim Farkından Kenar Belirleme İşlemi

Bu konu iki önce notlar içinde işlenmişti.

Orijinal Resim ile Netleştirilmiş Resim Farkından Kenar Belirleme İşlemi

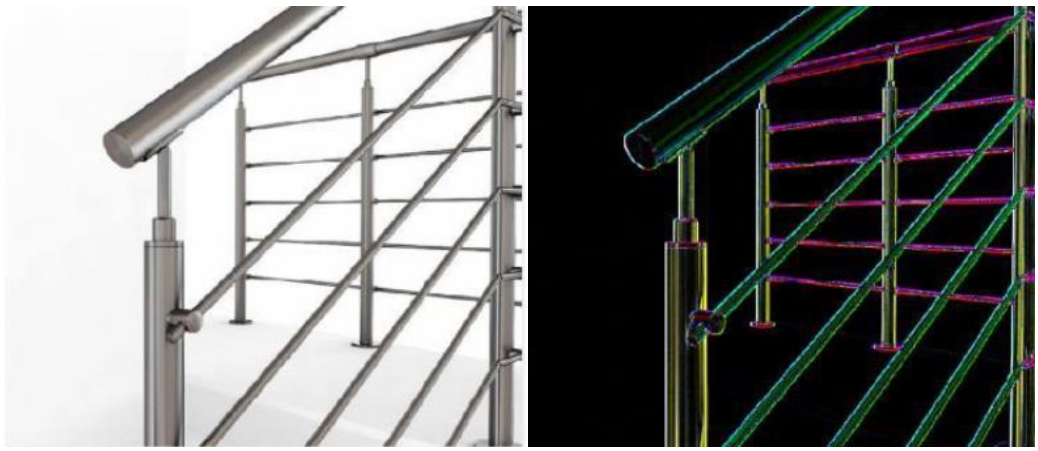
Bu konuyu deneyerek görmeye çalışın.

Ödevler

Ödev 1: Aşağıdaki kenar bulma algoritmalarının programlarını yazınız. 5 tane farklı tarzlarda resim üzerinde en iyi kenar bulan algoritma hangisidir (Bakış açısına göre yada aranan özelliğe göre değişebilir ama sizin kendi yorumunuz olsun) karşılaştırın. Örneğin manzara resminde en iyi kenar bulan Sobeldir, İnsa yüzü görüntüsünde en iyi kenar bulan Aşındırma algoritaması gibi yorumlar yapın ve bunları görsel olarak gösterin. Programlayacağınız algoritmalar;

a) Sobel
b) Prewitt
c) Cany (bu algoritma ders notlarında yoktur. Araştırın programını kendiniz yazın. Diğerlerinin kodları notlarda vardır. Sadece kendinize uyarlayın.)
d) Bulanıklaştırma ile kenar bulma (6 notlarda)
e) Aşındırma ile kenar bulma (8 notlarda)

Ödev 2: Çelik kontrüksiyon şeklinde aşağıdaki örneklere benzer yapıların üzerindeki çizgilerin açılarını bulup bunları renklendirin. Bu işlem için Sobel ve Compass algoritmalarını kullanın. Hangi renklerin hangi açıyı gösterdiğini anlayabilmek için kenarda bir grafik oluşturun yani resmin dış tarafında belirlediğiniz renkleri gösteren ve kaç derece açıya karşılık geldiğini gösteren bir grafik olsun. (her adım 10 derece şeklinde gösterilebilir. 360 derece açıların hepsini göstersin).

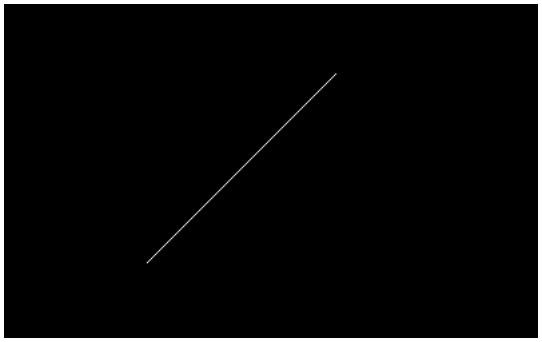
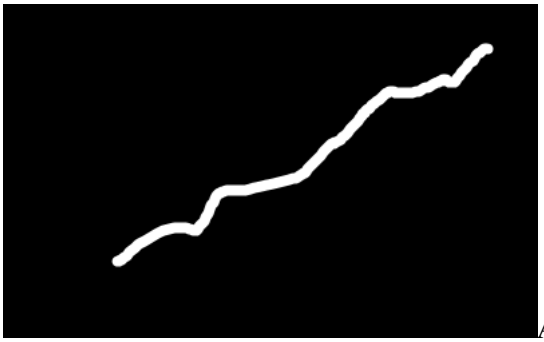


Ödev 3: 6 ve 7 nolu dökümanlardaki kodları hazır olarak verilmiş olan tüm algoritmaları kendi yazdığınız program içerisinde deneyin. Her algoritmanın en iyi sonuç verdiği örnek resimler bularak uygulamasını gösterin. Denediğiniz her algoritmayı hangi resimlerde daha iyi sonuç verdiğinin sebebini yorumlayarak birer cümle ile açıklayın.

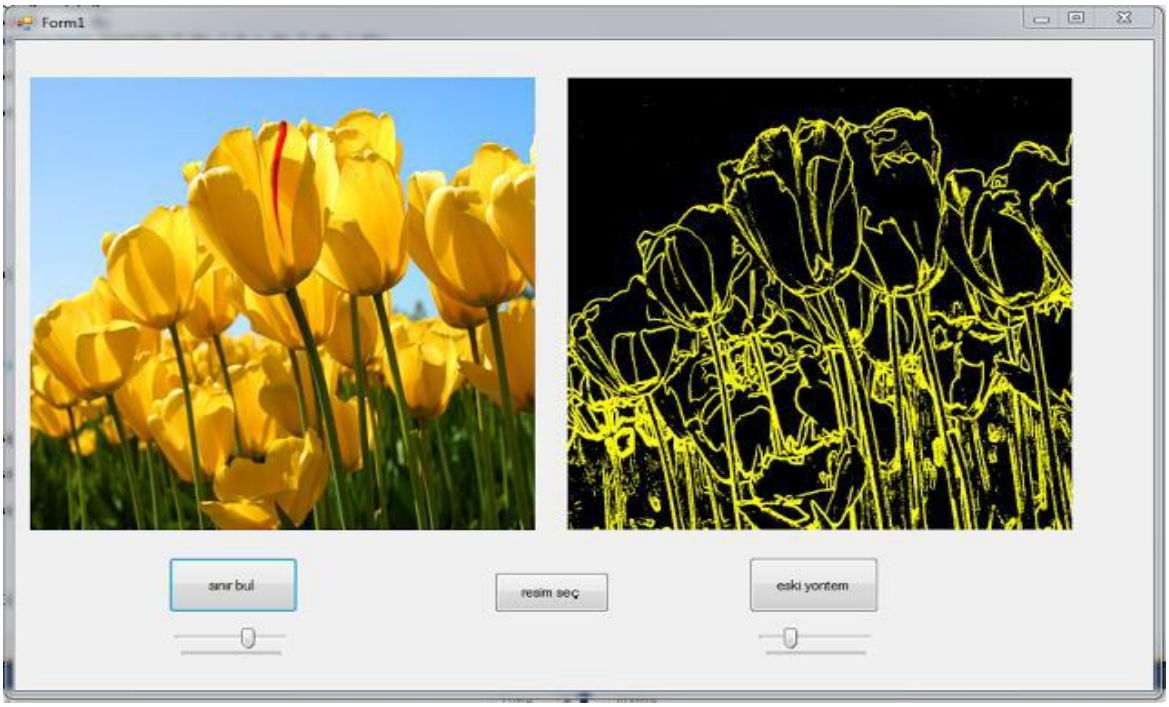
Ödev 4: Compas algoritmasını farklı 3 matris türüyle deneyerek renkli çizgileri kaliteli bir şekilde programı yazınız.

Ödev 5: Siyah zemin üzerine düz bir çizgi çizdirin. Kenar bulma algoritmalarından Sobel kullanarak (2 matris ile) bu çizginin açısını buldurun. Açı değerini Textbox içerisine yazdırın. Aynı açı bulma işini Compass algoritması ile (8 matris kullanarak) tekrar hesaplatın. Hangisi daha hassas hesaplamaktadır karşılaştırın.

Birde resim üzerine düz olmayan yaklaşık olarak düz giden fakat dağınık bir şekilde duran resmin içerisindeki görüntünün açısını hesaplatın. Böylece net bir çizgide ne kadar doğru hesaplıyor bulun. Ayrıca dağınık bir çizgide ortalama açıyı bulabiliyor mu gösterin. Örnek resimler aşağıdaki gibi olabilir.


Açı:48

Açı:35 gibi

Ödev 6: Aşağıdaki örnek programın çalışma mantığını bulun ve kullanılan matris ve yapı ortaya çıkarın. Buradaki kodlarda değişken adlarını daha anlaşılır, kelimelerden oluşacak şekilde yeniden düzenleyin. Program içerisinde kullanılan Listeler dizi yapısının nasıl çalıştığını anlatın. (Tek boyutlu listeler İTP 3 nolu notlarda vardır). Matris değerleri için bu yapıyı diğer Ödevlerde de programlayarak, kullanabildiğinizi gösterin.



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace goruntuisleme
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        Bitmap resim;
        private void button1_Click(object sender, EventArgs e)
        {
            openFileDialog1.Filter = "Resim Dosyaları|" +
"*.bmp;*.jpg;*.gif;*.wmf;*.tif;*.png";


          if (openFileDialog1.ShowDialog() == DialogResult.OK)
          {
              pictureBox2.Image = null;
              pictureBox1.Image = Image.FromFile(openFileDialog1.FileName);
          }


public Bitmap sinirbul(Bitmap resim)
       {
           Bitmap ret = new Bitmap(resim.Width, resim.Height);
           for (int i = 1; i < resim.Width - 1; i++)
           {
               for (int j = 1; j < resim.Height - 1; j++)
               {
                   Color cr = resim.GetPixel(i + 1, j);
                   Color cl = resim.GetPixel(i - 1, j);
                   Color cu = resim.GetPixel(i, j - 1);
                   Color cd = resim.GetPixel(i, j + 1);
                   Color cld = resim.GetPixel(i - 1, j + 1);
                   Color clu = resim.GetPixel(i - 1, j - 1);
                   Color crd = resim.GetPixel(i + 1, j + 1);
                   Color cru = resim.GetPixel(i + 1, j - 1);
                   int power = getMaxD(cr.R, cl.R, cu.R, cd.R, cld.R, clu.R,
cru.R, crd.R);
```

```
                    if (power > 50)
                        ret.SetPixel(i, j, Color.Yellow);
                    else
                        ret.SetPixel(i, j, Color.Black);
                }
            }
            return ret;
        }
        private int getD(int cr, int cl, int cu, int cd, int cld, int clu, int
cru, int crd, int[,] matrix)
        {
            return Math.Abs(matrix[0, 0] * clu + matrix[0, 1] * cu + matrix[0, 2]
* cru
               + matrix[1, 0] * cl + matrix[1, 2] * cr
                  + matrix[2, 0] * cld + matrix[2, 1] * cd + matrix[2, 2] * crd);
        }
        private int getMaxD(int cr, int cl, int cu, int cd, int cld, int clu, int
cru, int crd)
        {
            int max = int.MinValue;
            for (int i = 0; i < templates.Count; i++)
            {
                int newVal = getD(cr, cl, cu, cd, cld, clu, cru, crd,
templates[i]);
                if (newVal > max)
                    max = newVal;
            }
            return max;
        }

            private List<int[,]> templates = new List<int[,]>
   {
      new int[,] {{ -3, -3, 5 }, { -3, 0, 5 }, { -3, -3, 5 } },


     new int[,] {{ -3, 5, 5 }, { -3, 0, 5 }, { -3, -3, -3 } },
     new int[,] {{ 5, 5, 5 }, { -3, 0, -3 }, { -3, -3, -3 } },
     new int[,] {{ 5, 5, -3 }, { 5, 0, -3 }, { -3, -3, -3 } },
     new int[,] {{ 5, -3, -3 }, { 5, 0, -3 }, { 5, -3, -3 } },
     new int[,] {{ -3, -3, -3 }, { 5, 0, -3 }, { 5, 5, -3 } },
     new int[,] {{ -3, -3, -3 }, { -3, 0, -3 }, { 5, 5, 5 } },
     new int[,] {{ -3, -3, -3 }, { -3, 0, 5 }, { -3, 5, 5 } }
  };
```

Yukarıdaki kodların, bizim kodlara dönüştürülmüş hali. Kodlar Compass Algoritması mantığında çalışıyor. 9*9 boyutunda 8 tane matris arasında bir pixel için en büyük değeri bulup, eşikleme yaparak o pikselin değerini belirliyor.

```
// -------------------------------- DÖNÜŞTÜRÜLEN PROGRAM BLOGU ------------------------------
-------------
private void toolStripButton2_Click(object sender, EventArgs e)
{
    Bitmap GirisResmi, CikisResmi;
    GirisResmi = new Bitmap(pictureBox1.Image);
    int ResimGenisligi = GirisResmi.Width;
    int ResimYuksekligi = GirisResmi.Height;
    CikisResmi = new Bitmap(ResimGenisligi, ResimYuksekligi);
    int x, y;
    Color Renk;
    int P1, P2, P3, P4, P5, P6, P7, P8, P9;
    int GM, G;
    //int[,] Matris = { { -3, -3, 5, -3, 0, 5, -3, -3, 5 }, { -3, 5, 5, -3, 0, 5, -3, -3, -3 },
{ 5, 5, 5, -3, 0, -3, -3, -3, -3 }, { 5, 5, -3, 5, 0, -3, -3, -3, -3 }, { 5, -3, -3, 5, 0, 3, 5,
```

```
-3, -3 }, { -3, -3, -3, 5, 0, -3, 5, 5, -3 }, { -3, -3, -3, -3, 0, -3, 5, 5, 5 }, { -3, -3, -3, -3, 0, 5, -3, 5, 5 } }; // Kirsch Matrisi
    //int[,] Matris = { { -1, -1, -1, 1, -2, 1, 1, 1, 1 }, { -1, -1, 1, -1, -2, 1, 1, 1, 1 }, { -1, 1, 1, -1, -2, 1, -1, 1, 1 }, { 1, 1, 1, -1, -2, 1, -1, -1, 1 }, { 1, 1, 1, 1, -2, 1, -1, -1, -1 }, { 1, 1, 1, 1, -2, -1, 1, -1, -1 }, { 1, 1, -1, 1, -2, -1, 1, 1, -1 }, { 1, -1, -1, 1, -2, -1, 1, 1, 1 } }; // Compass Matrisi
    //int[,] Matris = { { -1, 0, 1, -2, 0, 2, -1, 0, 1 }, { 0, 1, 2, -1, 0, 1, -2, -1, 0 }, { 1, 2, 1, 0, 0, 0, -1, -2, -1 }, { 2, 1, 0, 1, 0, -1, 0, -1, -2 }, { 1, 0, -1, 2, 0, -2, 1, 0, -1 }, { 0, -1, -2, 1, 0, -1, 2, 1, 0 }, { -1, -2, -1, 0, 0, 0, 1, 2, 1 }, { -2, -1, 0, -1, 0, 1, 0, 1, 2 } }; // Sobel Matrisi;
    int[,] Matris = { { -1, 0, 1, -1, 0, 1, -1, 0, 1 }, { 0, 1, 1, -1, 0, 1, -1, -1, 0 }, { 1, 1, 1, 0, 0, 0, -1, -1, -1 }, { 1, 1, 0, 1, 0, -1, 0, -1, -1 }, { 1, 0, -1, 1, 0, -1, 1, 0, -1 }, { 0, -1, -1, 1, 0, -1, 1, 1, 0 }, {-1, -1, -1, 0, 0, 0, 1, 1, 1 }, {-1, -1, 0, -1, 0, 1, 0, 1, 1 } }; // Robinson Matrisi

    for (x = 1; x < ResimGenisligi - 1; x++) //Resmi taramaya şablonun yarısı kadar dış kenarlardan içeride başlayacak ve bitirecek.
    {
        for (y = 1; y < ResimYuksekligi - 1; y++)
        {
            Renk = GirisResmi.GetPixel(x - 1, y - 1);
            P1 = (Renk.R + Renk.G + Renk.B) / 3;
            Renk = GirisResmi.GetPixel(x, y - 1);
            P2 = (Renk.R + Renk.G + Renk.B) / 3;
            Renk = GirisResmi.GetPixel(x + 1, y - 1);
            P3 = (Renk.R + Renk.G + Renk.B) / 3;
            Renk = GirisResmi.GetPixel(x - 1, y);
            P4 = (Renk.R + Renk.G + Renk.B) / 3;
            Renk = GirisResmi.GetPixel(x, y);
            P5 = (Renk.R + Renk.G + Renk.B) / 3;
            Renk = GirisResmi.GetPixel(x + 1, y);
            P6 = (Renk.R + Renk.G + Renk.B) / 3;
            Renk = GirisResmi.GetPixel(x - 1, y + 1);
            P7 = (Renk.R + Renk.G + Renk.B) / 3;
            Renk = GirisResmi.GetPixel(x, y + 1);
            P8 = (Renk.R + Renk.G + Renk.B) / 3;
            Renk = GirisResmi.GetPixel(x + 1, y + 1);
            P9 = (Renk.R + Renk.G + Renk.B) / 3;
            GM = 0;

            for (int i = 0; i < 8; i++)
            {
                G = Math.Abs(P1 * Matris[i, 0] + P2 * Matris[i, 1] + P3 * Matris[i, 2] + P4 * Matris[i, 3] + P5 * Matris[i, 4] + P6 * Matris[i, 5] + P7 * Matris[i, 6] + P8 * Matris[i, 7] + P9 * Matris[i, 8]);
                if (G > GM) GM = G;
            }
            if (GM > 50)
                CikisResmi.SetPixel(x, y, Color.Yellow);
            else
                CikisResmi.SetPixel(x, y, Color.Black);
        }
    }
    pictureBox2.Image = CikisResmi;
}
```

Araştırma:

Aşağıdaki linkte verilen gelişmiş kırpma işleminde Sobel Filtresi kullanılmaktadır. Bu kırpma işleminde kullanılan algoritmayı programlayın.

http://www.cescript.com/2015/07/icerik-tabanli-imge-olcekleme.html

Yukarıdaki enerji haritası yardımıyla farklı teknikler kullanılarak boyut indirgemesi yapılabilir. Bunlardan biri her satırdan, o satırın en düşük enerjili piksel değerini silmektir. Ancak böyle bir silme işlemi her satırda farklı sütunlardan pikseller sileceğinden görüntüde kaymalara/bozulmalara neden olacaktır. Kaymaları engellemek için kullanılacak bir diğer yöntem ise her seferinde en düşük enerjili sütunu kaldırmaktır. Bu durumda ise imge içerisinde bazı önemli nesnelerin silinebilmesi söz konusudur. Bildiride önerilen ve yukarıdaki olumsuzlukları gideren yöntem en düşük enerjili yolu silmeyi önermektedir. Enerjili yolu (seam), görüntünün tepesinden başlayıp tabanında son bulan ve her satırdan tek bir kez geçen eğrilere verilen isimdir. Amacımız imge boyunu azaltırken oluşacak bilgi kaybını en aza indirmek olduğundan, olası tüm yollar içerisinden en düşük enerjili yolun bulunarak silinmesi gerekmektedir.

Ancak arama uzayımız imgenin genişliği ile doğrusal, yüksekliği ile üstel büyüdüğünden, olası tüm yolların bulunup en düşük enerjili eğrinin seçilmesi verimli olmamaktadır. Amacımız, bu yollar içerisinden en düşük enerjili olanı bulmak olduğundan, dinamik programlamayı kullanarak görüntü boyu ile doğrusal karmaşıklıkta problemi çözebiliriz. Algoritma uzun ve karmaşık olduğundan detayları bir sonraki yazıya bırakıyorum. Şimdilik bulunan bu yolun path dizisinde saklandığını ve bu dizinin y' inci elemanı, eğrinin x koordinatını (x = path[y]) içerdiğini bilmemiz yeterli.

****

Canny Edge Detector

Common Names: Canny edge detector

Brief Description

The Canny operator was designed to be an optimal edge detector (according to particular criteria --- there are other detectors around that also claim to be optimal with respect to slightly different criteria). It takes as input a gray scale image, and produces as output an image showing the positions of tracked intensity discontinuities.

How It Works

The Canny operator works in a multi-stage process. First of all the image is smoothed by Gaussian convolution. Then a simple 2-D first derivative operator (somewhat like the Roberts Cross) is applied to the smoothed image to highlight regions of the image with high first spatial derivatives. Edges give rise to ridges in the gradient magnitude image. The algorithm then tracks along the top of these ridges and sets to zero all pixels that are not actually on the ridge top so as to give a thin line in the output, a process known as *non-maximal suppression*. The tracking process exhibits hysteresis controlled by two thresholds: *T1* and *T2*, with *T1* > *T2*. Tracking can only begin at a point on a ridge higher than *T1*. Tracking then continues in both directions out from that point until the height of the ridge falls below *T2*. This hysteresis helps to ensure that noisy edges are not broken up into multiple edge fragments.

Guidelines for Use

The effect of the Canny operator is determined by three parameters --- the width of the Gaussian kernel used in the smoothing phase, and the upper and lower thresholds used by the tracker. Increasing the width of the Gaussian kernel reduces the detector's sensitivity to noise, at the expense of losing some of the finer detail in the image. The localization error in the detected edges also increases slightly as the Gaussian width is increased.

Usually, the upper tracking threshold can be set quite high, and the lower threshold quite low for good results. Setting the lower threshold too high will cause noisy edges to break up.

Setting the upper threshold too low increases the number of spurious and undesirable edge fragments appearing in the output.

One problem with the basic Canny operator is to do with Y-junctions *i.e.* places where three ridges meet in the gradient magnitude image. Such junctions can occur where an edge is partially occluded by another object. The tracker will treat two of the ridges as a single line segment, and the third one as a line that approaches, but doesn't quite connect to, that line segment.

We use the image

to demonstrate the effect of the Canny operator on a natural scene.

Using a Gaussian kernel with standard deviation 1.0 and upper and lower thresholds of 255 and 1, respectively, we obtain

Most of the major edges are detected and lots of details have been picked out well --- note that this may be too much detail for subsequent processing. The `Y-Junction effect' mentioned above can be seen at the bottom left corner of the mirror.

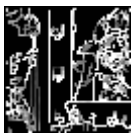
The image

is obtained using the same kernel size and upper threshold, but with the lower threshold increased to 220. The edges have become more broken up than in the previous image, which is likely to be bad for subsequent processing. Also, the vertical edges on the wall have not been detected, along their full length.

The image

is obtained by lowering the upper threshold to 128. The lower threshold is kept at 1 and the Gaussian standard deviation remains at 1.0. Many more faint edges are detected along with some short `noisy' fragments. Notice that the detail in the clown's hair is now picked out.

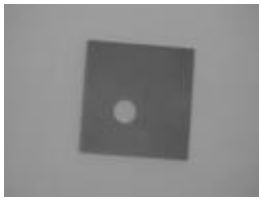
The image



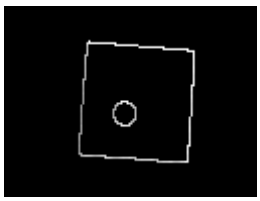
is obtained with the same thresholds as the previous image, but the Gaussian used has a standard deviation of 2.0. Much of the detail on the wall is no longer detected, but most of the strong edges remain. The edges also tend to be smoother and less noisy.

Edges in artificial scenes are often sharper and less complex than those in natural scenes, and this generally improves the performance of any edge detector.

The image



shows such an artificial scene, and



is the output from the Canny operator.

The Gaussian smoothing in the Canny edge detector fulfills two purposes: first it can be used to control the amount of detail that appears in the edge image and second, it can be used to suppress noise.

To demonstrate how the Canny operator performs on noisy images we use

which contains Gaussian noise with a standard deviation of *15*. Neither the Roberts Cross nor the Sobel operator are able to detect the edges of the object while removing all the noise in the image. Applying the Canny operator using a standard deviation of *1.0* yields

All the edges have been detected and almost all of the noise has been removed. For comparison,

is the result of applying the Sobel operator and thresholding the output at a value of *150*.

We use

to demonstrate how to control the details contained in the resulting edge image. The image

is the result of applying the Canny edge detector using a standard deviation of *1.0* and an upper and lower threshold of *255* and *1*, respectively. This image contains many details; however, for an automated recognition task we might be interested to obtain only lines that correspond to the boundaries of the objects. If we increase the standard deviation for the Gaussian smoothing to *1.8*, the Canny operator yields

Now, the edges corresponding to the uneveness of the surface have disappeared from the image, but some edges corresponding to changes in the surface orientation remain. Although

these edges are `weaker' than the boundaries of the objects, the resulting pixel values are the same, due to the saturation of the image. Hence, if we scale down the image before the edge detection, we can use the upper threshold of the edge tracker to remove the weaker edges. The image

is the result of first scaling the image with *0.25* and then applying the Canny operator using a standard deviation of *1.8* and an upper and lower threshold of *200* and *1*, respectively. The image shows the desired result that all the boundaries of the objects have been detected whereas all other edges have been removed.

Although the Canny edge detector allows us the find the intensity discontinuities in an image, it is not guaranteed that these discontinuities correspond to actual edges of the object. This is illustrated using

We obtain

by using a standard deviation of *1.0* and an upper and lower threshold of *255* and *1*, respectively. In this case, some edges of the object do not appear in the image and many edges in the image originate only from reflections on the object. It is a demanding task for an automated system to interpret this image. We try to improve the edge image by decreasing the upper threshold to *150*, as can be seen in

We now obtain most of the edges of the object, but we also increase the amount of noise. The result of further decreasing the upper threshold to *100* and increasing the standard deviation to *2* is shown in

Common Variants

The problem with Y-junctions mentioned above can be solved by including a model of such junctions in the ridge tracker. This will ensure that no spurious gaps are generated at these junctions.

Interactive Experimentation

You can interactively experiment with this operator by clicking here.

Exercises

1. Adjust the parameters of the Canny operator so that you can detect the edges of

   while removing *all* of the noise.

2. What effect does increasing the Gaussian kernel size have on the magnitudes of the gradient maxima at edges? What change does this imply has to be made to the tracker thresholds when the kernel size is increased?
3. It is sometimes easier to evaluate edge detector performance after thresholding the edge detector output at some low gray scale value (*e.g.* 1) so that all detected edges are marked by bright white pixels. Try this out on the third and fourth example images of the clown mentioned above. Comment on the differences between the two images.
4. How does the Canny operator compare with the Roberts Cross and Sobel edge detectors in terms of speed? What do you think is the slowest stage of the process?
5. How does the Canny operator compare in terms of noise rejection and edge detection with other operators such as the Roberts Cross and Sobel operators?
6. How does the Canny operator compare with other edge detectors on simple artificial 2-D scenes? And on more complicated natural scenes?

7. Under what situations might you choose to use the Canny operator rather than the Roberts Cross or Sobel operators? In what situations would you definitely not choose it?

References

R. Boyle and R. Thomas *Computer Vision: A First Course*, Blackwell Scientific Publications, 1988, p 52.

J. Canny *A Computational Approach to Edge Detection*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 8, No. 6, Nov. 1986.

E. Davies *Machine Vision: Theory, Algorithms and Practicalities*, Academic Press, 1990, Chap. 5.

R. Gonzalez and R. Woods *Digital Image Processing*, Addison-Wesley Publishing Company, 1992, Chap. 4.

Local Information

Specific information about this operator may be found here.

More general advice about the local HIPR installation is available in the *Local Information* introductory section.

Canny edge detector

1. Smooth image with a Gaussian filter
2. Approximate gradient magnitude and angle (use Sobel, Prewitt . . .)

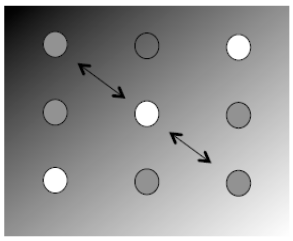
$$M[x,y] \approx \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \qquad \alpha[x,y] \approx \tan^{-1}\left(\frac{\partial f}{\partial y} \Big/ \frac{\partial f}{\partial x}\right)$$

3. Apply nonmaxima suppression to gradient magnitude
4. Double thresholding to detect strong and weak edge pixels
5. Reject weak edge pixels not connected with strong edge pixels

[Canny, IEEE Trans. PAMI, 1986]

Canny nonmaxima suppression

- Quantize edge normal to one of four directions: horizontal, -45°, vertical, +45°
- If $M[x,y]$ is smaller than either of its neighbors in edge normal direction
  ➔ suppress; else keep.



Canny thresholding and suppression of weak edges

- Double-thresholding of gradient magnitude

$$\text{Strong edge:} \quad M[x,y] \geq \theta_{high}$$
$$\text{Weak edge:} \quad \theta_{high} > M[x,y] \geq \theta_{low}$$

- Typical setting: $\theta_{high} / \theta_{low} = 2...3$
- Region labeling of edge pixels
- Reject regions without strong edge pixels

Canny edge detector

$\sigma = \sqrt{2}$ $\sigma = 2\sqrt{2}$ $\sigma = 4\sqrt{2}$

Canny Kenar Bulma Algoritması

```
//CANNY ALGORITMASI **********************************
private void toolStripButton3_Click(object sender, EventArgs e)
{
    Bitmap newBitmap = (Bitmap)pictureBox1.Image;
    Bitmap newBitmap1 = new Bitmap(newBitmap.Width, newBitmap.Height);
    newBitmap1 = MakeGrayscale(newBitmap);
    newBitmap1 = MakeSmooth(newBitmap1);
    newBitmap1 = DetectEdge(newBitmap1);
    pictureBox2.Image = newBitmap1;

}

private Bitmap MakeGrayscale(Bitmap original)
{
    try
    {
        Color originalColor;
        Color newColor;
        Bitmap newBitmap = new Bitmap(original.Width, original.Height);
        for (int i = 0; i < original.Width; i++)
            for (int j = 0; j < original.Height; j++)
            {
                originalColor = original.GetPixel(i, j);
                int grayScale = (int)((originalColor.R * 0.3) + (originalColor.G * 0.59) +
(originalColor.B * 0.11));
                newColor = Color.FromArgb(grayScale, grayScale, grayScale);
                newBitmap.SetPixel(i, j, newColor);
            }
        return newBitmap;
    }
    catch
    {
        throw new NotImplementedException();
    }
}
private Bitmap MakeSmooth(Bitmap original)
{
    int runningSum = 0;
    int tempSum = 0;
    int xcoord;
    int ycoord;
    Color newColor;
    int[,] kernel = new int[5, 5] {
        {1, 4, 7, 4, 1},
        {4, 16, 26, 16, 4},
        {7, 26, 41, 26, 7},
        {4, 16, 26, 16, 4},
        {1, 4, 7, 4, 1}
        };
    Color[,] pixels = new Color[5, 5];
    Bitmap newBitmap = new Bitmap(original.Width, original.Height);
    for (int i = 0; i < original.Width; i++)
        for (int j = 0; j < original.Height; j++)
        {
            for (int x = -2; x < 3; x++)
                for (int y = -2; y < 3; y++)
                {
                    xcoord = i + x;
                    ycoord = j + y;
                    if (xcoord < 0 || xcoord > original.Width - 1)
                        xcoord = i - x;
                    if (ycoord < 0 || ycoord > original.Height - 1)
```

```
                        ycoord = j - y;
                    pixels[x + 2, y + 2] = original.GetPixel(xcoord, ycoord);
                }
            for (int k = 0; k < 5; k++)
                for (int l = 0; l < 5; l++)
                    tempSum += kernel[k, l] * pixels[k, l].R;
            runningSum = tempSum / 273;
            newColor = Color.FromArgb(runningSum, runningSum, runningSum);
            newBitmap.SetPixel(i, j, newColor);
            tempSum = 0;
            runningSum = 0;
        }
    return newBitmap;
}
private Bitmap DetectEdge(Bitmap original)
{
    Bitmap newBitmap = new Bitmap(original.Width, original.Height);
    int xleft;
    int xright;
    int ytop;
    int ybot;
    double gx;
    double gy;
    double tempAngle;
    Color color1, color2;
    double[,] magnitudes = new double[original.Width, original.Height];
    double[,] angles = new double[original.Width, original.Height];
    bool[,] isEdge = new bool[original.Width, original.Height];
    double maxMag = 0;
    for (int i = 0; i < original.Width; i++)
        for (int j = 0; j < original.Height; j++)
        {
            xleft = i - 1;
            xright = i + 1;
            ytop = j - 1;
            ybot = j + 1;
            if (xleft < 0)
                xleft = xright;
            if (xright > original.Width - 1)
                xright = xleft;
            if (ytop < 0)
                ytop = ybot;
            if (ybot > original.Height - 1)
                ybot = ytop;
            color1 = original.GetPixel(xright, j);
            color2 = original.GetPixel(xleft, j);
            gx = (color1.R - color2.R) / 2;
            color1 = original.GetPixel(i, ybot);
            color2 = original.GetPixel(i, ytop);
            gy = (color1.R - color2.R) / 2;
            magnitudes[i, j] = Math.Abs(gx) + Math.Abs(gy);
            if (magnitudes[i, j] > maxMag)
                maxMag = magnitudes[i, j];
            tempAngle = Math.Atan(gy / gx);
            tempAngle = tempAngle * 180 / Math.PI;
            if ((tempAngle >= 0 && tempAngle < 22.5) || (tempAngle > 157.5 && tempAngle <= 180)
|| (tempAngle <= 0 && tempAngle > -22.5) || (tempAngle < -157.5 && tempAngle >= -180))
                tempAngle = 0.0;
            else if ((tempAngle > 22.5 && tempAngle < 67.5) || (tempAngle < -22.5 && tempAngle >
-67.5))
                tempAngle = 45.0;
            else if ((tempAngle > 67.5 && tempAngle < 112.5) || (tempAngle < -67.5 && tempAngle
> -112.5))
                tempAngle = 90.0;
            else if ((tempAngle > 112.5 && tempAngle < 157.5) || (tempAngle < -112.5 &&
tempAngle > -157.5))
```

```
                tempAngle = 135.0;
            angles[i, j] = tempAngle;
        }
    for (int i = 0; i < original.Width; i++)
        for (int j = 0; j < original.Height; j++)
        {
            if ((i - 1 < 0 || i + 1 > original.Width - 1 || j - 1 < 0 || j + 1 > original.Height
- 1))
            {
                isEdge[i, j] = false;
            }
            else if (angles[i, j] == 0.0)
            {
                if (magnitudes[i, j] > magnitudes[i - 1, j] && magnitudes[i, j] > magnitudes[i +
1, j])
                    isEdge[i, j] = true;
                else
                    isEdge[i, j] = false;
            }
            else if (angles[i, j] == 90.0)
            {
                if (magnitudes[i, j] > magnitudes[i, j - 1] && magnitudes[i, j] > magnitudes[i,
j + 1])
                    isEdge[i, j] = true;
                else
                    isEdge[i, j] = false;
            }
            else if (angles[i, j] == 135.0)
            {
                if (magnitudes[i, j] > magnitudes[i - 1, j - 1] && magnitudes[i, j] >
magnitudes[i + 1, j + 1])
                    isEdge[i, j] = true;
                else
                    isEdge[i, j] = false;
            }
            else if (angles[i, j] == 45.0)
            {
                if (magnitudes[i, j] > magnitudes[i + 1, j - 1] && magnitudes[i, j] >
magnitudes[i - 1, j + 1])
                    isEdge[i, j] = true;
                else
                    isEdge[i, j] = false;
            }
        }
    double lowerThreshold = maxMag * 0.10;
    for (int i = 0; i < original.Width; i++)
        for (int j = 0; j < original.Height; j++)
        {
            if (isEdge[i, j] && magnitudes[i, j] > lowerThreshold)
            {
                if (angles[i, j] == 0.0)
                {
                    if (angles[i, j] == angles[i - 1, j] || angles[i, j] == angles[i + 1, j])
                    {
                        if (magnitudes[i - 1, j] > lowerThreshold)
                            newBitmap.SetPixel(i - 1, j, Color.White);
                        else
                            newBitmap.SetPixel(i - 1, j, Color.Black);
                        if (magnitudes[i + 1, j] > lowerThreshold)
                            newBitmap.SetPixel(i + 1, j, Color.White);
                        else
                            newBitmap.SetPixel(i + 1, j, Color.Black);
                    }
                }
                else if (angles[i, j] == 90.0)
                {
```

```
                    if (angles[i, j] == angles[i, j - 1] || angles[i, j] == angles[i, j + 1])
                    {
                        if (magnitudes[i, j - 1] > lowerThreshold)
                            newBitmap.SetPixel(i, j - 1, Color.White);
                        else
                            newBitmap.SetPixel(i, j - 1, Color.Black);
                        if (magnitudes[i, j + 1] > lowerThreshold)
                            newBitmap.SetPixel(i, j + 1, Color.White);
                        else
                            newBitmap.SetPixel(i, j + 1, Color.Black);
                    }
                }
                else if (angles[i, j] == 135.0)
                {
                    if (angles[i, j] == angles[i - 1, j - 1] || angles[i, j] == angles[i + 1, j
+ 1])
                    {
                        if (magnitudes[i - 1, j - 1] > lowerThreshold)
                            newBitmap.SetPixel(i - 1, j - 1, Color.White);
                        else
                            newBitmap.SetPixel(i - 1, j - 1, Color.Black);
                        if (magnitudes[i + 1, j + 1] > lowerThreshold)
                            newBitmap.SetPixel(i + 1, j + 1, Color.White);
                        else
                            newBitmap.SetPixel(i + 1, j + 1, Color.Black);
                    }
                }
                else if (angles[i, j] == 45.0)
                {
                    if (angles[i, j] == angles[i + 1, j - 1] || angles[i, j] == angles[i - 1, j
+ 1])
                    {
                        if (magnitudes[i + 1, j - 1] > lowerThreshold)
                            newBitmap.SetPixel(i + 1, j - 1, Color.White);
                        else
                            newBitmap.SetPixel(i + 1, j - 1, Color.Black);
                        if (magnitudes[i - 1, j + 1] > lowerThreshold)
                            newBitmap.SetPixel(i - 1, j + 1, Color.White);
                        else
                            newBitmap.SetPixel(i - 1, j + 1, Color.Black);
                    }
                }
            }
            else
                newBitmap.SetPixel(i, j, Color.Black);
        }
    return newBitmap;
}
```

Araştırma: Sobel Filtresi ile ilgili olarak aşağıda yazılmış olan kodları inceleyiniz. Yazılan bu programın notlarda geçen anlatımdan farkı nedir? Programlama tarzından çıkarılabilecek bilgiler nelerdir?

```
//SOBEL FİLTRESİ*****************************
private void btnSobelFiltresi_Click(object sender, EventArgs e)
{
    Bitmap sobelResim = SobelUygula();
    pictureBox5.Image = sobelResim;
    btnResmiDonustur.Enabled = true;
}

private Bitmap SobelUygula()
{
```

```
    Bitmap temp = new Bitmap(pictureBox2.Image);
    BitmapData tempData = temp.LockBits(new Rectangle(0, 0, temp.Width, temp.Height),
                                                        ImageLockMode.ReadWrite,
PixelFormat.Format24bppRgb);
    uzunlukX = pictureBox2.Image.Width;
    uzunlukY = pictureBox2.Image.Height;

    dizi = new int[uzunlukX, uzunlukY];
    //progressBar1.Maximum = pictureBox2 .Image.Width * pictureBox2 .Image.Height;

    unsafe
    {
        byte* ptrAdres = (byte*)tempData.Scan0;     //bağlantı adresini al
        int kalan = tempData.Stride - tempData.Width * 3;
        for (int i = 0; i < tempData.Height; i++)
        {
            for (int j = 0; j < tempData.Width; j++)
            {
                dizi[j, i] = (int)((double)ptrAdres[0] * 0.11 + (double)ptrAdres[1] * 0.59 +
(double)ptrAdres[2] * 0.3);  //red green blue deÄŸerleriyle gri iÃ§in matrisi oluÅŸtur
                ptrAdres += 3;      // 3 byte ileri git. bir pixel de 3 byte lÄ±k bilgi var
                if ((i % 5) == 0)//her on satırda bir göstergeyi güncelle
                {
                    progressBar1.Value = i * pictureBox2.Image.Height + j;
                    Application.DoEvents();
                }
            }
            ptrAdres += kalan;
        }
    }
    dizi = Filitre_SobelSatir_Sutun(dizi);
    int max = 0;

    for (int y = 0; y < dizi.GetLength(0); y++)
    {
        for (int x = 0; x < dizi.GetLength(1); x++)
        {
            if (dizi[y, x] < 0)
            {
                dizi[y, x] *= -1;
            }
            if (dizi[y, x] > max)
            {
                max = dizi[y, x];
            }
        }
    }

    for (int y = 0; y < dizi.GetLength(0); y++)
    {
        for (int x = 0; x < dizi.GetLength(1); x++)
        {
            dizi[y, x] = Convert.ToInt32((dizi[y, x] * 255));
            dizi[y, x] = Convert.ToInt32((dizi[y, x] / max));
        }
    }
    progressBar1.Visible = false;
    return ResmiCiz(dizi);

}


private Bitmap ResmiCiz(int[,] dizi)
{
        Bitmap tmp = new Bitmap(uzunlukX, uzunlukY);
```

```
        for (int y = 0; y < uzunlukY; y++)
        {
            for (int x = 0; x < uzunlukX; x++)
            {
                tmp.SetPixel(x, y, Color.FromArgb(dizi[x, y], dizi[x, y], dizi[x, y]));
            }
        }
        return tmp;
}

public int[,] Filitre_SobelSatir_Sutun(int[,] resim)
{
    int[,] resim1 = Filitrele(Sobel_SatirMask, resim);
    int[,] resim2 = Filitrele(Sobel_SutunMask, resim);
    int[,] resim3 = Filitrele(Sobel_SatirSutunMask_L, resim);
    int[,] resim4 = Filitrele(Sobel_SatirSutunMask_R, resim);

    int boy = resim.GetLength(0);
    int en = resim.GetLength(1);

    int[,] tmp = new int[boy, en];

    for (int y = 1; y < boy - 1; y++)
    {
        for (int x = 1; x < en - 1; x++)
        {
            tmp[y, x] = Convert.ToInt32(Math.Sqrt((resim1[y, x]) * (resim1[y, x]) +
                                                  (resim2[y, x]) * (resim2[y, x])) +
                                        Math.Sqrt((resim3[y, x]) * (resim3[y, x]) +
                                                  (resim4[y, x]) * (resim4[y, x])));
        }
    }
    return tmp;
}

public int[,] Filitrele(int[,] maske, int[,] resim)
{
    int boy = resim.GetLength(0);
    int en = resim.GetLength(1);

    int[,] temp = new int[boy, en];

    for (int y = 1; y < boy - 1; y++)
    {
        for (int x = 1; x < en - 1; x++)
        {
            temp[y, x] = ((maske[0, 0] * resim[y - 1, x - 1]) + (maske[0, 1] * resim[y - 1, x])
+ (maske[0, 2] * resim[y - 1, x + 1]) +
                          (maske[1, 0] * resim[y, x - 1]) + (maske[1, 1] * resim[y, x]) +
(maske[1, 2] * resim[y, x + 1]) +
                          (maske[2, 0] * resim[y + 1, x - 1]) + (maske[2, 1] * resim[y + 1,
x]) + (maske[2, 2] * resim[y + 1, x + 1]));
        }
    }
    return temp;
}
```

Ödev 1: Kenar bulma algoritması ile bir cismin kenarlarını tespit edin. Bu kapalı alan içinde kalan yüzeyin ağırlık merkezini bulan algoritmayı geliştirin.

Ödev 2: Aynı yöntemi kullanarak Cismin Alanı kaç piksel kare olduğunu tespit edin.

Ödev 3: Aynı yöntemle cimsin çevre uzunluğunun kaç piksel oldunu bulun

Ödev 4: Yukarıdaki ödevleri piksel uzunluğu ile mm yi ölçekleyerek yani görüntü içerisinde her mm nin kaç piksel ile ifade edildiğini bularak tüm hesaplamaları mm cinsinden bulun. Bunun için görüntü alanına bir cetvel yada kumpas koyarak ölçeği oradan çıkarabilirsiniz. Yada Ölçüsü bilinen bir şekil koyarak onunla kıyaslama yaparak program kendisi otomatik bulsun.