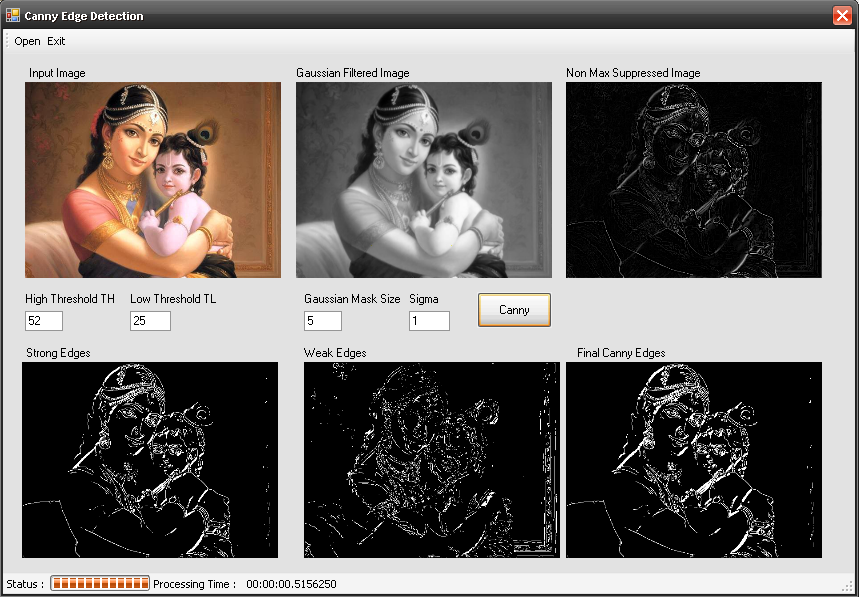
**Canny Edge Detection in C#**

****

The purpose of edge detection in general is to significantly reduce the amount of data in an image, while preserving the structural properties to be used for further image processing. Several algorithms exists, and this worksheet focuses on a particular one developed by John F. Canny (JFC) in 1986. Even though it is quite old, it has become one of the standard edge detection methods and it is still used in research.

The aim of JFC was to develop an algorithm that is optimal with regards to the following criteria:

1. **Detection**: The probability of detecting real edge points should be maximized while the probability of falsely detecting non-edge points should be minimized. This corresponds to maximizing the signal-to-noise ratio.

2. **Localization**: The detected edges should be as close as possible to the real edges.

3. **Number of responses**: One real edge should not result in more than one detected edge (one can argue that this is implicitly included in the first requirement).

With Canny’s mathematical formulation of these criteria, Canny’s Edge Detector is optimal for

a certain class of edges (known as step edges). A C# implementation of the algorithm is presented here.

The readers are advised to do more research on canny edge detection method for detailed theory.

**The Canny Edge Detection Algorithm**

The algorithm runs in 5 separate steps:

1. Smoothing: Blurring of the image to remove noise.

Implemented through Gaussian Filtering with Specific Kernel Size (N) and Gaussian Envelope Parameter Sigma.

The Gaussian Filter mask is generated by following Function :

private void GenerateGaussianKernel(int N, float S ,out int Weight)

{

float Sigma = S ;

float pi;

pi = (float)Math.PI;

int i, j;

int SizeofKernel=N;

float [,] Kernel = new float [N,N];

GaussianKernel = new int [N,N];

float[,] OP = new float[N, N];

float D1,D2;

D1= 1/(2\*pi\*Sigma\*Sigma);

D2= 2\*Sigma\*Sigma;

float min=1000;

for (i = -SizeofKernel / 2; i <= SizeofKernel / 2; i++)

{

for (j = -SizeofKernel / 2; j <= SizeofKernel / 2; j++)

{

Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] = ((1 / D1) \* (float)Math.Exp(-(i \* i + j \* j) / D2));

if (Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] < min)

min = Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j];

}

}

int mult = (int)(1 / min);

int sum = 0;

if ((min > 0) && (min < 1))

{

for (i = -SizeofKernel / 2; i <= SizeofKernel / 2; i++)

{

for (j = -SizeofKernel / 2; j <= SizeofKernel / 2; j++)

{

Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] = (float)Math.Round(Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] \* mult, 0);

GaussianKernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] = (int)Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j];

sum = sum + GaussianKernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j];

}

}

}

else

{

sum = 0;

for (i = -SizeofKernel / 2; i <= SizeofKernel / 2; i++)

{

for (j = -SizeofKernel / 2; j <= SizeofKernel / 2; j++)

{

Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] = (float)Math.Round(Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] , 0);

GaussianKernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j] = (int)Kernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j];

sum = sum + GaussianKernel[SizeofKernel / 2 + i, SizeofKernel / 2 + j];

}

}

}

//Normalizing kernel Weight

Weight= sum;

return;

}

Following subroutine removes noise by Gaussian Filtering

private int[,] GaussianFilter(int[,] Data)

{

GenerateGaussianKernel(KernelSize, Sigma,out KernelWeight);

int[,] Output = new int[Width, Height];

int i, j,k,l;

int Limit = KernelSize /2;

float Sum=0;

Output = Data; // Removes Unwanted Data Omission due to kernel bias while convolution

for (i = Limit; i <= ((Width - 1) - Limit); i++)

{

for (j = Limit; j <= ((Height - 1) - Limit); j++)

{

Sum = 0;

for (k = -Limit; k <= Limit; k++)

{

for (l = -Limit; l <= Limit; l++)

{

Sum = Sum + ((float)Data[i + k, j + l] \* GaussianKernel [Limit + k, Limit + l]);

}

}

Output[i, j] = (int)(Math.Round(Sum/ (float)KernelWeight));

}

}

return Output;

}

2. Finding gradients: The edges should be marked where the gradients of the image haslarge magnitudes.

Sobel X and Y Masks are used to generate X & Y Gradients of Image; next function implements differentiation using sobel Filter Mask

private float[,] Differentiate(int[,] Data, int[,] Filter)

{

int i, j,k,l, Fh, Fw;

Fw = Filter.GetLength(0);

Fh = Filter.GetLength(1);

float sum = 0;

float[,] Output = new float[Width, Height];

for (i = Fw / 2; i <= (Width - Fw / 2) - 1; i++)

{

for (j = Fh / 2; j <= (Height - Fh / 2) - 1; j++)

{

sum=0;

for(k=-Fw/2; k<=Fw/2; k++)

{

for(l=-Fh/2; l<=Fh/2; l++)

{

sum=sum + Data[i+k,j+l]\*Filter[Fw/2+k,Fh/2+l];

}

}

Output[i,j]=sum;

}

}

return Output;

}

3. Non-maximum suppression: Only local maxima should be marked as edges.

We find gradient direction and using these direction we perform non maxima suppression (Read “Digital Image Processing- by R Gonzales-Pearson Education)

// Perform Non maximum suppression:

// NonMax = Gradient;

for (i = 0; i <= (Width - 1); i++)

{

for (j = 0; j <= (Height - 1); j++)

{

NonMax[i, j] = Gradient[i, j];

}

}

int Limit = KernelSize / 2;

int r, c;

float Tangent;

for (i = Limit; i <= (Width - Limit) - 1; i++)

{

for (j = Limit; j <= (Height - Limit) - 1; j++)

{

if (DerivativeX[i, j] == 0)

Tangent = 90F;

else

Tangent = (float)(Math.Atan(DerivativeY[i, j] / DerivativeX[i, j]) \* 180 / Math.PI); //rad to degree

//Horizontal Edge

if (((-22.5 < Tangent) && (Tangent <= 22.5)) || ((157.5 < Tangent) && (Tangent <= -157.5)))

{

if ((Gradient[i, j] < Gradient[i, j + 1]) || (Gradient[i, j] < Gradient[i, j - 1]))

NonMax[i, j] = 0;

}

//Vertical Edge

if (((-112.5 < Tangent) && (Tangent <= -67.5)) || ((67.5 < Tangent) && (Tangent <= 112.5)))

{

if ((Gradient[i, j] < Gradient[i + 1, j]) || (Gradient[i, j] < Gradient[i - 1, j]))

NonMax[i, j] = 0;

}

//+45 Degree Edge

if (((-67.5 < Tangent) && (Tangent <= -22.5)) || ((112.5 < Tangent) && (Tangent <= 157.5)))

{

if ((Gradient[i, j] < Gradient[i + 1, j - 1]) || (Gradient[i, j] < Gradient[i - 1, j + 1]))

NonMax[i, j] = 0;

}

//-45 Degree Edge

if (((-157.5 < Tangent) && (Tangent <= -112.5)) || ((67.5 < Tangent) && (Tangent <= 22.5)))

{

if ((Gradient[i, j] < Gradient[i + 1, j + 1]) || (Gradient[i, j] < Gradient[i - 1, j - 1]))

NonMax[i, j] = 0;

}

}

}

4. Double thresholding: Potential edges are determined by thresholding.

5. Edge tracking by hysteresis: Final edges are determined by suppressing all edges that

are not connected to a very certain (strong) edge.

This is performed by a recursive function which performs double thresholding by two thresholds High Threshold(TH) and Low Threshold (TL) and 8-connectivity analysis

private void HysterisisThresholding(int[,] Edges)

{

int i, j;

int Limit= KernelSize/2;

for (i = Limit; i <= (Width - 1) - Limit; i++)

for (j = Limit; j <= (Height - 1) - Limit; j++)

{

if (Edges[i, j] == 1)

{

EdgeMap[i, j] = 1;

}

}

for (i = Limit; i <= (Width - 1) - Limit; i++)

{

for (j = Limit; j <= (Height - 1) - Limit; j++)

{

if (Edges[i, j] == 1)

{

EdgeMap[i, j] = 1;

Travers(i, j);

VisitedMap[i, j] = 1;

}

}

}

return;

}

private void Travers(int X, int Y)

{

if (VisitedMap[X, Y] == 1)

{

return;

}

//1

if (EdgePoints[X + 1, Y] == 2)

{

EdgeMap[X + 1, Y] = 1;

VisitedMap[X + 1, Y] = 1;

Travers(X + 1, Y);

return;

}

//2

if (EdgePoints[X + 1, Y - 1] == 2)

{

EdgeMap[X + 1, Y - 1] = 1;

VisitedMap[X + 1, Y - 1] = 1;

Travers(X + 1, Y - 1);

return;

}

//3

if (EdgePoints[X, Y - 1] == 2)

{

EdgeMap[X , Y - 1] = 1;

VisitedMap[X , Y - 1] = 1;

Travers(X , Y - 1);

return;

}

//4

if (EdgePoints[X - 1, Y - 1] == 2)

{

EdgeMap[X - 1, Y - 1] = 1;

VisitedMap[X - 1, Y - 1] = 1;

Travers(X - 1, Y - 1);

return;

}

//5

if (EdgePoints[X - 1, Y] == 2)

{

EdgeMap[X - 1, Y ] = 1;

VisitedMap[X - 1, Y ] = 1;

Travers(X - 1, Y );

return;

}

//6

if (EdgePoints[X - 1, Y + 1] == 2)

{

EdgeMap[X - 1, Y + 1] = 1;

VisitedMap[X - 1, Y + 1] = 1;

Travers(X - 1, Y + 1);

return;

}

//7

if (EdgePoints[X, Y + 1] == 2)

{

EdgeMap[X , Y + 1] = 1;

VisitedMap[X, Y + 1] = 1;

Travers(X , Y + 1);

return;

}

//8

if (EdgePoints[X + 1, Y + 1] == 2)

{

EdgeMap[X + 1, Y + 1] = 1;

VisitedMap[X + 1, Y + 1] = 1;

Travers(X + 1, Y + 1);

return;

}

//VisitedMap[X, Y] = 1;

return;

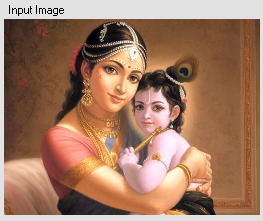
}

//Canny Class Ends

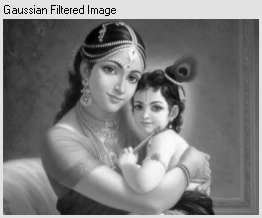
}

The Different Steps are Shown Here :

1. Original Image



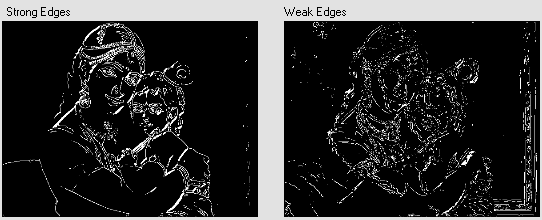
2. Gaussian Filtered Image (N=5, Sigma = 1)



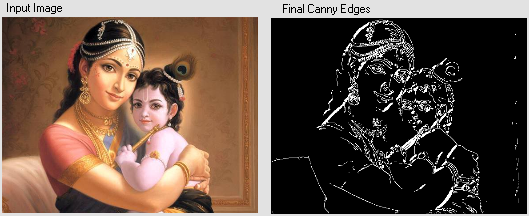
3. Non Maximum Suppressed image



4. Strong & Weak Edges detected



5. Final Edges – Hysteresis thresholding by TH=52, TL=25 (TH:TL == 2:1 or 3:1)



Supporting articles are included in the source directory.