In this post I continue to experiment with the de-blurring of images using the Wiener filter. The Wiener filter attempts to de-blur by assuming that the source signal is convolved with a blur kernel in the presence of noise.   I have also included the blur kernel as estimated by E. Simon in the code. I am including the de-blurring with 3 different blur kernel radii and different values for the Wiener constant K.  While the de-blurring is still a long way off there is some success.

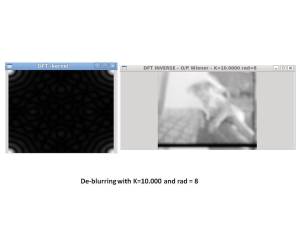
One of the reasons I have assumed a non-blind blur kernel and try to de-convolve with that. The Wiener filter tries to minimize the Mean Square Error (MSE)  which can be expressed as  
e(f) = E[X(f) – X1(f)]^2                                – (1)

where e(f) is the Mean Square Error(MSE) in the frequency domain, X(f) is the original image and X1(f) is the estimated signal which we get by de-convolving the Wiener filter with the observed blurred image i.e. and E[] is the expectation

X1(f) = G(f) \* Y(f)                                        -(2)  
where G(f) is the Wiener de-convolution filter and Y(f) is the observed blurred image  
Substituting (2) in (1) we get  
e(f) = E[X(f) – G(f) \*Y(f)]^2

If the above equation is solved we can effectively remove the blur.  
Feel free to post comments, opinions or ideas.

Watch this space …  
I will be back! Hasta la Vista!

[](https://gigadom.files.wordpress.com/2011/11/pica.jpg)  
[](https://gigadom.files.wordpress.com/2011/11/picb.jpg)[](https://gigadom.files.wordpress.com/2011/11/picd.jpg)

#include <stdio.h>  
#include <stdlib.h>  
#include “cxcore.h”  
#include “cv.h”  
#include “highgui.h”

#define kappa 10.0  
#define rad 8

int main(int argc, char \*\* argv)  
{  
int height,width,step,channels,depth;  
uchar\* data1;

CvMat \*dft\_A;  
CvMat \*dft\_B;

CvMat \*dft\_C;  
IplImage\* im;  
IplImage\* im1;

IplImage\* image\_ReB;  
IplImage\* image\_ImB;

IplImage\* image\_ReC;  
IplImage\* image\_ImC;  
IplImage\* complex\_ImC;  
CvScalar val;

IplImage\* k\_image\_hdr;  
int i,j,k;  
char str[80];  
FILE \*fp;  
fp = fopen(“test.txt”,”w+”);

int dft\_M,dft\_N;  
int dft\_M1,dft\_N1;

CvMat\* cvShowDFT1(IplImage\*, int, int,char\*);  
void cvShowInvDFT1(IplImage\*, CvMat\*, int, int,char\*);

im1 = cvLoadImage( “kutty-1.jpg”,1 );  
cvNamedWindow(“Original-color”, 0);  
cvShowImage(“Original-color”, im1);  
im = cvLoadImage( “kutty-1.jpg”, CV\_LOAD\_IMAGE\_GRAYSCALE );  
if( !im )  
return -1;

cvNamedWindow(“Original-gray”, 0);  
cvShowImage(“Original-gray”, im);

// Create a random noise matrix  
fp = fopen(“test.txt”,”w+”);  
int val\_noise[357\*383];  
for(i=0; i <im->height;i++){  
for(j=0;j<im->width;j++){  
fprintf(fp, “%d “,(383\*i+j));  
val\_noise[383\*i+j] = rand() % 128;  
}  
fprintf(fp, “\n”);  
}

CvMat noise = cvMat(im->height,im->width, CV\_8UC1,val\_noise);

// Add the random noise matric to the image  
cvAdd(im,&noise,im, 0);

cvNamedWindow(“Original + Noise”, 0);  
cvShowImage(“Original + Noise”, im);

cvSmooth( im, im, CV\_GAUSSIAN, 7, 7, 0.5, 0.5 );  
cvNamedWindow(“Gaussian Smooth”, 0);  
cvShowImage(“Gaussian Smooth”, im);

// Create a blur kernel  
IplImage\* k\_image;  
float r = rad;  
float radius=((int)(r)\*2+1)/2.0;

int rowLength=(int)(2\*radius);  
printf(“rowlength %d\n”,rowLength);  
float kernels[rowLength\*rowLength];  
printf(“rowl: %i”,rowLength);  
int norm=0; //Normalization factor  
int x,y;  
CvMat kernel;  
for(x = 0; x < rowLength; x++)  
for (y = 0; y < rowLength; y++)  
if (sqrt((x – (int)(radius) ) \* (x – (int)(radius) ) + (y – (int)(radius))\* (y – (int)(radius))) <= (int)(radius))  
norm++;  
// Populate matrix  
for (y = 0; y < rowLength; y++) //populate array with values  
{  
for (x = 0; x < rowLength; x++) {  
if (sqrt((x – (int)(radius) ) \* (x – (int)(radius) ) + (y – (int)(radius))  
\* (y – (int)(radius))) <= (int)(radius)) {  
//kernels[y \* rowLength + x] = 255;  
kernels[y \* rowLength + x] =1.0/norm;  
printf(“%f “,1.0/norm);  
}  
else{  
kernels[y \* rowLength + x] =0;  
}  
}  
}

/\*for (i=0; i < rowLength; i++){  
for(j=0;j < rowLength;j++){  
printf(“%f “, kernels[i\*rowLength +j]);  
}  
}\*/

kernel= cvMat(rowLength, // number of rows  
rowLength, // number of columns  
CV\_32FC1, // matrix data type  
&kernels);  
k\_image\_hdr = cvCreateImageHeader( cvSize(rowLength,rowLength), IPL\_DEPTH\_32F,1);  
k\_image = cvGetImage(&kernel,k\_image\_hdr);

height = k\_image->height;  
width = k\_image->width;  
step = k\_image->widthStep/sizeof(float);  
depth = k\_image->depth;

channels = k\_image->nChannels;  
//data1 = (float \*)(k\_image->imageData);  
data1 = (uchar \*)(k\_image->imageData);  
cvNamedWindow(“blur kernel”, 0);  
cvShowImage(“blur kernel”, k\_image);

dft\_M = cvGetOptimalDFTSize( im->height – 1 );  
dft\_N = cvGetOptimalDFTSize( im->width – 1 );

//dft\_M1 = cvGetOptimalDFTSize( im->height+99 – 1 );  
//dft\_N1 = cvGetOptimalDFTSize( im->width+99 – 1 );

dft\_M1 = cvGetOptimalDFTSize( im->height+3 – 1 );  
dft\_N1 = cvGetOptimalDFTSize( im->width+3 – 1 );

printf(“dft\_N1=%d,dft\_M1=%d\n”,dft\_N1,dft\_M1);

// Perform DFT of original image  
dft\_A = cvShowDFT1(im, dft\_M1, dft\_N1,”original”);  
//Perform inverse (check)  
//cvShowInvDFT1(im,dft\_A,dft\_M1,dft\_N1, “original”); – Commented as it overwrites the DFT

// Perform DFT of kernel  
dft\_B = cvShowDFT1(k\_image,dft\_M1,dft\_N1,”kernel”);  
//Perform inverse of kernel (check)  
//cvShowInvDFT1(k\_image,dft\_B,dft\_M1,dft\_N1, “kernel”);- Commented as it overwrites the DFT

// Multiply numerator with complex conjugate  
dft\_C = cvCreateMat( dft\_M1, dft\_N1, CV\_64FC2 );

printf(“%d %d %d %d\n”,dft\_M,dft\_N,dft\_M1,dft\_N1);

// Multiply DFT(blurred image) \* complex conjugate of blur kernel  
cvMulSpectrums(dft\_A,dft\_B,dft\_C,CV\_DXT\_MUL\_CONJ);  
//cvShowInvDFT1(im,dft\_C,dft\_M1,dft\_N1,”blur1″);

// Split Fourier in real and imaginary parts  
image\_ReC = cvCreateImage( cvSize(dft\_N1, dft\_M1), IPL\_DEPTH\_64F, 1);  
image\_ImC = cvCreateImage( cvSize(dft\_N1, dft\_M1), IPL\_DEPTH\_64F, 1);  
complex\_ImC = cvCreateImage( cvSize(dft\_N1, dft\_M1), IPL\_DEPTH\_64F, 2);  
printf(“%d %d %d %d\n”,dft\_M,dft\_N,dft\_M1,dft\_N1);

//cvSplit( dft\_C, image\_ReC, image\_ImC, 0, 0 );  
cvSplit( dft\_C, image\_ReC, image\_ImC, 0, 0 );

// Compute A^2 + B^2 of denominator or blur kernel  
image\_ReB = cvCreateImage( cvSize(dft\_N1, dft\_M1), IPL\_DEPTH\_64F, 1);  
image\_ImB = cvCreateImage( cvSize(dft\_N1, dft\_M1), IPL\_DEPTH\_64F, 1);

// Split Real and imaginary parts  
cvSplit( dft\_B, image\_ReB, image\_ImB, 0, 0 );  
cvPow( image\_ReB, image\_ReB, 2.0);  
cvPow( image\_ImB, image\_ImB, 2.0);  
cvAdd(image\_ReB, image\_ImB, image\_ReB,0);  
val = cvScalarAll(kappa);  
cvAddS(image\_ReB,val,image\_ReB,0);

//Divide Numerator/A^2 + B^2  
cvDiv(image\_ReC, image\_ReB, image\_ReC, 1.0);  
cvDiv(image\_ImC, image\_ReB, image\_ImC, 1.0);

// Merge Real and complex parts  
cvMerge(image\_ReC, image\_ImC, NULL, NULL, complex\_ImC);  
sprintf(str,”O/P Wiener – K=%6.4f rad=%d”,kappa,rad);

// Perform Inverse  
cvShowInvDFT1(im, (CvMat \*)complex\_ImC,dft\_M1,dft\_N1,str);

cvWaitKey(-1);  
return 0;  
}

CvMat\* cvShowDFT1(IplImage\* im, int dft\_M, int dft\_N,char\* src)  
{  
IplImage\* realInput;  
IplImage\* imaginaryInput;  
IplImage\* complexInput;  
CvMat\* dft\_A, tmp;  
IplImage\* image\_Re;  
IplImage\* image\_Im;  
char str[80];  
double m, M;  
realInput = cvCreateImage( cvGetSize(im), IPL\_DEPTH\_64F, 1);  
imaginaryInput = cvCreateImage( cvGetSize(im), IPL\_DEPTH\_64F, 1);  
complexInput = cvCreateImage( cvGetSize(im), IPL\_DEPTH\_64F, 2);  
cvScale(im, realInput, 1.0, 0.0);  
cvZero(imaginaryInput);  
cvMerge(realInput, imaginaryInput, NULL, NULL, complexInput);

dft\_A = cvCreateMat( dft\_M, dft\_N, CV\_64FC2 );  
image\_Re = cvCreateImage( cvSize(dft\_N, dft\_M), IPL\_DEPTH\_64F, 1);  
image\_Im = cvCreateImage( cvSize(dft\_N, dft\_M), IPL\_DEPTH\_64F, 1);

// copy A to dft\_A and pad dft\_A with zeros  
cvGetSubRect( dft\_A, &tmp, cvRect(0,0, im->width, im->height));  
cvCopy( complexInput, &tmp, NULL );  
if( dft\_A->cols > im->width )  
{  
cvGetSubRect( dft\_A, &tmp, cvRect(im->width,0, dft\_A->cols – im->width, im->height));  
cvZero( &tmp );  
}

// no need to pad bottom part of dft\_A with zeros because of  
// use nonzero\_rows parameter in cvDFT() call below  
cvDFT( dft\_A, dft\_A, CV\_DXT\_FORWARD, complexInput->height );  
strcpy(str,”DFT -“);  
strcat(str,src);  
cvNamedWindow(str, 0);

// Split Fourier in real and imaginary parts  
cvSplit( dft\_A, image\_Re, image\_Im, 0, 0 );

// Compute the magnitude of the spectrum Mag = sqrt(Re^2 + Im^2)  
cvPow( image\_Re, image\_Re, 2.0);  
cvPow( image\_Im, image\_Im, 2.0);  
cvAdd( image\_Re, image\_Im, image\_Re, NULL);  
cvPow( image\_Re, image\_Re, 0.5 );

// Compute log(1 + Mag)  
cvAddS( image\_Re, cvScalarAll(1.0), image\_Re, NULL ); // 1 + Mag  
cvLog( image\_Re, image\_Re ); // log(1 + Mag)

cvMinMaxLoc(image\_Re, &m, &M, NULL, NULL, NULL);  
cvScale(image\_Re, image\_Re, 1.0/(M-m), 1.0\*(-m)/(M-m));  
cvShowImage(str, image\_Re);  
return(dft\_A);  
}

void cvShowInvDFT1(IplImage\* im, CvMat\* dft\_A, int dft\_M, int dft\_N,char\* src)  
{  
IplImage\* realInput;  
IplImage\* imaginaryInput;  
IplImage\* complexInput;  
IplImage \* image\_Re;  
IplImage \* image\_Im;  
double m, M;  
char str[80];  
realInput = cvCreateImage( cvGetSize(im), IPL\_DEPTH\_64F, 1);  
imaginaryInput = cvCreateImage( cvGetSize(im), IPL\_DEPTH\_64F, 1);  
complexInput = cvCreateImage( cvGetSize(im), IPL\_DEPTH\_64F, 2);  
image\_Re = cvCreateImage( cvSize(dft\_N, dft\_M), IPL\_DEPTH\_64F, 1);  
image\_Im = cvCreateImage( cvSize(dft\_N, dft\_M), IPL\_DEPTH\_64F, 1);

//cvDFT( dft\_A, dft\_A, CV\_DXT\_INV\_SCALE, complexInput->height );  
cvDFT( dft\_A, dft\_A, CV\_DXT\_INV\_SCALE, dft\_M);  
strcpy(str,”DFT INVERSE – “);  
strcat(str,src);  
cvNamedWindow(str, 0);

// Split Fourier in real and imaginary parts  
cvSplit( dft\_A, image\_Re, image\_Im, 0, 0 );

// Compute the magnitude of the spectrum Mag = sqrt(Re^2 + Im^2)  
cvPow( image\_Re, image\_Re, 2.0);  
cvPow( image\_Im, image\_Im, 2.0);  
cvAdd( image\_Re, image\_Im, image\_Re, NULL);  
cvPow( image\_Re, image\_Re, 0.5 );

// Compute log(1 + Mag)  
cvAddS( image\_Re, cvScalarAll(1.0), image\_Re, NULL ); // 1 + Mag  
cvLog( image\_Re, image\_Re ); // log(1 + Mag)

cvMinMaxLoc(image\_Re, &m, &M, NULL, NULL, NULL);  
cvScale(image\_Re, image\_Re, 1.0/(M-m), 1.0\*(-m)/(M-m));  
//cvCvtColor(image\_Re, image, CV\_GRAY2RGBA);  
cvShowImage(str, image\_Re);  
}