4. Exercise - Given

File: main.cpp

```
main(int argc, char** argv)
```

- Loads image, path given in argv[1]
- Adds distortion: Gaussian blur (stddev in argv[3]) and
  Gaussian noise (SNR in argv[2])
- Calls restoration functions
- Saves restored images

File: dip4.cpp

```
Mat degradeImage(Mat& img, Mat& degradedImg, double filterDev, double snr)
```

`img`	input image
`degradedImage`	output image
`filterDev`	standard deviation of Gaussian blur
`snr`	signal-to-noise ratio
`return`	filter kernel used for blurring

- Adds Gaussian blur and Gaussian noise

4. Exercise – To Do

`Mat inverseFilter(Mat& degraded, Mat& filter)`

`degraded`	input image
`filter`	filter that caused distortion
`return`	restored image

- Applies (modified) inverse filter to restore image (e.g. $\epsilon = 0.05$)

`Mat wienerFilter(Mat& degraded, Mat& filter, double snr)`

`degraded`	input image
`filter`	filter that caused distortion
`snr`	signal-to-noise ratio
`return`	restored image

- Applies Wiener filter to restore image

<u>Note:</u>
- circShift(..)
- Proper usage of cv::dft(..), i.e. compressed output format
- Spectra are complex valued, i.e. 1/P is complex-valued!!
- Output image might contain large values. i.e. |out(x,y)| > 255
- Useful functions: cv::merge(..), cv::split(..), cv::threshold(..)

circShift

```
Mat Dip4::circShift(Mat& in, int dx, int dy){

   Mat out = in.clone();

   int x, y, new_x, new_y;

   for(y=0; y<in.rows; y++){

      // calulate new y-coordinate
      new_y = y + dy;
      if (new_y<0)
         new_y = new_y + in.rows;
      if (new_y>=in.rows)
        new_y = new_y - in.rows;

      for(x=0; x<in.cols; x++){
         // calculate new x-coordinate
         new_x = x + dx;
         if (new_x<0)
            new_x = new_x + in.cols;
         if (new_x>=in.cols)
            new_x = new_x - in.cols;
         out.at<float>(new_y, new_x) = in.at<float>(y, x);
      }
   }
   return out;
```

inverseFilter/wienerFilter

```
Mat Dip4::inverseFilter(Mat& degraded, Mat& filter){

   Mat tmp;

   // DFT image
   Mat dft_in;
   vector<Mat> i_cmpl;
   i_cmpl.push_back(degraded.clone());
   i_cmpl.push_back(Mat::zeros(degraded.rows, degraded.cols, CV_32FC1));
   merge(i_cmpl, dft_in);
   dft( dft_in, dft_in, DFT_COMPLEX_OUTPUT);
   split(dft_in, i_cmpl);

   // DFT kernel
   Mat dft_kernel = Mat::zeros( degraded.rows, degraded.cols, CV_32FC2);
   vector<Mat> f_cmpl;
   // preparation
   // copy kernel into new matrix
   tmp = Mat::zeros( degraded.rows, degraded.cols, CV_32FC1);
   Mat roi = tmp(Rect(0,0,filter.cols,filter.rows));
   filter.copyTo(roi);
   // center filterkernel
   tmp = circShift( tmp, -filter.cols/2, -filter.rows/2);
   // dft
   f_cmpl.push_back(tmp);
   f_cmpl.push_back(Mat::zeros(tmp.rows, tmp.cols, CV_32FC1));
   merge(f_cmpl, dft_kernel);
   dft( dft_kernel, dft_kernel, DFT_COMPLEX_OUTPUT );
   split(dft_kernel, f_cmpl);
```

InverseFilter

```
// define inverse filter
// get |F| = ( F_r^2 + F_i^2 )^0.5
Mat amp, phase;
cartToPolar(f_cmpl[0], f_cmpl[1], amp, phase);

// get max( |F| )
double minAmp, maxAmp, eps = 0.05;
minMaxLoc(amp, &minAmp, &maxAmp);

amp.setTo(1, amp < eps*maxAmp);
phase.setTo(0, amp < eps*maxAmp);
polarToCart(amp, phase, f_cmpl[0], f_cmpl[1]);

for(int y = 0; y < dft_kernel.rows; y++){
   for(int x = 0; x < dft_kernel.cols; x++){

      double a = i_cmpl[0].at<float>(y,x);
      double b = i_cmpl[1].at<float>(y,x);
      double c = f_cmpl[0].at<float>(y,x);
      double d = f_cmpl[1].at<float>(y,x);

      i_cmpl[0].at<float>(y,x) = (a*c+b*d)/(c*c + d*d);
      i_cmpl[1].at<float>(y,x) = (b*c-a*d)/(c*c + d*d);
   }
}
merge(i_cmpl, dft_in);

Mat out;
dft( dft_in, out, DFT_INVERSE + DFT_REAL_OUTPUT + DFT_SCALE);

return out;
}
```

wienerFilter

```
    for(int y = 0; y < dft_kernel.rows; y++){
       for(int x = 0; x < dft_kernel.cols; x++){

          double a = i_cmpl[0].at<float>(y,x);
          double b = i_cmpl[1].at<float>(y,x);
          double c = f_cmpl[0].at<float>(y,x);
          double d = f_cmpl[1].at<float>(y,x);

          double e = c / ( (c*c + d*d) + 1.0/(snr*snr));
          double f = -d / ( (c*c + d*d) + 1.0/(snr*snr));

          i_cmpl[0].at<float>(y,x) = a*e - b*f;
          i_cmpl[1].at<float>(y,x) = a*f + b*e;
       }
    }
    merge(i_cmpl, dft_in);

    Mat out;
    dft( dft_in, out, DFT_INVERSE + DFT_REAL_OUTPUT + DFT_SCALE);

    return out;

}
```