aberrationRendering.py

Laura K. Young

1 Rendering aberrated stimuli

1.1 Aberrated stimulus rendering in python

This iPython notebook contains an interactive example of using aberrationRendering.py to create aberrated stimuli. A description of the steps and the theory behind them are given in the supplementary material for the following article. Please cite it when you use the code:

Young, L. K., and Smithson, H. E. (2014), Critical band masking reveals the effects of optical distortions on the channel mediating letter identification. Frontiers in Psychology, 5:1060.

In [48]:
```
import aberrationRendering_final as aberrationRendering
import pylab
import numpy
from __future__ import division
```

1.2 Wavefront representation

First we specify the height (and width) of the array (default is 512), the number of radial orders to reconstruct (default is 5) and if the indexing scheme should be printed to the screen (default is False).

In [49]:
```
array_size = 512
n_radial_orders = 5
z = aberrationRendering.Zernike(array_size, order=n_radial_orders,
                                print_indices=True)
```

```
Index: n, m
 0:  0, 0
 1:  1,-1
 2:  1, 1
 3:  2,-2
 4:  2, 0
 5:  2, 2
 6:  3,-3
 7:  3,-1
 8:  3, 1
 9:  3, 3
```

```
10:  4,-4
11:  4,-2
12:  4, 0
13:  4, 2
14:  4, 4
15:  5,-5
16:  5,-3
17:  5,-1
18:  5, 1
19:  5, 3
20:  5, 5
```

The coordinates, single index mapping and the matrix of Zernike modes can be retrieved as follows:

In [50]:
```
r = z.r
theta = z.theta
indices = z.index_mapping

# Equation 2
zernike_matrix = z.zernike_matrix
```

As an example, we have chosen to display defocus, Z_2^0, which has a single index of (mode =) 4:

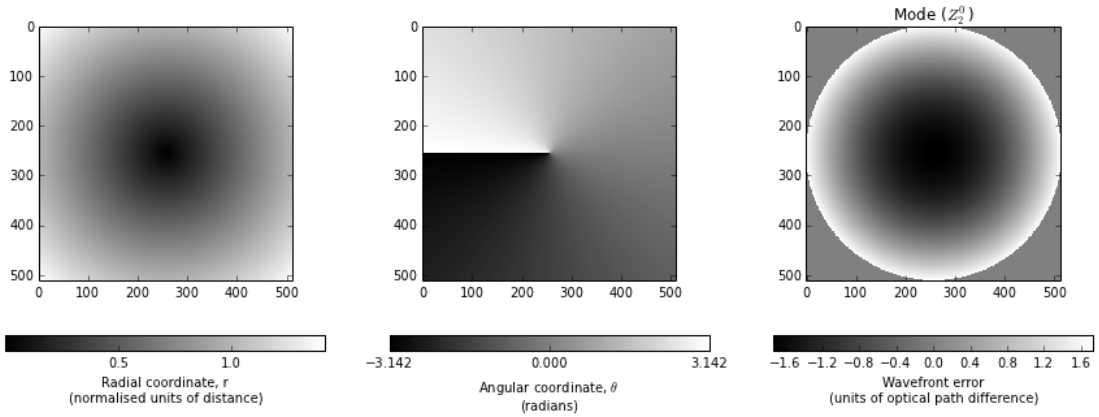
In [51]:
```
mode = 4

f = pylab.figure(1, figsize=(15, 5))

ax1 = f.add_subplot(131)
im1 = ax1.imshow(r, interpolation='nearest', cmap='gray')
cb1 = pylab.colorbar(im1, orientation='horizontal',
                     ticks=numpy.linspace(0.0, 1.5, 4))
cb1.set_label('Radial coordinate, r' + '\n(normalised units of distance)')

ax2 = f.add_subplot(132)
im2 = ax2.imshow(theta, interpolation='nearest', cmap='gray')
cb2 = pylab.colorbar(im2, orientation='horizontal',
                     ticks=numpy.linspace(-numpy.pi, numpy.pi, 3))
cb2.set_label(r'Angular coordinate, $\theta$' + '\n(radians)')

ax3 = f.add_subplot(133)
im3 = ax3.imshow(zernike_matrix[mode], interpolation='nearest',
                 cmap='gray')
cb3 = pylab.colorbar(im3, orientation='horizontal')
ax3.set_title('Mode ($Z_{%i}^{%i}$)' %(indices[mode,0], indices[mode,1]))
cb3.set_label(r'Wavefront error' + '\n(units of optical path difference)')
```



We can represent any wavefront error by computing the matrix multiplication of the Zernike matrix with the Zernike coefficient vector. In this example we have chosen 0.6 rms microns of defocus (Z_2^0, mode = 4) and 0.3 rms microns of coma (Z_3^1, mode = 8).

In [52]:
```
number_zernike_modes = zernike_matrix.shape[0]
zernike_coefficients = numpy.zeros((number_zernike_modes))
zernike_coefficients[4] = 0.6e-6
zernike_coefficients[8] = 0.3e-6

# Equation 1
wavefront_error = aberrationRendering.wavefront(zernike_coefficients,
                                                zernike_matrix)
```

N.b.: If a diffraction limited PSF is required, simply pass an array of zeros as the Zernike coefficients to the function that calculates the wavefront. Next we specify the pupil amplitude function,

In [53]:
```
amplitude_function = aberrationRendering.amplitudeFunctionCircle(array_size)
```

We then check that there are a sufficient number of pixels across the wavefront,

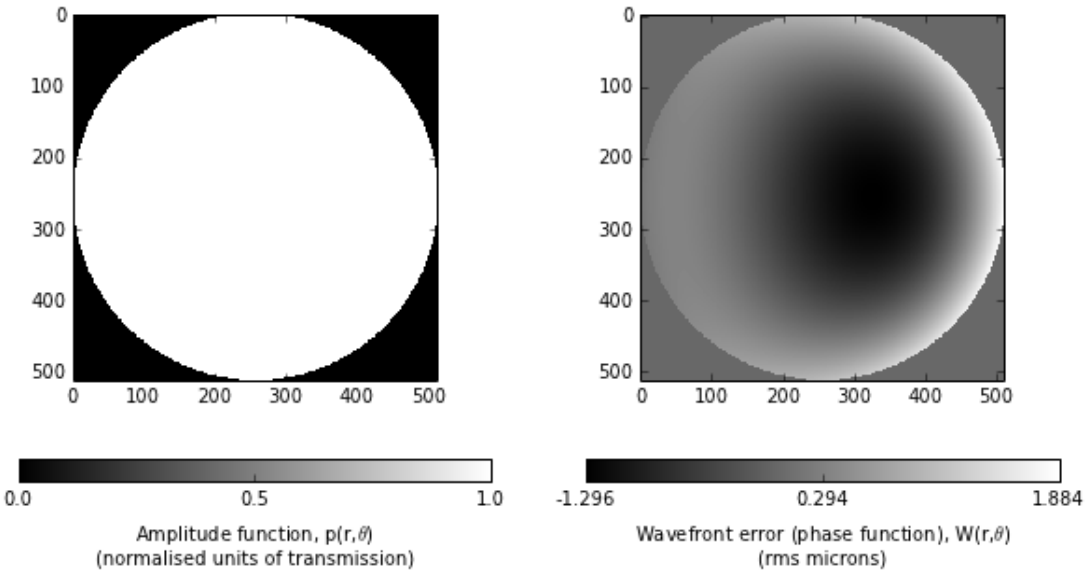
In [54]:
```
oversampling = 4
wavefront_gradient_check, psf_fov_check = \
aberrationRendering.checkSampling(wavefront_error, amplitude_function,
                                  array_size, oversampling)
```

and then display it:

In [55]:
```
f = pylab.figure(2, figsize=(10,5))

ax1 = f.add_subplot(121)
im1 = ax1.imshow(amplitude_function, interpolation='nearest',cmap='gray')
cb1 = pylab.colorbar(im1, orientation='horizontal',
                     ticks=numpy.linspace(0, 1.0, 3))
cb1.set_label(r'Amplitude function, p(r,$\theta$)' +
              '\n(normalised units of transmission)')

ax2 = f.add_subplot(122)
im2 = ax2.imshow(wavefront_error, interpolation='nearest', cmap='gray')
ticks = numpy.linspace(wavefront_error.min(), wavefront_error.max(), 3)
cb2 = f.colorbar(im2, orientation='horizontal',ticks=ticks)
ticks_microns = (ticks * 1.0e6).round(3)
cb2.ax.set_xticklabels(ticks_microns)
cb2.set_label(r'Wavefront error (phase function), W(r,$\theta$)' +
              '\n(rms microns)')
```



PSF generation

The PSF is the Fourier transform of the complex pupil function, computed using the following function:

In [56]:
```
wavelength = 550e-9
pupil_radius = 2.5e-3
psf = aberrationRendering.PSF(wavefront_error, amplitude_function,
                      wavelength=wavelength, oversampling=oversampling)
psf_diffraction_limited = aberrationRendering.PSF(zernike_matrix[0]*0.0,
    amplitude_function, wavelength=wavelength, oversampling=oversampling)
```

and to display it (note there may be aliasing effects associated with the size of the figure window, these should not be present in psf array):

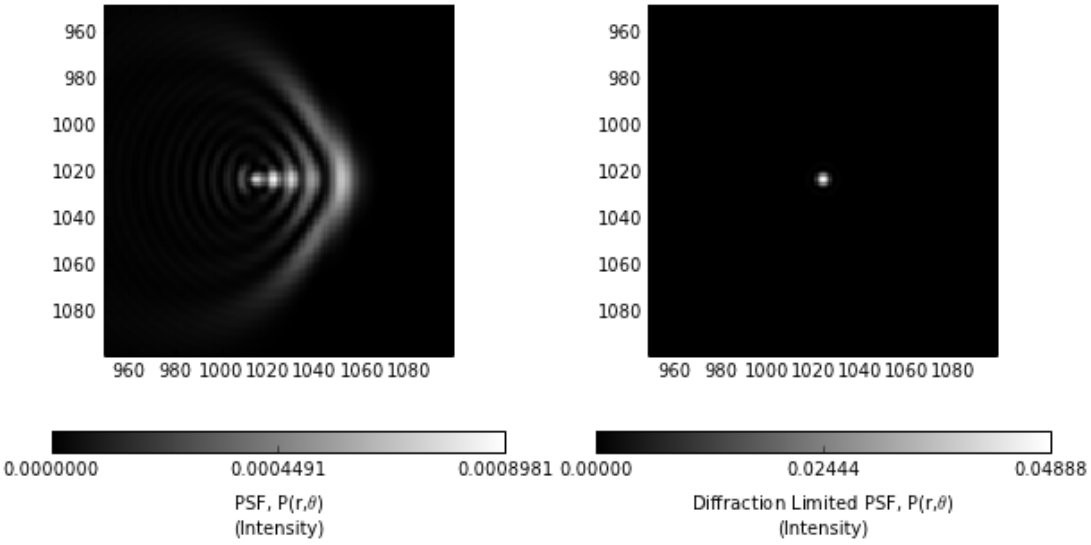
In [57]:
```
f = pylab.figure(3,figsize=(10, 5))

ax1 = f.add_subplot(121)
im1 = ax1.imshow(psf, interpolation='nearest', cmap='gray')
cb1 = pylab.colorbar(im1, orientation='horizontal',
                     ticks=numpy.linspace(psf.min(), psf.max(), 3))
cb1.set_label(r'PSF, P(r,$\theta$)' + '\n(Intensity)')
centre = psf.shape[0] // 2
# Just display the centre.
pylab.axis([centre - 75, centre + 75, centre + 75, centre - 75])

ax2 = f.add_subplot(122)
im2 = ax2.imshow(psf_diffraction_limited, interpolation='nearest',
                 cmap='gray')
cb1 = pylab.colorbar(im2, orientation='horizontal',
                     ticks=numpy.linspace(psf_diffraction_limited.min(),
                                          psf_diffraction_limited.max(), 3))
cb1.set_label(r'Diffraction Limited PSF, P(r,$\theta$)' + '\n(Intensity)')
centre = psf_diffraction_limited.shape[0] // 2
# Just display the centre.
pylab.axis([centre - 75, centre + 75, centre + 75, centre - 75])
```

Out [57]:

```
[949, 1099, 1099, 949]
```



1.3 Convolution of the PSF with an input intensity pattern

The pixel scale (in radians) in the PSF is calculated with the following function,

In [58]:
```
pixel_scale = aberrationRendering.psfPixelScale(wavelength=wavelength,
                    oversampling=oversampling, pupil_radius=pupil_radius)
pixel_scale_arcminutes = pixel_scale * 180.0 * 60.0 / numpy.pi
```

and the number of pixels required in the input intensity pattern (e.g. a 1° black square on a white background with a field of view across the entire array of 2°)

N.b.: An input intensity pattern can be created from an image file using the python Image module or similar.

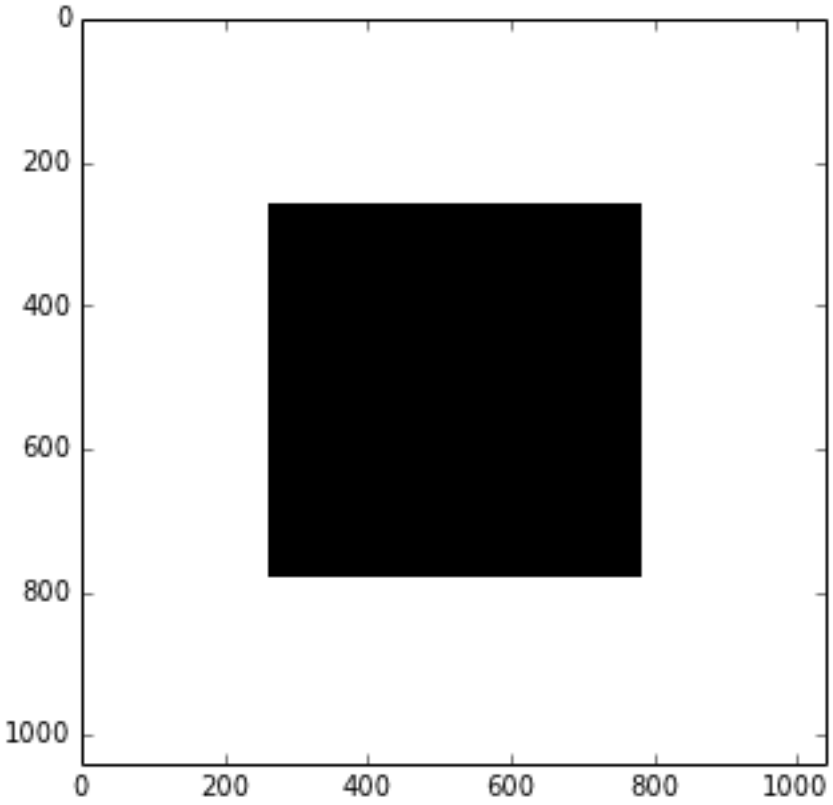
In [59]:
```
field_of_view_arcminutes = 120.0
n_pixels = aberrationRendering.numberPixels(field_of_view_arcminutes,
                wavelength=wavelength, oversampling=oversampling,
                pupil_radius=pupil_radius)
print 'The number of pixels required in input intensity pattern is: %i' \
                                                              %n_pixels
```

```
The number of pixels required in input intensity pattern is: 1040
```

We then create our input intensity pattern (here the input is 8-bit) with the correct number of pixels, which ensures the pixel scale is correct:

In [60]:
```
square = numpy.ones((n_pixels, n_pixels), numpy.uint8) * 255
square[n_pixels/4:n_pixels*3/4,n_pixels/4:n_pixels*3/4] = 0

f = figure(4, figsize=(5, 5))
ax = f.add_subplot(111)
im = ax.imshow(square, interpolation='nearest', cmap='gray')
```

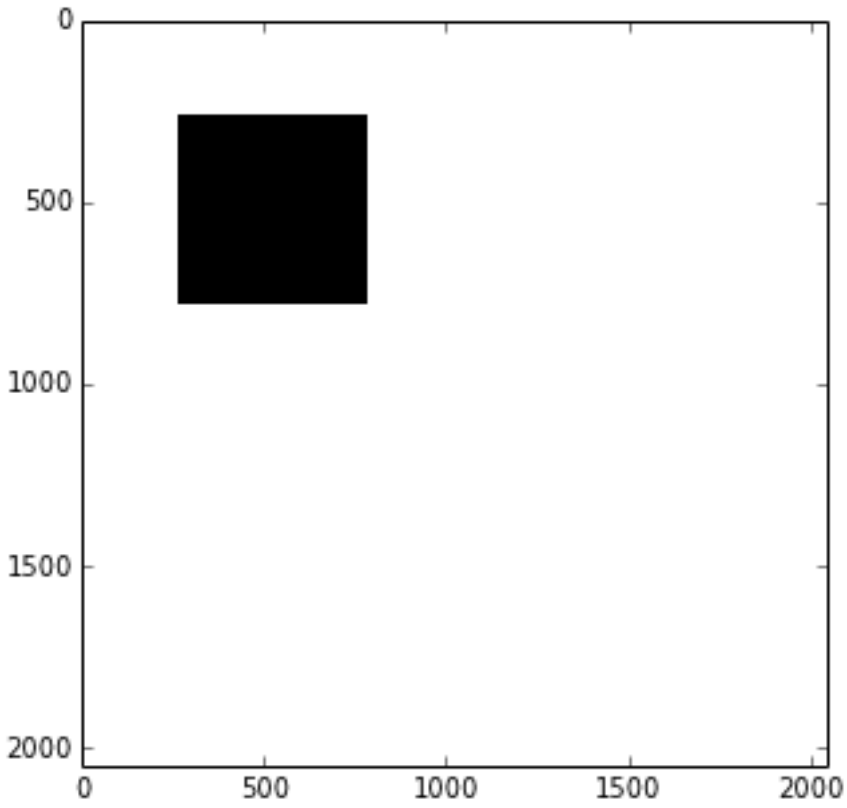


The input intensity pattern must have the same number of pixels (and therefore the same field of view) as the PSF so we pad it. This increases the field of view of the input intensity pattern and the number of pixels, so the pixel scale remains correct.

In [61]:
```
square_full = numpy.ones((psf.shape), numpy.uint8) * 255
square_full[:square.shape[0],:square.shape[1]] = square
```

We then display it:

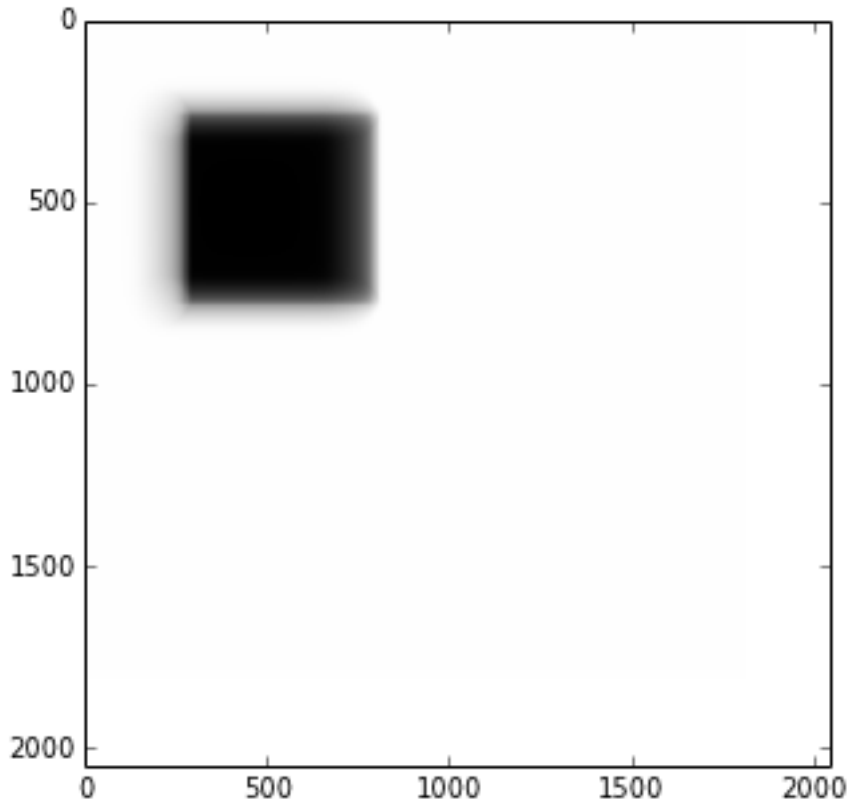
In [62]:
```
f = figure(5, figsize=(5, 5))
ax = f.add_subplot(111)
im = ax.imshow(square_full, interpolation='nearest', cmap='gray')
```



and then convolve it with the PSF

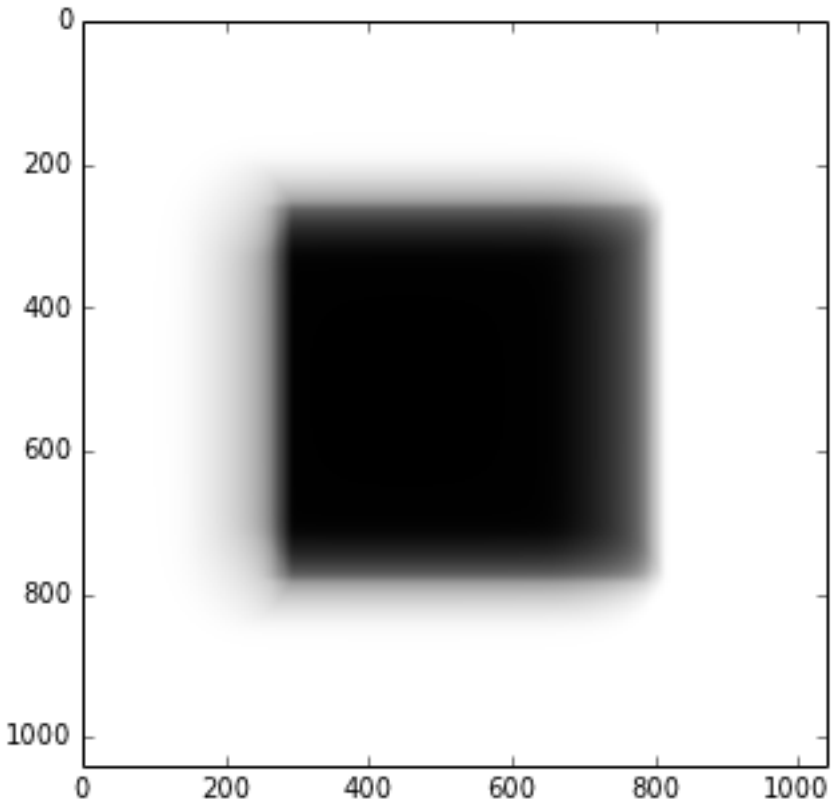
In [63]:
```
output_intensity_pattern = aberrationRendering.convolveImage(square_full,
                                                             psf)

f = figure(6, figsize=(5, 5))
ax = f.add_subplot(111)
im = ax.imshow(output_intensity_pattern, interpolation='nearest',
               cmap='gray')
```



and we can crop this back to the original field of view if required.

In [64]:
```
output_cropped = output_intensity_pattern[:square.shape[0],:square.shape[1]]
f = figure(7, figsize=(5, 5))
ax = f.add_subplot(111)
im = ax.imshow(output_cropped, interpolation='nearest', cmap='gray')
```



This final array will have the same data type (bit-depth) as the input image can be resized for the display. If you want an output that is an array os float, pass the input intensity pattern as an array of floats. The array can be saved as bitmap using the python Image module or similar.