Course reader: *Image noise*

- Image noise is a source of corruption and confusion in pictures, satellite data, microscope data, and so on.

- On the other hand, image "noise" or randomness can also be used purposefully, for example, to create textured landscapes in video games or cartoons.

- Noise can take several different forms, have different origins, and have several different distributions. This section introduces several common types of noise.

- "Image white noise" or "image static noise," also called "TV noise," is the simple extension of 1D white noise onto the plane — a 2D matrix of random numbers, and the value at each pixel is converted to a color value. It looks more like noise in grayscale than in color.

- "Checkerboard" noise comes in two flavors: *random* and *regular*:
  1. Random checkerboards have, well, a random black&white pattern. They can be created by thresholding and binarizing 2D random noise, where the threshold will determine the proportion of white vs. black squares.
  2. Regular checkerboards are like a chess board (or a checkers board). They can also have more intricate patterns. I showed one way to create them, by generating a matrix of increasing integers and testing whether each element is divisible by 2 (thus, odd or even).

- The term "Perlin noise" might be new to you, but I'm sure you've seen Perlin noise before. It's widely used to create mountains, clouds, oceans, and other image textures in computer-generated graphics.

- Perlin noise can be generated in any number of dimensions. I showed only 2D noise because Perlin images are most commonly used. But the principle is the same for any number of dimensions, including 1D time series.

- The essence of Perlin noise is to generate different random images with different levels of granularity or smoothness (technically different spatial frequencies; you can think of it like zooming in or zooming out). Then the Perlin image is a *weighted sum* of those images, with the smoother (lower-frequency) images being weighted more strongly.

- Perlin noise is generated by interpolating random noise matrices to different scales and then sub-sampling from them. This is not the most computationally efficient method for modern computers, but it was a good algorithm in the early 1980's.

- You can achieve the same qualitative effect through the inverse Fourier transform. This works by creating a power spectrum with desired characteristics (e.g., only low frequencies) and then inverting that to get back to the "space" domain.

- Technically, this procedure is called "2D circular convolution between random Fourier coefficients and a bank of Gaussian kernels" but that's quite a mouthful, so let's just called it 2D FFT noise. The mechanics and math behind this procedure are not very difficult, but it does require more background knowledge of the 2D Fourier transform.

- There is an important take-home message from Perlin and 2D-FFT noise, and also from pink and Brownian time series noise: Noise looks more "realistic" when lower frequencies have more energy than higher frequencies. That will come to no surprise to people who are familiar with the fractal aspects of nature: Most systems in nature (biological and non-biological) have 1/f-like power spectra.

Exercises

1. What does the amplitude spectrum of white image noise look like? Guess you answer first, based on the idea that 2D image noise is an extension of 1D time series noise. Then compute and display the amplitude spectrum in MATLAB.

2. In the smooth random noise example, you can make the noise look like a sandy beach by adding random normal noise. You'll need to max-value-normalize the smoothed map first, and it will look better if you set the colormap to "pink."

3. You can create a checkerboard-like image using the function -1^{j+k}, where j and k are the row and column indices. Try it!

4. You can also create a checkerboard-like image using the formula i^{jk}, where $i = \sqrt{-1}$, that is, the imaginary operator. This matrix is complex (that is, each matrix element is complex and contains a real part and an imaginary part), and you can make an image of (1) the real part, (2) the imaginary part, (3) the sum of the real and imaginary parts, and (4) the difference of the real and imaginary parts. Bonus question for those familiar with the Fourier transform: Why is the magnitude of every matrix element $= 1$?

5. One of the reasons why programming is such a powerful tool for learning math is that you can use the code to gain insight and intuition into a problem even if you are not comfortable staring at equations. In the 2D-FFT noise example, change the boundaries for the Gaussian widths (variable `widths`). Keep running the code with different boundaries until you feel like you have some intuition for what effects the limits and the number of elements will have on the final image result.

Answers

1. It also looks like white noise. Code:

PYTHON

```
import matplotlib.pyplot as plt
import scipy.fftpack
import numpy as np

img = np.random.randn(100,100)
plt.imshow(abs(scipy.fftpack.fft2(img)))
plt.show()
```

MATLAB

```
imagesc(abs(fft2(randn(100))));
```

2. Here's the code:

PYTHON

```
img = img/np.max(img) # max-val norm.
plt.imshow(img+np.random.rand(n,n),cmap='pink')
plt.show()
```

MATLAB

```
img = img./max(img(:)); % max-val norm.
imagesc(img+randn(n)) % add noise
colormap pink
```

3. The coding can be a bit tricky because you have to watch out for proper parentheses:

PYTHON

```
for i in range(40):
for j in range(40):
m[i,j] = (-1)**(i+j)
```

MATLAB

```
for i=1:40
   for j=1:40
      m(i,j) = (-1)^(i+j);
   end
end
imagesc(m)
```

4. The resulting images look really neat:

PYTHON

```
import matplotlib.pyplot as plt
import numpy as np

m = np.zeros((40,40),dtype=complex)
for i in range(40):
for j in range(40):
m[i,j] = 1j**(i*j) # or try (-1i)

plt.subplot2grid((2,2),(0,0)), plt.imshow(np.real(m))
plt.subplot2grid((2,2),(0,1)), plt.imshow(np.imag(m))
plt.subplot2grid((2,2),(0,0)), plt.imshow(np.real(m))
plt.subplot2grid((2,2),(1,0)), plt.imshow(np.real(m)+np.imag(m))
plt.subplot2grid((2,2),(1,1)), plt.imshow(np.real(m)-np.imag(m))
plt.show()
```

MATLAB

```
for i=1:40
   for j=1:40
      m(i,j) = 1i^(i*j); % or try (-1i)
   end
end
subplot(221), imagesc(real(m))
subplot(222), imagesc(imag(m))
subplot(223), imagesc(real(m)+imag(m))
subplot(224), imagesc(real(m)-imag(m))
```

5. There's no "answer" here. Just play with the code and enjoy!