RandomMatrixTheoryHintsPython

January 9, 2024

Random Matrix Theory

(Sethna, "Entropy, Order Parameters, and Complexity", ex. 1.6, developed with Piet Brouwer)

This is an ipython notebook. This hints file is unusually detailed, but only covers the computational parts (a & f) of the exercise: don't neglect the analytical portions! If you are familiar with Mathematica, feel free to skip parts you don't need, or start your own notebooks.

Lectures about Python, useful both for beginners and experts, can be found at http://scipy-lectures.github.io.

I recommend installing the Anaconda distribution.

Go to https://www.continuum.io/downloads

Pick your operating system (Windows, Mac OS X, Linux)

Follow instructions for Python 3.5.

To make it run much faster (probably necessary for some assignments):

Be sure not to pay for it! Get an academic subscription, using your Cornell (.edu) e-mail, from https://www.continuum.io/anaconda-academic-subscriptions-available * Set up an account, using your Cornell e-mail * Go to My Settings under your name in the upper right hand side of the Anaconda Cloud window, and * Go to Add Ons, the lowest menu entry in the left-hand column * Download the licenses, and follow the installation instructions link just above the MKL licence button (or go to http://docs.continuum.io/advanced-installation)

Open the notebook by (1) copying this file into a directory, (2) in that directory typing:

```
ipython notebook
```

and (3) selecting the notebook. In later exercises, if you do animations, you may choose to not have the plots inline.

Like Mathematica, you type in commands and then hit 'Shift Return' to execute them.

```
[ ]:  a = 3.
      print(a)
```

Python is not a complete scientific environment. Functions like log, sin, and exp are provided by external packages. The plotting package pylab supports many of these; '%matplotlib inline' puts the graphs into the notebook. We usually want more functionality than that, so we import

a package called 'numpy'. We'll also be using some of their linear algebra routines, which are in a sub-package 'linalg'.

```
[1]: %matplotlib inline
     from matplotlib.pyplot import plot, show, hist, title
     from numpy import *
     from numpy.linalg import eigvalsh
```

One of the most active and unusual applications of ensembles is *random matrix theory*, used to describe phenomena in nuclear physics, mesoscopic quantum mechanics, and wave phenomena. Random matrix theory was invented in a bold attempt to describe the statistics of energy level spectra in nuclei. In many cases, the statistical behavior of systems exhibiting complex wave phenomena – almost any correlations involving eigenvalues and eigenstates – can be quantitatively modeled using ensembles of matrices with completely random, uncorrelated entries!

The most commonly explored ensemble of matrices is the Gauusian orthogonal ensemble (GOE). Generating a member takes two steps: * Generate an NxN matrix whose elements are independent Gaussian distributions of zero mean and standard deviation $\sigma = 1$. * Add the matrix to its transpose to symmetrize it.

Start by finding out how Python generates random numbers. Type ?random to find out about scipy's random number generators.

Try typing 'random.random()' a few times. Try calling it with an integer argument. Use 'hist' (really pylab.hist) to make a histogram of 1000 numbers generated by random.random. Is the distribution Gaussian, uniform, or exponential? The flag 'density' tells us to not just count the number of occurences in each bin, but to divide by the width of the bin and the number of counts (to approximate a normalized probability distribution).

```
[ ]: ?random
     print(random.random())
     print(random.random(6))
     hist(random.random(1000), density=True);
```

Use random.standard_normal to generate a Gaussian distribution of one thousand random numbers. Check that it is a standard Gaussian

$$\rho(x) = \frac{1}{\sqrt{2\pi}\sigma}e^{-x^2/2\sigma^2}$$

by using 'hist'.

```
[ ]: print(random.standard_normal(...))
     hist(random.standard_normal(...), density=...);
     x = arange(-3,3,0.01)
     plot(x,(1./sqrt(2*pi))*exp(-x**2/2),'r')
     show()
```

One final matrix operation. To generate a symmetric matrix of the GOE form, we'll want to add a matrix to its transpose. Generate a 2x3 matrix of normally distributed random numbers, and print it and its transpose.

```
[ ]: m = random.standard_normal((2,3));
     print(m)
     print(transpose(m))
```

Now define a function GOE(N) (using 'def') which generates a GOE NxN matrix.

```
[ ]: def GOE(N):
         """
         Creates an NxN element of the Gaussian Orthogonal Ensemble,
         by creating an NxN matrix of Gaussian random variables
             using the random array function random.standard_normal([shape])
             with [shape] = (N,N).
         and adding it to its transpose (applying transpose to the matrix)
         #
         Typing GOE(4) a few times, check it's symmetric and random
         """
         m = random. ...((...,...))
         m = m + transpose(m)
         return m
```

Test it by generating a 4x4 matrix. Is it symmetric?

```
[ ]: GOE(4)
```

Now write a function GOE_Ensemble(num, N) generating an ensemble (a long list with 'num' elements) of GOE NxN matrices. Call GOE(N) to fill the list.

```
[ ]: def GOE_Ensemble(num, N):
         """
         Creates a list "ensemble" of length num of NxN GOE matrices.
         #
         You can start with an empty list, and fill it with 'append'
             ensemble = []
             for n in range(num):
                 ensemble.append(GOE(N))
             return ensemble
         or you can use the convenient 'list comprehension'
             ensemble = [GOE(N) for n in range(num)]
         #
         Check GOE_Ensemble(3,2) gives 3 2x2 symmetric matrices
         """
         ensemble = ...
         return ensemble
```

Here you may want to generate an ensemble, and check if the diagonal elements and the off-diagonal elements have different standard deviations, as you calculated in part (c). (This difference is part of the definition of the GOE ensemble, and is needed to make the ensemble rotation invariant.)

One of the most striking properties that large random matrices share is the distribution of level

splittings.

To look at eigenvalues of our symmetric matrices, we use 'eigvalsh'. We'll want them sorted. And we'll want to find the difference between the two middle-most eigenvalues.

Notice, in python, arrays are indexed lambda = [lambda[0], ..., lambda[N-1]], so the middle two are N/2 and N/2-1. Note that in the new version of python dividing integers $4/2 = 2.0$ gives a floating point number, so we need to use N//2 to get an integer. (This is a good thing: having $2/10 = 0$ causes lots of problems when you write routines that don't know whether their inputs are floats or ints.)

```python
N=4
midpoint = int(N/2)
mat = GOE(N)
print(mat)
eig = sort(eigvalsh(mat))
print(eig)
print(eig[N//2]-eig[N//2-1])
```

```python
def CenterEigenvalueDifferences(ensemble):
    """
    For each symmetric matrix in the ensemble, calculates the difference
    between the two center eigenvalues of the ensemble, and returns the
    differences as an array.
    #
    Given an ensemble of symmetric matrices,
    finds their size N, using the first member ensemble[0]
        len(m) gives the number of rows in m
    starts with an empty list of differences
    for each m in ensemble
      finds the eigenvalues
        eigvals(m) gives the eigenvalues of m
      sorts them
        sort(e) sorts a list from smallest to largest
      appends eigenvalue[N/2] - eigenvalue[N/2-1] to the list
         (Note: python lists go from 0 ... N-1, so for N=2 this
          gives the difference between the only two eigenvalues)
    #
    Check
        ensemble = GOE_Ensemble(3,2)
        CenterEigenvalueDifferences(ensemble)
    gives three positive numbers, that look like eigenvalues of the 2x2 matrices
    """
    # Size of matrix
    N = len(...)
    diffs = []
    # Note that you can 'iterate' not only over integers, but over elements of
    # a list (for mat in ensemble: ...)
```

4

```
        for mat in ensemble:
            eigenvalues = sort(...)
            diffs.append(...)
        return diffs
```

Check that

```
    ensemble = GOE_Ensemble(3,2)
    CenterEigenvalueDifferences(ensemble)
```

gives three positive numbers, that look like eigenvalue differences of 2x2 matrices, Check that

```
    hist(CenterEigenvalueDifferences(ensemble), bins=30, density=True);
```

gives three spikes at the three eigenvalue differences.

```
[ ]: ensemble = GOE_Ensemble(...)
     print(CenterEigenvalueDifferences(...))
     hist(CenterEigenvalueDifferences(...), bins=50, density=True);
```

Now try it with 10000 or more. Notice the smooth distribution, with a linear tail near zero.

```
[ ]: M = 10000
     N = 2
     ensemble = GOE_Ensemble(M, N);
     diffs = CenterEigenvalueDifferences(ensemble)
     hist(diffs, bins=50, density=True);
```

What is this dip in the eigenvalue splitting probability near zero? It is called *level repulsion*.

Wigner's surmise (meaning 'guess') was that this distribution was universal, that is, independent of the ensemble of matrices. Clearly another ensemble, with all our matrix elements multiplied by two, will have eigenvalue differences twice as big. Wigner surmised that the eigenvalue differences, divided by the mean of the differences would be universal. Plot these normalized centered differences against your analytical prediction for 2x2 matrices (see text), which should be
$\rho_{Wigner}(s) = (\pi s/2) \exp(-\pi s^2/4)$

```
[ ]: hist(diffs/mean(diffs), bins=50, normed=True);
     s = arange(0.,3.,0.01)
     rhoWigner = ...
     plot(s, rhoWigner, 'r');
     title('2x2 matrix eig diffs')
```

Try N=4 and N=10; do they look similar?

```
[ ]: M = 10000
     N = 4
     ensemble = ...
     diffs = ...
     hist(...);
```

```
    plot(...);
    title(...)
```

```
[ ]:  M = 10000
      N = 10
      ensemble = ...
      diffs = ...
      hist(...);
      plot(...);
      title(...)
```

Note that we're typing the same commands several times. This is a bad programming practice (a 'bad smell') – whenever writing the same code more than a couple of lines long, write it as a subroutine. That way, when you fix an error, it'll get fixed everywhere you use it.

```
[ ]:  def CompareEnsembleWigner(ensemble):
          diffs = ...
          hist(...);
          plot(...)

      CompareEnsembleWigner(ensemble)
```

Your 2x2 formula is pretty good, but turns out to be up to 2% off for larger matrices; Wigner was wrong about that. But he was correct that the eigenvalue splitting is universal, if you consider large matrices.

We can test this by trying a different ensemble. For example, let's try random symmetric matrices filled with +-1. In the past, we've created these matrices the hard way. But let's just take the signs of the matrix elements generated by the GOE ensemble!

```
[ ]:  sign(GOE_Ensemble(4,3))
```

Define a PM1 ensemble (Plus or Minus 1)

```
[ ]:  def PM1_Ensemble(num, N):
          """
          Generates a +-1 ensemble, as for GOE_Ensemble above.
          Use with CompareEnsembleWigner to test that N=2 looks different
          from the Wigner ensemble, but larger N looks close to the GOE ensemble.
          #
          This is a powerful truth: the eigenvalue differences of very general classes
          of symmetric NxN matrices all have the same probability distribution
          as N goes to infinity. This is what we call universality.
          """
          return sign(...)
```

Try comparing the splittings for 2x2 matrices with the prediction.

```
[ ]:  CompareEnsembleWigner(PM1_Ensemble(10000,2))
```

Is it well approximated by the Wigner surmise? Why not? (Hint: how many different symmetric 2x2 matrices of ± 1 are there?)

Try it with 4x4 and 10x10 matrices.

```
[ ]: CompareEnsembleWigner(...)
```

10x10 should work better, except for a funny bump at zero.

```
[ ]: CompareEnsembleWigner(...)
```

The GOE ensemble has some nice statistical properties (see text).

This is our first example of an *emergent symmetry*. Many different ensembles of symmetric matrices, as the size N goes to infinity, have eigenvalue and eigenvector distributions that are invariant under orthogonal transformations *even though the original matrix ensemble did not have this symmetry*. Similarly, rotational symmetry emerges in random walks on the square lattice as the number of steps N goes to infinity, and also emerges on long length scales for Ising models at their critical temperatures.