Circulant-Matrices

September 7, 2017

```
In [1]: using PyPlot, Interact
```

1 Circulant Matrices

In this lecture, I want to introduce you to a new type of matrix: **circulant** matrices. Like Hermitian matrices, they have orthonormal eigenvectors, but unlike Hermitian matrices we know *exactly* what their eigenvectors are! Moreover, their eigenvectors are closely related to the famous Fourier transform and Fourier series. Even more importantly, it turns out that circulant matrices and the eigenvectors lend themselves to **incredibly efficient** algorithms called FFTs, that play a central role in much of computational science and engineering.

Consider a system of n identical masses m connected by springs k, sliding around a *circle* without friction. Similar to lecture 26, the vector $\vec{s}$ of displacements satifies $m\frac{d^2\vec{s}}{dt^2} = -kA\vec{s}$, where A is the $n \times n$ matrix:

$$A = \begin{pmatrix} 2 & -1 & & & & & -1 \\ -1 & 2 & -1 & & & & \\ & -1 & 2 & -1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -1 & 2 & -1 & \\ & & & & -1 & 2 & -1 \\ -1 & & & & & -1 & 2 \end{pmatrix}$$

(This matrix is real-symmetric and, less obviously, positive semidefinite. So, it should have orthogonal eigenvectors and real eigenvalues $\lambda \geq 0$.)

For example, if $n = 7$:

```
In [2]: A = [ 2 -1  0  0  0  0 -1
             -1  2 -1  0  0  0  0
              0 -1  2 -1  0  0  0
              0  0 -1  2 -1  0  0
              0  0  0 -1  2 -1  0
              0  0  0  0 -1  2 -1
             -1  0  0  0  0 -1  2]
```

https://github.com/stevengj/1806-spring17/raw/master/lectures/cyc

Figure 1: a ring of springs

```
Out[2]: 7×7 Array{Int64,2}:
  2  -1   0   0   0   0  -1
 -1   2  -1   0   0   0   0
  0  -1   2  -1   0   0   0
  0   0  -1   2  -1   0   0
  0   0   0  -1   2  -1   0
  0   0   0   0  -1   2  -1
 -1   0   0   0   0  -1   2
```

This matrix has a very special pattern: *every row is the same as the previous row, just shifted to the right by 1* (wrapping around "cyclically" at the edges). That is, each row is a circular shift of the first row.

This is called a circulant matrix. A 4×4 circulant matrix looks like:

$$C = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & c_0 & c_1 & c_2 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & c_2 & c_3 & c_0 \end{pmatrix}$$

The general form of an $n \times n$ circulant matrix C is:

$$C = \begin{pmatrix} c_0 & c_1 & c_2 & \cdots & c_{n-1} \\ c_{n-1} & c_0 & c_1 & c_2 & \cdots \\ c_{n-2} & c_{n-1} & c_1 & \cdots & \\ \ddots & \ddots & \ddots & \ddots & \ddots \\ c_1 & c_2 & \cdots & c_{n-1} & c_0 \end{pmatrix}$$

When working with circulant matrix, it is convenient to number entries from 0 to $n-1$ rather than from 1 to n!

1.1 Multiplying by circulant matrices: Convolutions

Suppose we have an $n \times n$ circulant matrix C that we want to multiply by a vector $x = (x_0, x_1, \ldots, x_n)$. It turns out that the result is a very special kind of operation:

$$y = Cx = \begin{pmatrix} c_0 & c_1 & c_2 & \cdots & c_{n-1} \\ c_{n-1} & c_0 & c_1 & c_2 & \cdots \\ c_{n-2} & c_{n-1} & c_0 & \cdots & \\ \ddots & \ddots & \ddots & \ddots & \ddots \\ c_1 & c_2 & \cdots & c_{n-1} & c_0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

Let's write down a formula for the entries of y:

$$y_0 = c_0x_0 + c_1x_1 + c_2x_2 + \cdots y_1 = c_{n-1}x_0 + c_0x_1 + c_1x_2 + \cdots y_3 = c_{n-2}x_0 + c_{n-1}x_1 + c_0x_2 + \cdots$$

Can you see the pattern? This is one of those cases that is actually clearer if we write out the summation:

$$y_k = \sum_{j=0}^{n} c_{j-k}x_j$$

There is a slight problem with this formula: the subscript $j-k$ can be < 0! No problem: we just *interpret the subscript periodically*, i.e. we let $c_{-1} = c_{n-1}$, $c_{-2} = c_{n-2}$, and so on. Equivalently, we define $c_{j \pm n} = c_j$. (We could say that the subscripts are modulo n.)

Multiplying by a circulant matrix is equivalent to a very famous operation called a circular convolution. Convolution operations, and hence circulant matrices, show up in lots of applications: **digital signal processing**, **image compression**, **physics/engineering simulations**, **number theory** and **cryptography**, and so on.

2 Eigenvectors of circulant matrices

One amazing property of circulant matrices is that **the eigenvectors are always the same**. The eigen-*values* are different for each C, but since we know the eigenvectors they are easy to diagonalize.

We can actually see one eigenvector right away. Let's call it $x^{(0)}$:

$$x^{(0)} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

This is an eigenvector because multiplying $Cx^{(0)}$ **simply sums each row of C**. But since each row of C contains the same entries (just in a different order), the sum is the same:

$$Cx^{(0)} = \underbrace{(c_0 + c_1 + \cdots + c_{n-1})}_{\lambda_0} x^{(0)}$$

Thus, one of the eigenvalues λ_0 of C is simply the sum of the row entries.

How can we find the other eigenvectors?

2.1 Roots of unity

The eigenvectors are simple to write down in terms of a very special value: a primitive root of unity:

$$\omega_n = e^{\frac{2\pi i}{n}}$$

The quantity ω_n has the very special property that $\omega_n^n = e^{2\pi i} = 1 = \omega_n^0$, but no smaller power equals 1. Therefore, $\omega_n^{j+n} = \omega_n^j \omega_n^n = \omega_n^j$: the **exponents of ω[U+2099] are periodic**. (Just like the c_j!)

For example, let's plot the powers of ω_7:

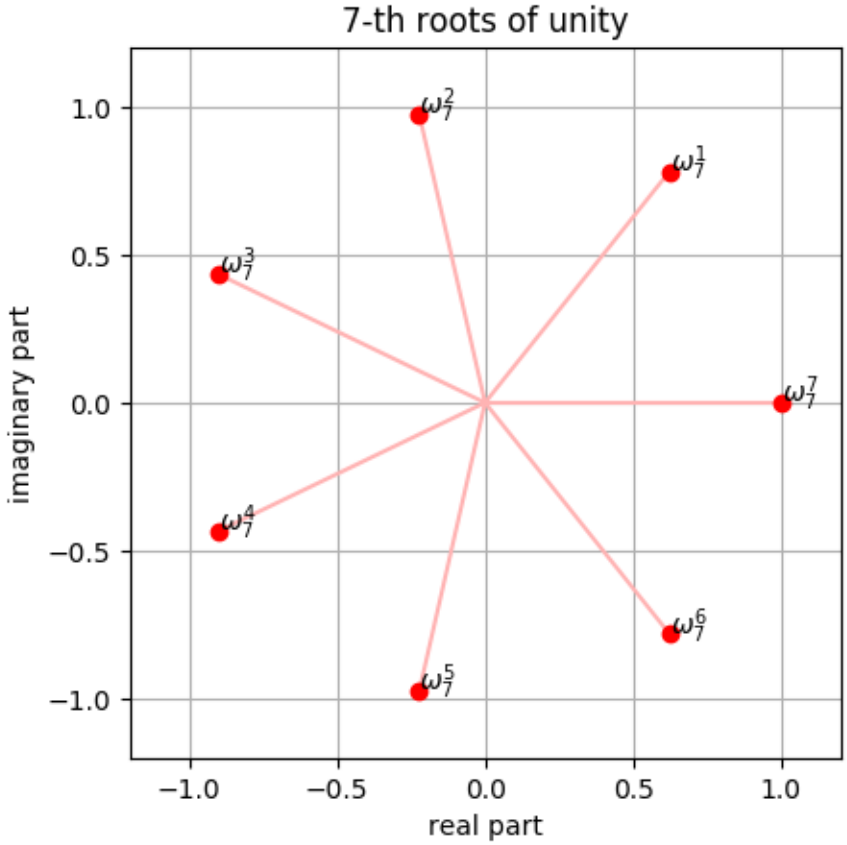
```
In [3]: ω7 = exp(2π*im / 7)
```

```
Out[3]: 0.6234898018587336 + 0.7818314824680298im
```

```
In [4]: fig = figure()
        @manipulate for n in slider(1:20, value=7)
            ω = exp(2π*im/n)
            withfig(fig) do
                for j = 1:n
                    z = ω ^ j
                    plot([0,real(z)], [0,imag(z)], ls="solid", color=(1,.7,.7))
                    plot(real(z), imag(z), "ro")
                    text(real(z), imag(z), "\$\\omega_{$n}^{$j}\$")
                end
                axis("square")
                grid()
                xlabel("real part")
                ylabel("imaginary part")
                title("$n-th roots of unity")
                xlim(-1.2,1.2)
                ylim(-1.2,1.2)
            end
        end
```

```
Interact.Slider{Int64}(Signal{Int64}(7, nactions=1),"",7,1:20,"horizontal",true,"d",true)
```

Out[4]:



They are called "roots of unity" because ω_n^j for $j = 0, \ldots, n-1$ (or $1, \ldots, n$) are *all* the solutions z to

$$z^n = 1.$$

2.2 Eigenvectors: The discrete Fourier transform (DFT)

In terms of ω_n, the eigenvectors of a circulant matrix are easy: the **k-th eigenvector** $x^{(k)}$ $(k = 0, \ldots, n-1)$ for **any n×n circulant matrix** is simply

$$x^{(k)} = \begin{pmatrix} \omega_n^{0k} \\ \omega_n^{1k} \\ \omega_n^{2k} \\ \vdots \\ \omega_n^{(n-1)k} \end{pmatrix}$$

Therefore, the matrix F whose columns are the eigenvectors is:

$$F = \begin{pmatrix} x^{(0)} & x^{(1)} & \cdots & x^{(n-1)} \end{pmatrix}$$

with entries

$$\boxed{F_{jk} = x_j^{(k)} = \omega_n^{jk} = e^{\frac{2\pi i}{n} jk}}.$$

Multiplying a vector by F is called a discrete Fourier transform (DFT). This is one of the most important matrices in the world! (It is sort of a finite, computer-friendly analogue to a Fourier series if you've seen those before.)

Before we show this, let's try it:

```
In [5]: # define a function to create the n×n matrix F for any n:
        F(n) = [exp((2π*im/n)*j*k) for j=0:n-1, k=0:n-1]

Out[5]: F (generic function with 1 method)
```

The 2×2 and 4×4 DFT matrices F are quite simple, for example

$$F_{2\times 2} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$F_{4\times 4} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

```
In [6]: round.(F(4))

Out[6]: 4×4 Array{Complex{Float64},2}:
         1.0+0.0im   1.0+0.0im   1.0+0.0im   1.0+0.0im
         1.0+0.0im   0.0+1.0im  -1.0+0.0im  -0.0-1.0im
         1.0+0.0im  -1.0+0.0im   1.0-0.0im  -1.0+0.0im
         1.0+0.0im  -0.0-1.0im  -1.0+0.0im   0.0+1.0im
```

Let's check that it diagonalizes our 7×7 mass-and-spring matrix A from earlier. We should have $F^{-1}AF = \Lambda$:

```
In [7]: round.(inv(F(7)) * A * F(7), 3) # F⁻¹AF = Λ, rounded to 3 digits

Out[7]: 7×7 Array{Complex{Float64},2}:
          0.0-0.0im   -0.0-0.0im   -0.0-0.0im  …    0.0-0.0im   -0.0-0.0im
         -0.0-0.0im  0.753-0.0im    0.0+0.0im       0.0-0.0im    0.0+0.0im
         -0.0+0.0im    0.0-0.0im  2.445+0.0im      -0.0+0.0im    0.0+0.0im
          0.0+0.0im    0.0+0.0im    0.0-0.0im       0.0-0.0im    0.0+0.0im
          0.0-0.0im    0.0+0.0im    0.0-0.0im      -0.0+0.0im   -0.0+0.0im
         -0.0+0.0im    0.0-0.0im    0.0-0.0im  …  2.445-0.0im   -0.0-0.0im
          0.0+0.0im    0.0+0.0im   -0.0+0.0im      -0.0-0.0im  0.753+0.0im
```

Compare the diagonal entries to the eigenvalues:

```
In [8]: eigvals(A)

Out[8]: 7-element Array{Float64,1}:
         -9.41142e-16
          0.75302
          0.75302
          2.44504
          2.44504
          3.80194
          3.80194

In [9]: diag(inv(F(7)) * A * F(7)) # diagonal entries
```

```
Out[9]: 7-element Array{Complex{Float64},1}:
             0.0-1.2326e-32im
         0.75302-5.32581e-17im
         2.44504+2.87857e-16im
         3.80194+1.90437e-16im
         3.80194+2.12572e-16im
         2.44504-1.93109e-16im
         0.75302+4.73147e-17im
```

Yup!

Since A is real-symmetric, you may wonder why the eigenvectors are not real. But they *could* be *chosen* real. For a real-symmetric circulant matrix, the real and imaginary parts of the eigenvectors are themselves eigenvectors. This is why most of the eigenvalues come in pairs! (The only eigenvalues that don't come in pairs correspond to eigenvectors $x^{(k)}$ that are purely real, e.g. $x^{(0)} = (1, 1, \ldots, 1)$.) These real and imaginary eigenvectors turn out to correspond to a discrete cosine transform (DCT) and a discrete sine transform (DST).

2.3 Derivation and eigenvalues

Why does this work? It's easy to see if we take our formula from above for Cx and multiply it by an eigenvector. Let $y = Cx^{(k)}$. Then the ℓ-th component is:

$$y_\ell = \sum_{j=0}^{n-1} c_{j-\ell} \omega_n^{jk} = \omega_n^{\ell k} \sum_{j=0}^{n-1} c_{j-\ell} \omega_n^{(j-\ell)k}$$

But the remaining sum is now **independent of** ℓ: because both c_j and ω_n^j are periodic in j, all $j \to j - \ell$ does is to re-arrange the numbers being summed (a circular shift), so you get the **same sum**. And $\omega_n^{\ell k} = x^{(k)}$, so we have:

$$Cx^{(k)} = \lambda_k x^{(k)}$$

where

$$\lambda_k = \sum_{j=0}^{n-1} c_j \omega_n^{jk}$$

But if we define a vector $\hat{c} = (\lambda_0, \lambda_1, \ldots, \lambda_{n-1})$, then

$$\hat{c} = Fc$$

That is, the **eigenvalues are the DFT of c** (where c = first row of C).

Let's check it:

```
In [10]: F(7) * A[:,1] # DFT of first row/column of A
```

```
Out[10]: 7-element Array{Complex{Float64},1}:
                      0.0+0.0im
          0.75302+1.11022e-16im
          2.44504-1.11022e-16im
          3.80194-6.66134e-16im
          3.80194-8.88178e-16im
          2.44504-2.22045e-16im
          0.75302+8.88178e-16im
```

```
In [11]: eigvals(A)
```

```
Out[11]: 7-element Array{Float64,1}:
          -9.41142e-16
           0.75302
           0.75302
           2.44504
           2.44504
           3.80194
           3.80194
```

Yup, they match!

3 Unitarity

The DFT matrix F is special in many ways. It is symmetric, but *not* Hermitian, so its eigenvalues are *not* real. However, it has **orthogonal columns**:

```
In [12]: round.(F(7)' * F(7), 3)
```

```
Out[12]: 7×7 Array{Complex{Float64},2}:
          7.0+0.0im  -0.0-0.0im  -0.0+0.0im  ...   0.0+0.0im   0.0+0.0im   0.0+0.0im
         -0.0+0.0im   7.0+0.0im  -0.0-0.0im     -0.0+0.0im   0.0+0.0im   0.0+0.0im
         -0.0-0.0im  -0.0+0.0im   7.0+0.0im     -0.0+0.0im   0.0+0.0im   0.0+0.0im
          0.0-0.0im  -0.0-0.0im  -0.0-0.0im      0.0+0.0im  -0.0+0.0im  -0.0+0.0im
          0.0-0.0im  -0.0-0.0im  -0.0-0.0im      7.0+0.0im   0.0+0.0im  -0.0+0.0im
          0.0-0.0im   0.0-0.0im   0.0-0.0im  ...   0.0-0.0im   7.0+0.0im  -0.0-0.0im
          0.0-0.0im   0.0-0.0im   0.0-0.0im     -0.0-0.0im  -0.0+0.0im   7.0+0.0im
```

(It is a straightforward exercise to show this, e.g. using the geometric-series summation formula.)

The columns are orthogonal but not orthonormal because they have length $\sqrt{n}$. But this means that if we divide by their length, then:

$$\frac{1}{\sqrt{n}}F$$

is a **unitary** matrix. Equivalently:

$$F^{-1} = \frac{1}{n}F^H = \frac{1}{n}\bar{F},$$

where we have used the fact that $F^T = F$. This is the **inverse discrete Fourier transform (IDFT)**.

Note that this means that **every circulant matrix C has orthogonal eigenvectors** (the columns of F). (Even if the matrix C is not Hermitian or one of the similar cases we have seen so far!)

4 Fast Fourier transforms (FFTs)

The product Fx for a vector x is the DFT of x. At first glance, it seems like it would require $\sim n^2$ operations (n dot products) like any other matrix–vector multiplication.

One of the most amazing facts of computational science is that it is possible to compute the DFT Fx in only $\sim n \log n$ operations (and $\sim n$ storage), by a fast Fourier transform (FFT) algorithm. FFT algorithms mean that DFTs and circulant matrices become practical to deal with even for huge n, and are are the **core of a huge number of practical computational algorithms**.

(From a linear-algebra viewpoint, it turns out that the "dense" matrix F factors into a **product** of $\sim \log n$ **sparse** matrices.)

FFTs aren't too complicated to understand (the simplest ones can be derived in a few lines of algebra), but they are a bit outside the scope of 18.06. But `fft(x)` functions are built-in to Julia and every other computational-science environment:

Figure 2: FFTW logo

```
In [13]: fft(A[:,1]) # computes the same thing as F(7) * A[:,1], but much faster

Out[13]: 7-element Array{Complex{Float64},1}:
              0.0+0.0im
          0.75302+0.0im
          2.44504+0.0im
          3.80194+0.0im
          3.80194+0.0im
          2.44504+0.0im
          0.75302+0.0im
```

But now that we know that F diagonalizes any circulant matrix, it leads to an amazing fact: **you can multiply Cx for any circulant matrix in ~ n log n operations and ~ n work**.

Specifically, $C = F\Lambda F^{-1} = F\frac{\Lambda}{n}\bar{F}$, so to multiply Cx we just need to:

1. Multiply $\hat{x} = \bar{F}x$. This can be done by an (inverse) FFT in $\sim nlogn$ operations.
2. Multiply each component of $\hat{x}$ by the eigenvalues λ_k/n. The eigenvalues can be computed by multiply F by the first row of C, which can be done by an FFT in $\sim n \log n$ operations.
3. Multiply by F via another FFT in $\sim n \log n$ operations.

This means that **circulant matrices are perfectly suited to iterative solver algorithms** (e.g. the power method or steepest-descent), just like sparse matrices!

4.1 A personal note

FFTs are near and dear to your instructor's heart — 20 years ago, partially for my final project in Prof. Alan Edelman's class 18.337, I developed a library for FFTs along with a friend of mine. Being arrogant MIT graduate students, we called it FFTW: The Fastest Fourier Transform in the West. Fortunately, FFTW somewhat lived up to its name, and it is now the FFT library in Matlab, in Julia, and in many other software packages.