AMSC/CMSC 460 Computational Methods, Fall 2007
UNIT 4: Matrix Computations

Dianne P. O'Leary

Matrix Computations

Read: Chapter 5. Skip: Sections 5.3, 5.4.2, and 5.5.

In solving linear systems, we work with matrices, so your book takes time out, in Chapter 5, to make sure we are comfortable with Matlab and matrices.

The Plan:

- Some ways to define matrices.
- Some basic operations on matrices.
- Sparse matrices.
- Some ways to measure the size of a matrix or vector.
- Fast matrix algorithms

Some ways to define matrices

Some ways to define matrices

1. We may be given a recipe for each element in the matrix.

   Example: A Hilbert matrix $\mathbf{A}$ of size 5×5, with element (i, j) equal to

   $$a_{ij} = \frac{1}{i + j - 1}.$$

```
n=5;
A = zeros(n,n); % optional
for i=1:n,
   for j=1:n,
       A(i,j) = 1/(i+j-1);
   end
end
```

   □

2. We may be given a recipe for each column or row of the matrix.

   Example: A Vandermonde matrix $\mathbf{A}$ is defined by a vector of elements $x_1, \ldots, x_n$. Its first column is all ones. Each later column is the preceding one times this vector.

```
n = length(x);
V(:,1) = ones(n,1);
for j=2:n,
   V(:,j) = x.*V(:,j-1);
end
```

   □

3. We may be told that the matrix has some sparsity, a significant number of zero elements.

   Example: A diagonal matrix

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 6 \end{bmatrix}$$

   can be defined by

```
D = diag([1 2 4 6]);
```

   □

   Example: A tridiagonal matrix

$$\mathbf{T} = \begin{bmatrix} 1 & 3 & 0 & 0 \\ 5 & 2 & 7 & 0 \\ 0 & 9 & 4 & 8 \\ 0 & 0 & 6 & 6 \end{bmatrix}$$

   can be defined by

```
T = diag([1 2 4 6]) + diag([5 9 6],-1) + diag([3 7 8], 1);
```

   □

4. We may build up larger matrices from smaller ones.

   Example: We can construct the matrix

$$\mathbf{B} = \begin{bmatrix} 1 & 3 & 0 & 0 & 1 & 0 & 0 & 0 \\ 5 & 2 & 7 & 0 & 0 & 2 & 0 & 0 \\ 0 & 9 & 4 & 8 & 0 & 0 & 4 & 0 \\ 0 & 0 & 6 & 6 & 0 & 0 & 0 & 6 \\ 1 & 0 & 0 & 0 & 1 & 3 & 0 & 0 \\ 0 & 1 & 0 & 0 & 5 & 2 & 7 & 0 \\ 0 & 0 & 1 & 0 & 0 & 9 & 4 & 8 \\ 0 & 0 & 0 & 1 & 0 & 0 & 6 & 6 \end{bmatrix}$$

   by noticing that it is composed from smaller blocks:

$$\mathbf{B} = \begin{bmatrix} \mathbf{T} & \mathbf{D} \\ \mathbf{I} & \mathbf{T} \end{bmatrix}.$$

   So one way to generate it is `B = [T, D; I, T]; []`

Some basic operations on matrices

Some basic operations on matrices

Here we'll consider two operations:

- Matrix-vector multiplication,
- Matrix-matrix multiplication.

Matrix-vector Multiplication

Suppose we have an $m \times n$ matrix $\mathbf{A}$ and an $n \times 1$ vector $\mathbf{x}$, and we wish to form $\mathbf{y} = \mathbf{A}\mathbf{x}$.

- The vector $\mathbf{y}$ is defined by dot products between rows of $\mathbf{A}$ and $\mathbf{x}$. Expressed element-by-element, the algorithm is

```
[m,n]=size(A);
y = zeros(m,1);
for i=1:m,
   for j=1:n,
        y(i) = y(i) + A(i,j)*x(j);
   end
end
```

- Expressed row-by-row, the algorithm is

```
[m,n]=size(A);
% The following command could be left out,
% but it is more efficient to put it in.
y = zeros(m,1);   % optional
for i=1:m,
     y(i) = A(i,:) * x;
end
```

- We can also express $\mathbf{Ax}$ in a column-oriented way, using saxpys:

$$\mathbf{Ax} = \mathbf{A}(:,1)x_1 + \mathbf{A}(:,2)x_2 + \ldots + \mathbf{A}(:,n)x_n\,.$$

  This can be implemented as

```
[m,n]=size(A);
y = zeros(m,1);
for j=1:n,
    y = y + A(:,j)*x(j);
end
```

- There is also a one-liner that does the job:

```
y = A * x;
```

The importance of exploiting zeros

Suppose our matrix is upper triangular:

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & 0 & \times \end{bmatrix}$$

(where elements marked "0" are really zero, and elements marked "$\times$" can take any value, possibly different for each position.)

If we used any of our hand-coded algorithms above, we would do 36 multiplications.

But this is a lot of wasted work, since many of the multiplications involve zero. How many? 1+2+3+4+5 = 15, nearly half!

It is important to take advantage of these zeros in order to reduce the work. See Section 5.2.2 for some examples.

Matrix-Matrix Multiplication

Again, there are many implementations:

- element-by-element.
- using dot products.
- using saxpys.
- using outer products.
- using a one-liner.

Example: How outer products can be used

$$
\begin{aligned}
\mathbf{AB} &= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \\
&= \begin{bmatrix} 1 \\ 4 \end{bmatrix} \begin{bmatrix} 1 & 4 & 7 \end{bmatrix} + \begin{bmatrix} 2 \\ 5 \end{bmatrix} \begin{bmatrix} 2 & 5 & 8 \end{bmatrix} + \begin{bmatrix} 3 \\ 6 \end{bmatrix} \begin{bmatrix} 3 & 6 & 9 \end{bmatrix} \\
&= \mathbf{A}(:,1) * \mathbf{B}(1,:) + \mathbf{A}(:,2) * \mathbf{B}(2,:) + \mathbf{A}(:,3) * \mathbf{B}(3,:)
\end{aligned}
$$

[]

Sparse matrices

Sparse matrices

A sparse matrix is one that has enough zeros that it is worthwhile to exploit them.

See Section 5.2.4.

Some ways to measure the size of a matrix or vector

Vector Norms

There are three convenient ways to measure the size of a vector. Here are the definitions:

- The Euclidean norm:

$$\|\mathbf{x}\|_2 = (x_1^2 + \ldots + x_n^2)^{1/2}\,.$$

  This is the one we are most familiar with.

  Unquiz: For $n = 2$, draw the set of vectors with Euclidean norm equal to 1.

- The one-norm, nicknamed the "Manhattan norm":

$$\|\mathbf{x}\|_1 = |x_1| + \ldots + |x_n|\,.$$

  This is cheaper to compute.

  Unquiz: For $n = 2$, draw the set of vectors with one-norm equal to 1.

- The infinity-norm:

$$\|\mathbf{x}\|_\infty = \max_{i=1,\ldots,n} |x_i|\,.$$

  This just measures the largest component.

  Unquiz: For $n = 2$, draw the set of vectors with infinity-norm equal to 1.

The norms are all related:

$$\begin{aligned} \|\mathbf{x}\|_\infty \leq \quad & \|\mathbf{x}\|_1 \quad \leq n\|\mathbf{x}\|_\infty, \\ \|\mathbf{x}\|_\infty \leq \quad & \|\mathbf{x}\|_2 \quad \leq \sqrt{n}\|\mathbf{x}\|_\infty. \end{aligned}$$

Matrix Norms

There are related ways to measure the norm of a matrix:. Here are the definitions:

$$
\begin{aligned}
\|\mathbf{A}\|_2 &= \max_{\|\mathbf{x}\|_2=1} \|\mathbf{A}\mathbf{x}\|_2 \\
\|\mathbf{A}\|_1 &= \max_{\|\mathbf{x}\|_1=1} \|\mathbf{A}\mathbf{x}\|_1 \\
&= \max_{j=1,\ldots,n} \sum_{i=1}^{m} |a_{ij}| \\
\|\mathbf{A}\|_\infty &= \max_{\|x\|_1=1} \|\mathbf{A}\mathbf{x}\|_\infty \\
&= \max_{i=1,\ldots,n} \sum_{j=1}^{m} |a_{ij}|
\end{aligned}
$$

The one-norm of a matrix is the max absolute column sum.
The infinity-norm of a matrix is the max absolute row sum.

There is no easy formula for the 2-norm of a matrix; it is the maximum singular value of the matrix, the square root of the largest eigenvalue of $\mathbf{A}\mathbf{A}^T$.

There is one other commonly used way to measure size of a matrix. If we stack the columns of $\mathbf{A}$ to make a vector of length mn, we could take its 2-norm:

$$\|\mathbf{A}\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{i,j}|^2 \right)^{1/2}.$$

This is called the Frobenius norm of the matrix.

Norms are computed in Matlab by the statements

```
norm(A,2)
norm(A,1)
norm(A,inf)
norm(A,'fro')
```

If you don't specify the second argument, Matlab uses 2.

Fast Matrix Algorithms

Fast Matrix Algorithms: The Fast Fourier Transform

Suppose we have taken n observations of some physical phenomenon, for instance, sunspot activity.

The discrete Fourier Transform (DFT) is a way to break the vector $\mathbf{x}$ of n observations into its frequency components, to determine, for instance, whether there is a cycle of 11 years in the observations.

This transform can be expressed as multiplication of $\mathbf{x}$ by a particular Vandermonde matrix $\mathbf{F}$ with parameters

$$1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1},$$

where

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

and $i = \sqrt{-1}$. Thus, $\omega = \cos(2\pi/n) - i\sin(2\pi/n)$.

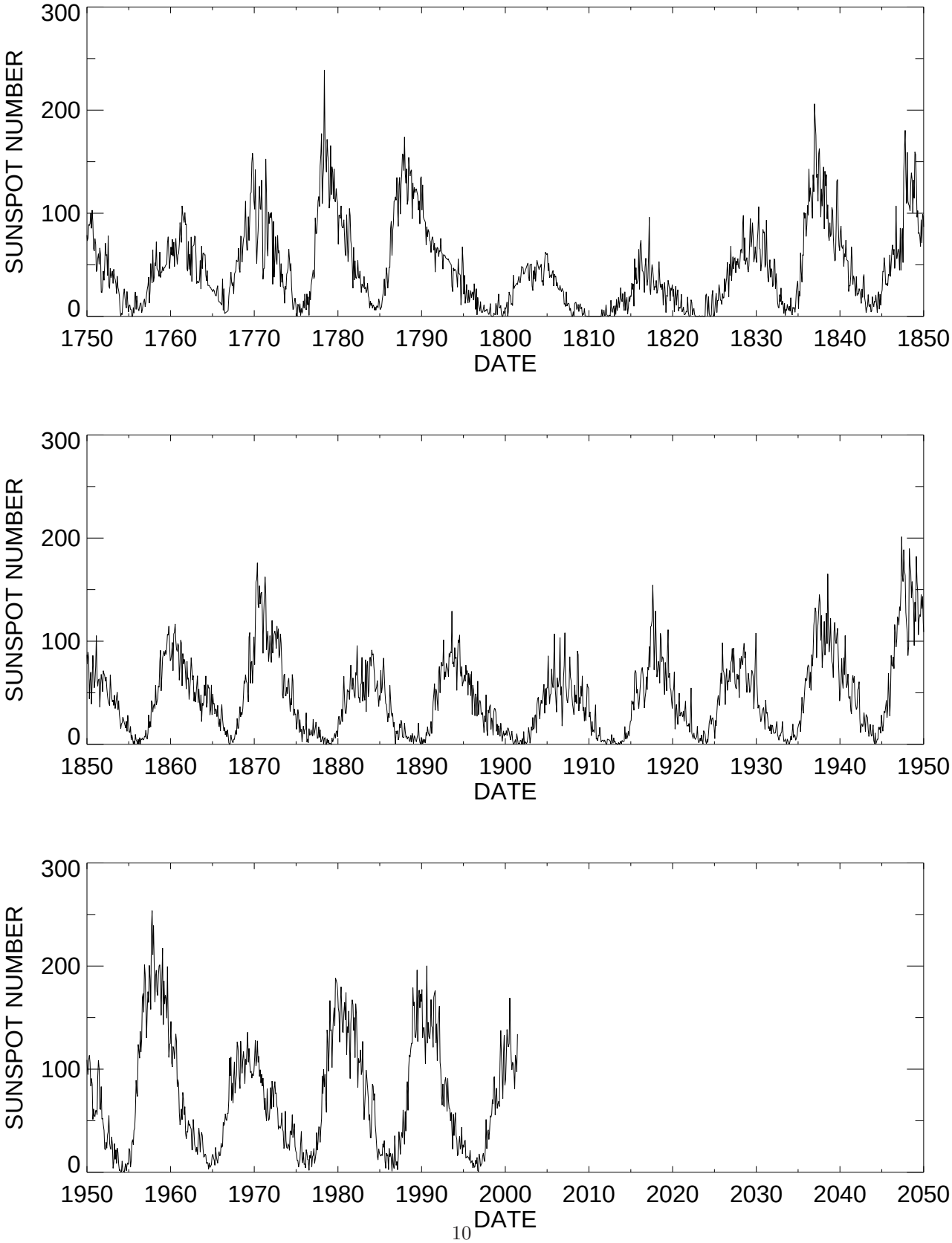
Example: For $n = 4$, we have $\omega_4 = e^{-2\pi i/4} = -i$, and the matrix is

$$\mathbf{F}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ 1 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}.$$

[]

To take a DFT, we could just set up the matrix and use Matlab matrix-vector multiplication. We would do n^2 multiplications.

Some very clever people (the idea has been rediscovered MANY times) figured out a faster way to do this. It is easiest to see for $n = 8$, so we start there. Then $z \equiv \omega_8 = e^{-2\pi i/8}$. Some facts:



NASA/MSFC/HATHAWAY ZURICH.PS 07/2001

Figure 1: Sunspot activity, 1750-2001. Data from http://science.nasa.gov/ssl/pad/solar/sunspots.htm

- $z^2 = \omega_4$.
- $z^8 = 1$.
- $z^4 = -1$.
- $z^5 = zz^4 = -z$

We have

$$\mathbf{F}_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & z & z^2 & z^3 & z^4 & z^5 & z^6 & z^7 \\ 1 & z^2 & z^4 & z^6 & 1 & z^2 & z^4 & z^6 \\ 1 & z^3 & z^6 & z^1 & z^4 & z^7 & z^2 & z^5 \\ 1 & z^4 & 1 & z^4 & 1 & z^4 & 1 & z^4 \\ 1 & z^5 & z^2 & z^7 & z^4 & z & z^6 & z^3 \\ 1 & z^6 & z^4 & z^2 & 1 & z^6 & z^4 & z^2 \\ 1 & z^7 & z^6 & z^5 & z^4 & z^3 & z^2 & z^1 \end{bmatrix}$$

We want to form $\mathbf{F}_8\mathbf{x}$, where $\mathbf{x}$ is a given vector. Note that we can reorder the columns of $\mathbf{F}_8$, as long as we also reorder the rows of $\mathbf{x}$, and still get the right answer. Let's put the odd (plum-colored) columns first, followed by the evens:

$$\mathbf{F}_8^{reordered} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & z^2 & z^4 & z^6 & z & z^3 & z^5 & z^7 \\ 1 & z^4 & 1 & z^4 & z^2 & z^6 & z^2 & z^6 \\ 1 & z^6 & z^4 & z^2 & z^3 & z & z^7 & z^5 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & z^2 & z^4 & z^6 & -z & -z^3 & -z^5 & -z^7 \\ 1 & z^4 & 1 & z^4 & -z^2 & -z^6 & -z^2 & -z^6 \\ 1 & z^6 & z^4 & z^2 & -z^3 & -z & -z^7 & -z^5 \end{bmatrix} = \begin{bmatrix} \mathbf{F}_4 & \mathbf{D}\mathbf{F}_4 \\ \mathbf{F}_4 & -\mathbf{D}\mathbf{F}_4 \end{bmatrix}$$

where

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & z & 0 & 0 \\ 0 & 0 & z^2 & 0 \\ 0 & 0 & 0 & z^3 \end{bmatrix}$$

This is pretty, but what good is it?

We now know that we can compute the DFT of $\mathbf{x}$ by forming

$$\begin{aligned} \mathbf{F}_8\mathbf{x} &= \mathbf{F}_8^{reordered}\mathbf{x}^{reordered} \\ &= \begin{bmatrix} \mathbf{F}_4 & \mathbf{D}\mathbf{F}_4 \\ \mathbf{F}_4 & -\mathbf{D}\mathbf{F}_4 \end{bmatrix} \begin{bmatrix} \mathbf{x}(1:2:8) \\ \mathbf{x}(2:2:8) \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{I} & \mathbf{D} \\ \mathbf{I} & -\mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{F}_4\mathbf{x}(1:2:8) \\ \mathbf{F}_4\mathbf{x}(2:2:8) \end{bmatrix}. \end{aligned}$$

Therefore, we can do the $n = 8$ DFT by computing two $n = 4$ DFTs and then scaling, at a cost of 2 times the $n = 4$ cost, plus 12 multiplications. (And if we are more careful, we can reduce the multiplications, too.)

Suppose we had an $n = 128$ DFT to compute. We could do this by two $n = 64$ DFTs plus scaling.

But each $n = 64$ DFT could be done by two $n = 32$ DFTs plus scaling ....

Thus we have a recursive algorithm for the DFT, at dramatically less cost. For example, for $n = 128$, the matrix-vector product DFT takes 214665 Matlab flops (floating point operations), while a fast algorithm based on this recursion (Matlab's `FFT`) takes 5053 flops. (See the table on p. 198.) The fast algorithm's work grows as $n \log_2 n$ rather than n^2.