Chapter 5: Matrix Computations

FuSen F. Lin

Department of Computer Science and Engineering
National Taiwan Ocean University

Scientific Computing, Fall 2011

- 5.0 Introduction
- 5.1 Setting Up Matrix Problems
- 5.2 Matrix Operations
- 5.3 Once Again, Setting Up Matrix Problems
- 5.4 Recursive Matrix Operations
- 5.5* Distributed Memory Matrix Multiplications

Introduction

- The next item on our agenda is the linear equation problem $Ax = b$. However, before we get into algorithmic details, it is important to study two calculations: matrix-vector multiplication and matrix-matrix Multiplication.
- We first pay attention to the act of setting up a matrix, particularly, when each matrix entry a_{ij} is an evaluation of a continuous function $f(x, y)$.
- Fast Fourier transform and the fast Strassen matrix multiply algorithm are presented as examples of recursion in matrix computations.

Setting Up Matrix Problems

- Before a matrix problem can be solved, it must be set up. In many applications, the amount of work associated with the set-up phase rivals the amount of work associated with the solution phase.
- Therefore, it is in our interest to acquire this activity and also occasionally to see how many of MATLAB's vector capabilities extend to the matrix level.

- If the entries in a matrix $A = (a_{ij})$ are specified by recipes, such as (the Hilbert matrix)

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

then a double-loop script can be used for its computation:

Hilbert matrix

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

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

Hilbert matrix

- Pre-allocation with **zeros(n,n)** reduces memory management overhead.
- If a matrix is symmetric, that is, $a_{ij} = a_{ji}$ for all i and j, then the (i, j) recipe need only be applied half the time.
- In MATLAB, there is a built-in function **A = hilb(n)** for Hilbert matrix that can be used in lieu of the preceding scripts.
- The setting-up of a matrix can often be made more efficient by exploiting relationships that exist between the entries.

- Consider the construction of the lower triangular matrix of binomial coefficients:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 1 & 3 & 3 & 1 & 0 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

- The binomial coefficient ”*m*-choose-*k*” is defined by

$$\binom{m}{k} = \begin{cases} \frac{m!}{k!(m-k)!}, & \text{if } 0 \le k \le m, \\ 0, & \text{otherwise.} \end{cases}$$

Matrix of Pascal Triangle

- Let the *ij* entry of the matrix we are setting up is defined by

$$p_{ij} = \binom{i-1}{j-1}.$$

- If we compute each entry using the factorial definition, then $O(n^3)$ flops are involved.
- Notice that P is lower triangular with ones on the diagonal and in the first column. An entry not in these locations is the sum of its "north" and "northwest" neighbors. That is,

$$p_{ij} = p_{i-1,j-1} + p_{i-1,j}.$$

- Therefore, we have the following set-up strategy:

Binomial Coefficient Matrix

```
P = zeros(n, n);
P(:,1) = ones(n,1);  % put the first column to be 1.
for i = 2:n,
   for j = 2:i,
      P(i,j) = P(i-1,j-1) +  P(i-1,j);
   end
end

% Also call it as Pascal triangle matrix.

% This script only involves $O(n^2)$ flops and is
% therefore an order of magnitude faster than the
% method that ignores the connections between the p_ij.
```

Vandermonde Matrices

- Many matrices are defined by a vector of parameters. Such as the Vandermonde matrices:

$$P = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & x_1^4 \\ 1 & x_2 & x_2^2 & x_2^3 & x_2^4 \\ 1 & x_3 & x_3^2 & x_3^3 & x_3^4 \\ 1 & x_4 & x_4^2 & x_4^3 & x_4^4 \\ 1 & x_5 & x_5^2 & x_5^3 & x_5^4 \end{bmatrix}$$

- The best set-up strategy is the column-oriented technique:

Vandermonde Matrix

```
n = length(x);
V(:,1) = ones(n,1);  % put the first column to be 1.
for j = 2:n,
    % Set up column j.
      V(:,j) = x .*  V(:,j-1);
end
```

Circulant Matrices (1)

- The circulant matrices are also of this genre (type). They also are defined by a vector of parameters, for example

$$C = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ a_4 & a_3 & a_2 & a_3 \\ a_3 & a_4 & a_1 & a_2 \\ a_2 & a_3 & a_4 & a_1 \end{bmatrix}$$

- Each row in a circulant is a shifted version of the row above it. Two kinds of set-up functions: The MATLAB *functions* are **circulant1** and **circulant2**.

Circulant Matrices (2)

- **circulant1** exploits the fact that

$$c_{ij} = a_{((n-i+j)_{\text{mod } n})+1}$$

  and is a scalar-level implementation.
- **circulant2** exploits the fact that $C(i,:)$ is a left shift of $C(i-1,:)$ and is a vector-level implementation.
- The script **ScircBench** compares t_1 (the time required by **circulant1**) with t_2 (the time required by **circulant2**).

- Circulant matrices are examples of Toeplitz matrices. Toeplitz matrices arise in many applications and are constant along their diagonals.
- For example,

$$T = \begin{bmatrix} c_1 & r_2 & r_3 & r_4 \\ c_2 & c_1 & r_2 & r_3 \\ c_3 & c_2 & c_1 & r_2 \\ c_4 & c_3 & c_2 & c_1 \end{bmatrix}$$

If **c** and **r** are n-vectors, then **T = toeplitz(c,r)** set up the matrix

$$t_{ij} = \begin{cases} c_{i-j+1}, & \text{if } i \geq j, \\ r_{j-i+1}, & \text{if } j > i. \end{cases}$$

Band Matrices

- Many important classes of matrices have lots of zeros. Such as the lower triangular matrices, upper triangular matrices, and tridiagonal matrices:

$$L = \begin{bmatrix} x & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 \\ x & x & x & x & 0 \\ x & x & x & x & x \end{bmatrix} \quad U = \begin{bmatrix} x & x & x & x & x \\ 0 & x & x & x & x \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & x \end{bmatrix}$$

$$T = \begin{bmatrix} x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 \\ 0 & x & x & x & 0 \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \end{bmatrix}$$

- The x-0 notation is a handy way to describe patterns of zeros and nonzeros in a matrix. Each "x" designates a nonzeros scalar.

Banded Matrices

- In general, a matrix $A = (a_{ij})$ has *lower bandwidth p* if $a_{ij} = 0$ whenever $i > j + p$. Thus, an upper triangular matrix has lower bandwidth 0 and a tridiagonal matrix has lower bandwidth 1.
- A matrix $A = (a_{ij})$ has *upper bandwidth q* if $a_{ij} = 0$ whenever $j > i + q$. Thus, a lower triangular matrix has upper bandwidth 0 and a tridiagonal matrix has upper bandwidth 1.
- For instance, here is a matrix with upper bandwidth 2 and lower bandwidth 3:

$$A = \begin{bmatrix} x & x & x & 0 & 0 & 0 & 0 \\ x & x & x & x & 0 & 0 & 0 \\ x & x & x & x & x & 0 & 0 \\ x & x & x & x & x & x & 0 \\ 0 & x & x & x & x & x & x \\ 0 & 0 & x & x & x & x & x \\ 0 & 0 & 0 & x & x & x & x \end{bmatrix}$$

Diagonal Matrices

- Diagonal matrices have upper and lower bandwidth 0 and can be established by using **diag** *function*.
- For instance, if **d** $= [10, 20, 30, 40]$ and **D = diag(d)**, then

$$D = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 20 & 0 & 0 \\ 0 & 0 & 30 & 0 \\ 0 & 0 & 0 & 40 \end{bmatrix}$$

- Two-argument calls to **diag** are also possible and can be used to create the other diagonals of a matrix.
- For instance, how to build the matrix: An entry a_{ij} is on the kth diagonal if $j - i = k$. That is, a matrix whose entries equal the diagonal values (called diagonal-value matrix). Such as

$$D = \begin{bmatrix} 0 & 1 & 2 & 3 \\ -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 \\ -3 & -2 & -1 & 0 \\ -4 & -3 & -2 & -1 \end{bmatrix}$$

Some Banded Matrices (2)

- If v is an m-vector, then **D = diag(v,k)** establishes an $(m+k)$-by-$(m+k)$ matrix that has a kth diagonal equal to v and is zero everywhere else. Thus

$$\mathbf{diag}([10,20,30],2) = \begin{bmatrix} 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 & 30 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- If A is a matrix, then **v = diag(A, k)** extracts the kth diagonal and assigns it (as a column vector) to **v**. (see and play **diagValue.m**)

Some Banded Matrices (3)

- The functions **tril** and **triu** can be used to punch out a banded portion of a given matrix.
- If **B = tril(A, k)**, then

$$b_{ij} = \begin{cases} a_{ij}, & \text{if } j \leq i + k, \\ 0, & \text{if } j > i + k. \end{cases}$$

- Analogously, if **B = triu(A, k)**, then

$$b_{ij} = \begin{cases} a_{ij}, & \text{if } j \geq i + k, \\ 0, & \text{if } j < i + k. \end{cases}$$

- For example, **A = tril(ones(6, 6), 1)**

Examples of banded Matrices

- The following commands are equivalent:
- **T = - triu(tril(ones(6, 6), 1), -1) + 3*eye(6,6)**
- **T = - diag(ones(5, 1), -1) + diag(2*ones(6, 1), 0) - diag(ones(5, 1), 1)**
- **T = toeplitz([2;-1; zeros(4, 1)], [2; -1; zeros(4,1)])**

Block Matrices (1)

- MATLAB supports the synthesis of matrices at scalar level or matrix level. That is, the notation

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

  means that A is a 2-by-3 matrix with entries a_{ij}, which can be scalars or matrices.

- Suppose A_{11}, A_{12}, A_{13}, A_{21}, A_{22}, and A_{23} have the following shapes:

$$A_{11} = \begin{bmatrix} u & u & u \\ u & u & u \\ u & u & u \end{bmatrix} \quad A_{12} = \begin{bmatrix} v \\ v \\ v \end{bmatrix} \quad A_{13} = \begin{bmatrix} w & w \\ w & w \\ w & w \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} x & x & x \\ x & x & x \end{bmatrix} \quad A_{22} = \begin{bmatrix} y \\ y \end{bmatrix} \quad A_{23} = \begin{bmatrix} z & z \\ z & z \end{bmatrix}$$

Block Matrices (2)

- We then define a 2-by-3 block matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

by

$$A = \left[\begin{array}{ccc|c|cc} u & u & u & v & w & w \\ u & u & u & v & w & w \\ u & u & u & v & w & w \\ \hline x & x & x & y & z & z \\ x & x & x & y & z & z \end{array}\right]$$

- The lines delineate the block entries. Of course, A is also a 5-by-6 scalar matrix.
- Block matrix manipulations are very important and can be effectively carried out in MATLAB (see **blockTest**).

Block Matrices (3)

- The block rows of a matrix are separated by semicolons, and it is important to make sure that the dimensions are consistent. The final result must be rectangular at the scalar level.
- The extraction of blocks requires the colon notation. The assignment **C = A(2:4, 5:6)** is equivalent to any of the following:

$$
\begin{aligned}
C &= [A(2:4,5)\ A(2:4,6)] \\
C &= [A(2,5:6), A(3,5:6), A(4,5:6)] \\
C &= [A(2,5)\ A(2,6); A(3,5)\ A(3,6); A(4,5)\ A(4,6)]
\end{aligned}
$$

Block Matrices (4)

- A block matrix can be conveniently represented as a cell array with matrix entries. The function **MakeBlock** does this when the underlying matrix can be expressed as a square block matrix with square blocks.
- As the examples, see **MakeBlockTest**.

Matrix-Vector Multiplication (1)

- Once a matrix is set up, it can participate in matrix-vector and matrix-matrix products. Although these operations are MATLAB one-liners, it is instructive to examine the different ways that they can be implemented.
- Suppose $A \in \mathbb{R}^{m \times n}$, and we wish to compute the matrix-vector product $y = Ax$, where $x \in \mathbb{R}^n$.
- The usual way this computation proceeds is to compute the dot products

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

one at a time for $i = 1 : m$. This leads to the following algorithm:

Matrix-Vector Multiplication: Dot Products

```
% Algorithm of dot product for y = Ax

[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
```

Matrix-Vector Multiplication (2)

- The one-line assignment $y = Ax$ is equivalent and requires $2mn$ flops.
- Instructively, we reconsider the preceding double loop and recognize that the j-loop oversees an inner product of the ith row of A and the x vector. We therefore have the *function* **MatVecRo**.
- This procedure is *row oriented* because A is accessed by row.

Matrix-Vector Multiplication: Row-Oriented

```
function y = MatVecRo(A, x)

 % y = MatVecRo(A,x)
 % Computes the matrix-vector product y = A*x
 % (via saxpys) where A is an m-by-n matrix and x is
 % a column-vector.

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

Matrix-Vector Multiplication (3)

- If A is accessed by column, then we have the*column-oriented* procedure, the MATLAB *function* **MatVecCo**.
- For example, we start with a 3-by-2 observation: $y = Ax =$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 7 + 2 \cdot 8 \\ 3 \cdot 7 + 4 \cdot 8 \\ 5 \cdot 7 + 6 \cdot 8 \end{bmatrix} = 7 \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} + 8 \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} = \begin{bmatrix} 23 \\ 53 \\ 83 \end{bmatrix}$$

- In other words, y is a linear combination of A's columns with the x_j being the coefficients.
- In terms of program transformation, this function **MatVecCo** is just **MatVecRo** with the i and j loops swapped.

Matrix-Vector Multiplication: Column-Oriented

```
function y = MatVecCo(A,x)

% y = MatVecCo(A,x)
% This computes the matrix-vector product y = A*x
% (via saxpys) where A is an m-by-n matrix and x is
% a columnn-vector.

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

Matrix-Vector Multiplication (4)

- The **saxpy** operation is the form

$$\text{vector} \longleftarrow \text{scalar} \cdot \text{vector} + \text{vector}.$$

- Along with the dot product, it is a key player in matrix computations. Here is an expanded view of the saxpy operation in **MatVecco**:

$$\begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ \vdots \\ y(m) \end{bmatrix} = \begin{bmatrix} A(1,j) \\ A(2,j) \\ \vdots \\ \vdots \\ A(m,j) \end{bmatrix} x(j) + \begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ \vdots \\ y(m) \end{bmatrix}$$

- This procedure is *row oriented* because A is accessed by row.

Structure-Exploiting Upper Triangular (1)

- In many matrix computations, the matrix are structured with lots of zeros (such as lower/upper triangular matrices or banded matrices). In such a context it may be possible to streamline the computations.
- We first examine the matrix-vector product problem $y = Ax$ where $A \in \mathbb{R}^{n\times n}$ is upper triangular. By looking at **MatVecRo**, the inner products include many zeros.
- We therefore can reduce the computations so that they only include the nonzero portion of the row. The command in **MatVecRo** should be modified by

$$\text{A(i,:)*x} \quad \longrightarrow \quad \text{A(i,i:n)*x(i:n)}$$

Structure-Exploiting Upper Triangular (2)

- The assignment to $y(i)$ requires $2i$ flops, so overall

$$\sum_{i=1}^{n}(2i) = 2(1+2+\cdots+n) = n(n+1)$$

flops required.

- Ignoring the $O(n)$ term, we merely state that the algorithm requires n^2 flops, and that our streamlining halved the number of floating point operations.

Structure-Exploiting Upper Triangular (3)

- The function **MatVecRo** can also be abbreviated. Note that $A(:,j)$ is zero in components $j+1$ through n, and so the "essential" saxpy to perform in the jth step is

$$\begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ \vdots \\ y(j) \end{bmatrix} = \begin{bmatrix} A(1,j) \\ A(2,j) \\ \vdots \\ \vdots \\ A(j,j) \end{bmatrix} x(j) + \begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ \vdots \\ y(j) \end{bmatrix}.$$

- The rendering of the key command is

$$y(1:j) = A(1:j,j) * x(j) + y(1:j);$$

Again, the number of required flops is halved.

Matrix-Matrix Multiplication (1)

- If $A \in \mathbb{R}^{m\times p}$ $B \in \mathbb{R}^{p\times n}$, then the product $C = AB$ is defined by
$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$
for all i and j that satisfy $1 \leq i \leq m$ and $1 \leq j \leq n$.
- In other words, each entry in C is the inner product of a row in A and a column in B. Thus, the fragment (see next page) computes the product AB and assigns the result to C.
- MATLAB supports matrix-matrix multiplication, and so this can be implemented with one-liner
$$C = A * B.$$

Matrix-matrix Multiplication: Scalar product

```
C = zeros(m,n);
for j = 1:n,
   for i = 1:m,
      for k=1:p,
         C(i,j) = C(i,j) + A(i,k)*B(k,j);
      end
   end
end
```

Matrix-Matrix Multiplication (2)

- There are a number of different ways to look at matrix multiplication, and we shall present four distinct versions.
- First, dot-product version: the innermost loop in the preceding script oversees the dot product between row i of A and column j of B. See **MatMatDot**.
- Second, saxpy version: the jth column of C is equal to A times the j column of B. See **MatMatSax**.
- Third, matrix-vector product version: By replacing the inner loop in saxpy operation with a single matrix-vector product. See **MatMatVec**.
- Fourth, outer product version: the product $A * B$ is the sum of p outer products which are the columns of A multiply the rows of B. See **MatMatOuter**.

Outer Product Version of Matrix Multiplication (1)

- The outer product between a column m-vector u and a row n-vector v is given by

$$uv^T = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_m \end{bmatrix} [v_1, v_2, \cdots, v_n] = \begin{bmatrix} u_1v_1 & u_1v_2 & \cdots & u_1v_n \\ u_2v_1 & u_2v_2 & \cdots & u_2v_n \\ \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ u_mv_1 & u_mv_2 & \cdots & u_mv_n \end{bmatrix}$$

- This is just the ordinary matrix multiplication of an m-by-1 matrix and a 1-by-n matrix (producing an m-by-n matrix).
- For instance,

$$\begin{bmatrix} 10 \\ 15 \\ 20 \end{bmatrix}_{3\times 1} [1, 2, 3, 4]_{1\times 4} = \begin{bmatrix} 10 & 20 & 30 & 40 \\ 15 & 30 & 45 & 60 \\ 20 & 40 & 60 & 80 \end{bmatrix}_{3\times 4}$$

- Therefore, the outer-product version of matrix multiplication $C = A * B$ is given by the sum of p outer products:

$$C = AB = [A(:,1)|A(:,2)|\cdots|A(:,p)] \begin{bmatrix} B(1,:) \\ B(2,:) \\ \vdots \\ \vdots \\ B(p,:) \end{bmatrix} = \sum_{k=1}^{p} A(:,k)B(k,:),$$

  which are the columns of A multiply the rows of B.
- The Script File **MatBench** benchmarks the four versions of matrix-multiply *functions* along with the (default) direct, one-liner $C = A * B$.

Sparse Matrices (1)

- For many matrices that arise in practice, the ratio

$$\frac{\text{Number of Nonzero Entries}}{\text{Number of zero Entries}}$$

  is very small. Matrices with this property are said to be **sparse**.
- An important class of sparse matrices are band matrices, such as the block tridiagonal matrix shown in Figure 5.1 (page 183).
- If A is sparse then
  1. it can be represented with reduced storage and
  2. matrix-vector products that involve A can be carried out with reduced number of flops.

Sparse Matrices (2)

- For example, If A is an n-by$n$ tridiagonal matrix then it can be represented with with three n-vectors and when it multiplies a vector only $5n$ flops are involved. However, this would not be the case if A is represented as a full matrix.
- Thus,

  A = diag(2*ones(n,1)) - diag(ones(n-1,1),-1) - diag(ones(n-1,1),1);

  y = A*rand(n,1);

  involves $O(n^2)$ storage and $O(n^2)$ flops.
- The *sparse* function addresses these issues in MATLAB. If A is a matrix then **S_A = sparse(A)** produces a sparse array representation of A.
- The sparse array **S_A** can be engaged in the same matrix operations as A and MATLAB will exploit the underlying sparse structure whenever possible.

Sparse Matrices (3)

- Consider the script

```
A = diag(2*ones(n,1)) - diag(ones(n-1,1),-1) - diag(ones(n-1, 1),1);
S_A = sparse(A);
y = A*rand(n,1);
```

- The representation **S_A** involves $O(n)$ storage and the product $O(n)$ flops.
- The script **ShowSparse** looks at the flop efficiency in more detail and produces the plot shown in Figure 5.2.
- There are more sophisticated ways to use **sparse** which the interested reader can be pursue via **help**.

Norms of Vectors (1)

- Norms are a vehicle for measuring distance in a vector space. A norm is just a generalization of absolute value.
- For a vector $x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$, the 1, 2, $p(> 1)$, and infinity norms are defined as

$$\begin{aligned}
||x||_1 &= |x_1| + |x_2| + \cdots + |x_n| = \sum_{i=1}^{n} |x_i| \\
||x||_2 &= \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \\
||x||_p &= (x_1^p + x_2^p + \cdots + x_n^p)^{1/p} = \left[\sum_{i=1}^{n} x_i^p \right]^{1/p} \\
||x||_\infty &= \max_{1 \le i \le n} |x_i| = \max\{|x_1|, \ldots, |x_n|\}
\end{aligned}$$

Norms of Vectors (2)

- Whenever we think about vectors of errors in an order-of-magnitude sense, then the choice of norm is generally not important.
- It can be shown that

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

  Thus, the 1-norm cannot be particularly small without the others following suit.

Norms of Vectors (3)

- In MATLAB, if **x** is a vector, **norm(x,1)**, **norm(x,2)**, and **norm(x,inf)** can be used to ascertain these quantities. A single-argument call to **norm** returns the 2-norm (i.e., **norm(x)**).
- A script **AveNorms** tabulates the ratios $||x||_1/||x||_\infty$ and $||x||_2/||x||_\infty$ for large collections of random *n*-vectors.

Norms of Matrices (1)

- The matrix norms we will consider have the norms

$$||A||_\infty = \max_{||\mathbf{x}||_\infty = 1} ||A\mathbf{x}||_\infty \quad \text{and} \quad ||A||_2 = \max_{||\mathbf{x}||_2 = 1} ||A\mathbf{x}||_2$$

- If $A = (a_{ij})$ is an $m \times n$ matrix, then

$$||A||_\infty = \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{ij}|, \quad ||A||_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}|, \quad ||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}.$$

- For example, find $||A||_\infty$ and $||A||_1$ for the matrix

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 0 & 3 & -1 \\ 5 & -1 & 1 \end{bmatrix}$$

Norms of Matrices (2)

- In MATLAB if A is a matrix then **norm(A,1)**, **norm(A,2)**, **norm(A,inf)**, and **norm(A,'fro')** can be used to compute these values.
- As a simple illustration of how matrix norms can be used to quantify error at matrix level, we prove a result about the roundoff errors that arise when an m-by-n matrix is stored.
- **Theorem 5:** If $\hat{A}$ is the stored version of $A \in \mathbb{R}^{m\times n}$, then $\hat{A}A + E$ where $E \in \mathbb{R}^{m\times n}$ and

$$||E||_1 \leq \text{eps} \cdot ||A||_1.$$

Norms of Matrices (3)

- PROOF: From Theorem 1, if $\hat{A} = (\hat{a}_{ij})$, then

$$\hat{a}_{ij} = fl(a_{ij}) = a_{ij}(1 + \epsilon_{ij}),$$

where $|\epsilon_{ij}| \leq \text{eps}$. Thus,

$$\begin{aligned} ||E||_1 &= ||\hat{A} - A||_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |\hat{a}_{ij} - a_{ij}| \\ &\leq \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}\epsilon_{ij}| \leq \text{eps} \cdot \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}| = \text{eps} \cdot ||A||_1. \end{aligned}$$

- This theorem says that the errors of order $\text{eps} \cdot ||A||_1$ arise when a real matrix A is stored in floating point. There is nothing to do with what kind of norms is chosen.
- when the effect of roundoff error is the issue, we will be content with order-of-magnitude approximation. For example, it can be shown that if A and B are matrices of floating point numbers, then

$$||fl(AB) - AB|| \approx \text{eps} \cdot ||A||||B||.$$

By $fl(AB)$ we mean the computed floating point product of A and B. See **ProdBound**.

Evaluation of a Function $f(x, y)$ with Vectors x and y

- On numerous occasions we have been required to evaluate a continuous function $f(x)$ on a vector of values.
- The analog of this in two dimensions is the evaluation of a function $f(x, y)$ on a pair of vectors x and y.
- Suppose that $f(x, y) = \exp^{-(x^2+3y^2)}$ and that we want to set up an n-by-n matrix F with the property that

$$f_{ij} = e^{-(x_i^2+3y_j^2)}$$

where $x_i = (i-1)/(n-1)$ and $y_j = (j-1)/(n-1)$. We can proceed at the scalar, vector, or matrix level.

Two-Dimt'l Tables of Function Values (1)

```
% the scalar level
v = linspace(0, 1, n);
F = zeros(n, n);
for i = 1:n,
   for j = 1:n,
      F(i,j) = exp(-v(i)^2 - 3*v(j)^2);
   end
end

% the vector level (set F up by column)
v = linspace(0, 1, n);
F = zeros(n, n);
for j = 1:n,
   F(:,j) = exp(-v.^2 - 3*v(j)^2);
end
```

Two-Dimt'l Tables of Function Values (2)

```
 % evaluate 'exp' on the matrix of arguments:
 v = linspace(0, 1, n);
 A = zeros(n, n);
 for i = 1:n,
    for j = 1:n,
       F(i,j) = exp(-(v(i)^2 + 3*v(j)^2));
    end
 end
 F = exp(A);

function F = SampleF(x, y)
% x is a column n-vector, y is a column m-vector and
% F is an m-by-n matrix with F(i,j)=exp(-x(i)^2-3y(i)^2)
 n = length(x);  m = length(y);
 A = -((3*y.^2)*ones(1,n) + ones(m,1)*(x.^2)');
 F = exp(A);
```

Evaluation of a Function $f(x, y)$ with Matrix Arguments

- Many of MATLAB's built-in functions, like *exp*, accept matrix arguments. The Assignment $F = exp(A)$ sets F to be a matrix that is the same as size as A with $f_{ij} = e^{a_{ij}}$ for all i and j.
- In general, the most efficient approach depends on the structure of the matrix arguments, the nature of the underlying function $f(x, y)$, and what is already available through M-files.
- In order to increase the efficiency of computations, it is best to be consistent with MATLAB's vectorizing philosophy (processing with vector or matrix level) when designing functions or programs.

Contour Plots

- If $f(x, y)$ is a function of two real variables, then a curve in the xy-plane of the form $f(x, y) = c$ is a contour.
- The function **contour** can be used to display such curves. See **ShowContour**.

Approximating Double Integrals (1)

- Let us consider the problem of approximating the double integral

$$I = \int_a^b \int_c^d f(x, y) dxdy$$

using a quadrature rule of the form

$$\int_a^b g(x)dx \approx (b - a) \sum_{i=1}^{N_x} \omega_i g(x_i) \equiv Q_x$$

in the x-direction and a quadrature rule of the form

$$\int_c^d g(y)dy \approx (d - c) \sum_{j=1}^{N_y} \mu_j g(y_j) \equiv Q_y$$

in the y-direction.

Approximating Double Integrals (2)

- Doing this, we obtain

$$
\begin{aligned}
I &= \int_a^b \left(\int_c^d f(x,y)dy \right) dx \approx (b-a) \sum_{i=1}^{N_x} \omega_i \left(\int_c^d f(x_i,y)dy \right) \\
&= (b-a) \sum_{i=1}^{N_x} \omega_i \left((d-c) \sum_{j=1}^{N_y} \mu_j f(x_i,y_j) \right) \\
&= (b-a)(d-c) \sum_{i=1}^{N_x} \omega_i \left(\sum_{j=1}^{N_y} \mu_j f(x_i,y_j) \right) \equiv Q
\end{aligned}
$$

Approximating Double Integrals (3)

- Observe that the quantity in parentheses is the ith component of the vector F_μ, where

$$F = \begin{bmatrix} f(x_1, y_1) & \cdots & f(x_1, y_{N_y}) \\ \vdots & \ddots & \vdots \\ f(x_{N_x}, y_1) & \cdots & f(x_{N_x}, y_{N_y}) \end{bmatrix}$$

and

$$\mu = \begin{bmatrix} \mu_1 \\ \vdots \\ \mu_{N_y} \end{bmatrix}$$

- It follows that

$$Q = (b-a)(d-c)\omega^T(F\mu), \quad \text{where} \quad \omega = \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_{N_x} \end{bmatrix}$$

- See **Show2Dquad** and **CompQNC2D**.

- Some of the most interesting algorithmic developments in matrix computations are recursive. Two examples are given in this section.
- The first is the fast Fourier transform, a super-quick way of computing a special, very important matrix-vector product.
- The second is a recursive matrix multiplication algorithm that involves markedly fewer flops than the conventional algorithm.

The Fast Fourier Transform (1)

- The discrete Fourier transform (DFT) matrix is a complex Vandermonde matrix. Complex numbers have the form $a + i \cdot b$, where $i = \sqrt{-1}$. If we define

$$\omega_4 = \exp(-2\pi i/4) = \cos(2\pi/4) - i \cdot \sin(2\pi/4) = -i$$

then the 4-by-4 DFT is given by

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ 1 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{bmatrix}$$

The Fast Fourier Transform (2)

- The parameter ω_4 is a fourth root of unity, meanimg that $\omega_4^4 = 1$. It follows that

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$$

- MATLAB supports complex matrix manipulation. The command
  $i = \sqrt{-1}$;
  F = [1 1 1 1 ; 1 $-i$ -1 i ; 1 -1 1 -1 ; 1 i -1 $-i$]
  assign the 4-by-4 DFT to F.

The Fast Fourier Transform (3)

- For general n, the DFT matrix is defined in terms of

$$\omega_n = \exp(-2\pi i/n) = \cos(2\pi/n) - i \cdot \sin(2\pi/n)$$

In particular, the n-by-n DFT matrix is defined by

$$F_n = (f_{pq}), \qquad f_{pq} = \omega_n^{(p-1)(q-1)}.$$

- Setting up the DFT matrix gives us an opportunity to sample MATLAB's complex arithmetic capabilities:

Setting the DFT matrix (1)

```
 F = ones(n, n);
 F(:, 2) = exp(-2*pi*sqrt(-1)/n).^(0:n-1);
 for k = 3:n,
    F(:, k) = F(:, 2) .* F(:, k-1);
 end

% Using the functions 'real' and 'imag' to extract
% the real and imaginary parts of a matrix.

 y = F*x;

 % is equivalent  to

 FR = real(F);  FI = imag(F);
 xR = real(x);  xI = imag(x);
 y = (FR*xR - FI*xI) + sqrt(-1)*(FR*xI + FI*xR);
```

Setting the DFT matrix (2)

```
% It is possible to compute 'y = F_n*x' without
% explicitly forming the DFT matrix 'F_n'.

n = length(x);
y = x(1)*ones(n,1);
for k = 2:n,
   y = y + exp(-2*pi*sqrt(-1)*(k-1)*(0:n-1)')*x(k);
end
```

The Fast Fourier Transform (4)

- The update carries out the saxpy computation

$$y = \begin{bmatrix} 1 \\ \omega_n^{k-1} \\ \omega_n^{2(k-1)} \\ \vdots \\ \omega_n^{(n-1)(k-1)} \end{bmatrix} x_k$$

- Notice that since $\omega_n^n = 1$, all power of ω_n are in the set $\{1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1}\}$. In particular, $\omega_n^m = \omega_n^{m \mod n}$.
- Thus, if

$$v = \exp(-2 * \mathrm{pi} * \mathrm{sqrt}(-1)/n) * (0 : n-1)'$$

$$z = \mathrm{rem}((k-1) * (0 : n-1)', n) + 1;$$

then $v(z)$ equals the kth column of F_n and we obtain the function **DFT**, which is an $O(n^2)$ algorithm.

The Fast Fourier Transform (5)

- An $O(n \log_2 n)$ implementation, called the algorithm of fast Fourier transform, by exploiting the structure of F_n with $n = 2^k$ (an integer power of 2). Consider the case $n = 8$, ...

Strassen Multiplication (1)

- The idea of Strassen algorithm for matrix multiplication is based on the 'block' matrix multiplication.
- Ordinarily, 2-by-2 matrix multiplication requires 8 multiplications and 4 additions:

$$
\begin{aligned}
\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} &= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \\
&= \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}
\end{aligned}
$$

Strassen Multiplication (2)

- In the Strassen multiplication scheme, the computations are rearranged so that they involve 7 multiplications and 18 additions:

$$
\begin{aligned}
P_1 &= (A_{11}+A_{22})(B_{11}+B_{22}) & P_2 &= (A_{21}+A_{22})B_{11} \\
P_3 &= A_{11}(B_{12}-B_{22}) & P_4 &= A_{22}(B_{21}-B_{11}) \\
P_5 &= (A_{11}+A_{12})B_{22} & P_6 &= (A_{21}-A_{11})(B_{11}+B_{12}) \\
P_7 &= (A_{12}-A_{22})(B_{21}+B_{22}) & & \\
C_{11} &= P_1+P_4-P_5+P_7 & C_{12} &= P_3+P_5 \\
C_{21} &= P_2+P_4 & C_{22} &= P_1+P_3-P_2+P_6
\end{aligned}
$$

- It is easy to verify that these recipes correctly define the product AB.

Strassen Multiplication (3)

- The Strassen specification holds when A_{ij} and B_{ij} are square matrices themselves. In this case, it amounts to a special method for computing 2-by-2 matrix products.
- The 7 multiplications are now m-by-m ($m = n/2$) matrix multiplication and require $2(7m^3)$ flops. The 18 additions are matrix additions and they involve $18m^2$ flops.
- Thus, for this block size the Strassen multiplications requires

$$2(7m^3) + 18m^2 = \frac{7}{8}(2n^3) + \frac{9}{2}n^2$$

flops while the corresponding figure for the conventional algorithm is given by $2n^3 - n^2$. We see that for large enough n, the Strassen approach involves less arithmetic (for $n > 22$).
- The idea can obviously be implemented recursively. See **Strass**.