Matrix Operations

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2012 ; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS 667-101 : Spring 2012 . All rights reserved.

Definitions

In the next few pages we review some definitions from matrix theory. This roughly corresponds to the material covered in the textbook from page 724 (Chapter 28, Matrix Operations) through page 734.

A matrix is a rectangular array of numbers. For example, A is a 3×4 matrix, 3 indicating the number of rows and 4 the number of columns.

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \end{pmatrix}$$

In A, a_{ij} represents the entry in row i and column j. The transpose of A denoted by A^T is A with rows becoming columns and columns becoming rows.

$$A^T = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 9 \\ 1 & 8 & 27 \end{pmatrix}$$

A (column) vector x is an $n \times 1$ matrix. A row vector x^T is an $1 \times n$ matrix. A unit vector e_i is one whose i-th row is 1 and all other entries 0. A zero matrix is a matrix with all elements zeros. A square matrix is an $n \times n$ matrix.

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, x^T = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}.$$

Special cases of matrices are (a) a diagonal (only a_{ii} elements are non-zero), (b) the identity matrix whose i-th column is e_i, $i = 1, \ldots, n$, (c) a tridiagonal where $a_{ij} = 0$ for $\mid i - j \mid > 1$, (d) an upper triangular ($a_{ij} = 0, i > j$ and a lower triangular ($a_{ij} = 0, i < j$, (e) a permutation matrix whose entries are zero except for a single one in every row and column.

A matrix A is symmetric if and only if $A = A^T$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Addition For two matrices A and B of dimension $n \times m$, matrix C of dimension also $n \times m$ is called the sum of A and B and we write $C = A + B$ if and only if $c_{ij} = a_{ij} + b_{ij}$.

A, B and C must have the same dimensions. If A, B, C are $n \times m$ matrices, then addition requires $O(nm)$ time.

Subtraction is defined similarly and requires as much time as well.

Multiplication For two matrices A, B, matrix $C = A \times B$ is defined as long as the column size of A equals the row size of B, i.e. if A is an $n \times m$ and B an $m \times p$ matrix, then C is an $n \times p$ matrix such that $c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$.

Matrix multiplication thus requires mnp multiplications and $np(m-1)$ additions.

The inverse of a $n \times n$ matrix A is a matrix A^{-1} such that $AA^{-1} = A^{-1}A = I$. If a matrix has an inverse it is called nonsingular or invertible.

n vectors are linearly independent if $\sum_i c_i x_i = 0$ implies $c_i = 0$ for all i.

Otherwise the n vectors are called dependent.

For example, the n vectors of an identity matrix are linear independent.

The column rank of and $n \times m$ matrix A is the size of the largest set of linearly independent columns. A row rank can similarly be defined.

The row rank of A equals its column rank and we simply call it the rank of A, i.e. $rank(A) \leq \min(m, n)$. A square matrix has full rank if $rank(A) = m = n$. A matrix A has full column rank if $rank(A) = m$.

A null vector x for a matrix A is a non-zero vector x such that $Ax = 0$.

Theorem. A square matrix has full rank iff (if and only if) it is nonsingular (i.e. the inverse of A exists).

Theorem. A square matrix is singular iff it has a null vector.

Theorem. A matrix A has full column rank if it does not have a null vector. it is nonsingular.

The determinant of A, denoted by $det(A)$ is a scalar (number).

(c) Copyright A. Gerbessiotis. All rights reserved.

Introduction

Dynamic programming, like the divide-and-conquer method, solves problems by dividing a problem into subproblems, solving the subproblems, and combining the solutions to subproblems. In d-and-c subproblems are usually independent of each other. DP however is used in cases where subproblems are not independent. A DP algorithm solves subproblems once, saves the solutions in a table and avoids recomputing the answer for a problem that has already been solved before. The development of a DP algorithm is broken into a sequence o steps.

(1) Characterize the structure of an optimal solution.

(2) Recursively define the value of an optimal solution.

(3) Compute the value of an optimal solution in a bottom-up fashion.

(4) Construct an optimal solution from computed information.

We are going to use dp techniques in solving the following problem.
Problem. Generalized matrix multiplication of q matrices.
Input. A matrix product $M = M_1 \times M_2 \times \ldots \times M_q$, where M_i is an $r_i \times r_{i+1}$ matrix.
Output. Compute M using the fewest number of operations.

Note that matrix multiplication is an associative operation i.e. $A \times (B \times C) = (A \times B) \times C$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Method. Dynamic Programming allows for the solution of this problem in $O(q^3)$ time.

Description. Let M_{ij} denote the result of the consecutive product chain $M_i \times \ldots \times M_j$ and let $c[i,j]$ be the cost of OPTIMALLY computing such a product. Let $k[i,j]$ be such that the optimal solution of M_{ij} requires the computation of $M_{ik} \times M_{k+1j}$, where $k = k[i,j]$. Therefore in an optimal computation of M_{ij}, M_{ik} and M_{k+1j} are first computed and then M_{ij} as the product of the two partial results.

Then we obtain the following dynamic programming formula.

$$c[i,i] = 0 \text{ for all } 1 \leq i \leq q.$$

$$c[i,i+1] = r_i r_{i+1} r_{i+2} \quad \text{and} \quad k[i,i+1] = i, \text{ for all } 1 \leq i < q.$$

$$c[i,i+s] = \min_{i \leq k < i+s} \{c[i,k] + c[k+1,i+s] + r_i r_{k+1} r_{i+s+1}\} \text{ for all } 1 \leq i \leq q-s \text{ and } 1 < s < q \text{ with}$$

$$k[i,i+s] = \text{value of k that obtains the above minimum}$$

(c) Copyright A. Gerbessiotis. All rights reserved.

An example

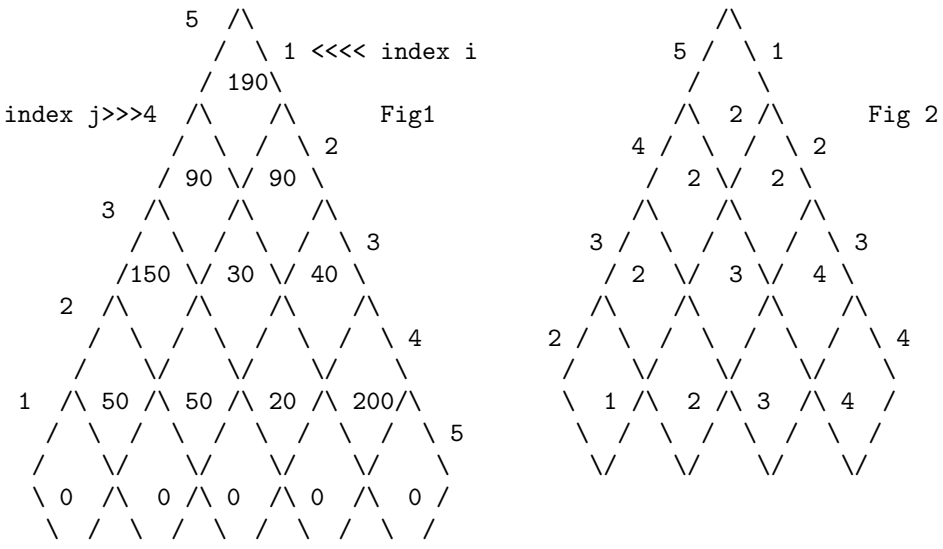


Fig1

Fig 2

1. STANDARD VIEW

2. ALTERNATIVE VIEW

i \ j=i+s	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200 <-------------c[i,i+1]
5					0 <-------------c[i,i]

(c) Copyright A. Gerbessiotis. All rights reserved.

An example (continued)

With reference to the left figure, $c[1,1] = c[2,2] = c[3,3] = c[4,4] = c[5,5] = 0$, and thus the first row of zeroes is filled. In order to fill the second line from the bottom $c[1,2] = 10 \times 5 \times 1 = 50$, and similarly $c[2,3] = 50$, $c[3,4] = 20$, and $c[4,5] = 200$. At the same time we record i in Figure 2 (in the corresponding box).

In order to fill the third line from the bottom we note that $c[1,3] = \min\{c[1,1]+c[2,3]+10\ 5\ 10, c[1,2]+c[3,3]+10\ 1\ 10\} = \min\{0+50+500, 50+0+100\} = 150$ and record $k = 2$ in Fig 2. Similarly $c[2,4] = \min\{c[2,2]+c[3,4]+5\ 1\ 2, c[2,3]+c[4,4]+5\ 10\ 2\} = \min\{0+20+10, 50+100\} = 30$ and record $k = 3$. $c[3,5] = \min\{c[3,3]+c[4,5]+1\ 10\ 10, c[3,4]+c[5,5]+1\ 2\ 10\} = \min\{0+200+100, 20+0+20\} = 40$ and record $k = 4$.

Then we fill the fourth row from the bottom. $c[1,4] = \min\{c[1,1]+c[2,4]+10\ 5\ 2, c[1,2]+c[3,4]+10\ 1\ 2, C[1,3]+c[4,4]+10\ 10\ 10\} = \min\{0+30+100, 50+20+20, 150+0+1000\} = 90$ and record $k = 2$ in Fig 2. $c[2,5] = \min\{c[2,2]+c[3,5]+5\ 1\ 10, c[2,3]+c[4,5]+5\ 10\ 10, C[2,4]+c[5,5]+5\ 2\ 10\} = \min\{0+40+50, 50+200+500, 30+0+100\} = 90$ and record $k = 2$ in Fig 2.

Finally, $c[1,5] = \min\{c[1,1]+c[2,5]+10\ 5\ 10, c[1,2]+c[3,5]+10\ 1\ 10, C[1,3]+c[4,5]+10\ 10\ 10, c[1,4]+c[5,5]+10\ 2\ 10\} = \min\{0+90+500, 50+40+100, 150+200+1000, 90+0+200\} = 190$ and record $k = 2$ in Fig 2.

Summarizing, in order to compute $M_1 \ldots M_5$ we need to perform 190 operations. We first compute $M_1 \times M_2$ and then $M_3 \times \times M_4 \times M_5$. The first product is computed directly in 50 multiplications by reading entry $c[1,2]$ from Figure 1. For M_{35} we obtain that we need to perform 40 operations since $c[3,5] = 40$ in Figure 1, and the way is by finding $M_3 \times M_4$ first and then multiplying the result with M_5 as derived from the $[3,5]$ entry of Figure 2. The first product $M_3 \times M_4$ requires 20 operations, and the second the remaining twenty multiplications. Therefore $M_{15} = (M_1 \times M_2) \times ((M_3 \times M_4) \times M_5)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Let us consider again the case of matrix multiplication for two $n \times n$ square matrices.
Consider the vector product of two vectors $a = (a_1, \ldots, a_n)$, and $b = (b_1, \ldots, b_n)$.

Let for the sake of an example n be even.

$$\begin{aligned} ab &= a_1b_1 + \ldots a_nb_n \\ &= ((a_1 + b_2)(a_2 + b_1) + (a_3 + b_4)(a_4 + b_3) + \ldots (a_{n-1} + b_{n-1})(a_n + b_n)) \\ &- (a_1a_2 + a_3a_4 + \ldots + a_{n-1}a_n) - (b_1b_2 + b_3b_4 + \ldots + b_{n-1}b_n)\,. \end{aligned}$$

1. We precompute the $a_1a_2 + \ldots + a_{n-1}a_n$ once per row of A and similarly $b_1b_2 + b_3b_4 + \ldots + b_{n-1}b_n$ once per column of B. We have $2n$ such rows/columns of A and B. For each such product-sum we perform $n/2$ M's (short for multiplications) and $n/2 - 1$ A's (short for additions). The total preprocessing count is thus $2nn/2 = n^2$ M's and $2n(n/2 - 1) = n^2 - 2n$ A's.

2. The preprocessing is performed once. In order to find the product $A \times B$ we need to find n^2 vector-products involving rows of A and columns B using the information collected in the preprocessing step. This is equivalent to performing the following computation, where a', b' are pieces of preprocessed information obtained in step 1. For each one of n^2 vector products we perform: $n/2$ M's for the products like $(a_1 + b_2)(a_2 + b_1)$, n A's for the additions (each of $n/2$ products like $(a_1 + b_2)(a_2 + b_1)$ requires two additions), $n/2 - 1$ additions to combine the products, and 2 A's (in fact subtractions) of the a', b' . The total count is thus $n/2$ M's and $n + (n/2 - 1) + 2 = 3n/2 + 1$ A's per single vector product. The total count is thus $n^2n/2 = n^3/2$ M's and $(3n/2 + 1)n^2$.

$$ab = ((a_1 + b_2)(a_2 + b_1) + (a_3 + b_4)(a_4 + b_3) + \ldots (a_{n-1} + b_{n-1})(a_n + b_n)) - a' - b'$$

3. The overall cost is the sum of the two individual ones of steps 1 and 2. This yields $n^3/2+n^2$ M's and $3n^3/2+n^2+n^2-2n = 3n^3/2 + 2n^2 - 2n$ A's.

The standard algorithm for matrix multiplication requires only n^3 M's and $n^3 - n^2$ A's.

(c) Copyright A. Gerbessiotis. All rights reserved.

Strassen's algorithm

Suppose we view matrix multiplication not the traditional way but devise for it a divide and conquer algorithm. Let us assume we are given an $n \times n$ matrix A, and let us assume without loss of generality that n is a power of two. Then we can view A as consisting of four $n/2 \times n/2$ matrices $A_{12}, A_{11}, A_{21}, A_{22}$. Then the product of two matrices A and B can be viewed as a blocked matrix product.

$$C = A \times B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

where, $C_{11} = A_{11}B_{11} + A_{12}B_{21}$, $C_{12} = A_{11}B_{12} + A_{12}B_{22}$, $C_{21} = A_{21}B_{11} + A_{22}B_{21}$, $C_{22} = A_{21}B_{12} + A_{22}B_{22}$. It is clear that for the calculation of C, 8 $n/2 \times n/2$ multiplications are required and 4 additions. Each addition requires $n^2/4$ additions for a total of n^2 operations. Then the total number of operations (additions and mutltiplications)

$$T(n) = 8T(n/2) + n^2$$

By the master method we derive that $T(n) = O(n^3)$. In the late 60s Strassen reformulated the calculations for C so that only 7 matrix multiplications (including its additions), and 7 additions and 8 subtractions are required. For Strassen's method we need

$$T(n) = 7T(n/2) + 15n^2/4$$

The solution to this recurrence is $T(n) = O(n^{\lg 7}) = O(n^{2.80})$. There has been much activity in reducing the exponent of matrix multiplication to something close to 2. The best exponent for twenty or so years has been an algorithm due to Coppersmith and Winograd where the exponent is $\omega = 2.376$ giving a running time of $T(n) = O(n^{\omega})$. This has been improved recently by Strothers (2010) where $\omega < 2.3737$ and Williams (2011) where $\omega < 2.3727$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Original scheduling

Strassen's original method involved 18 additions thus giving a recurrence of

$$T(n) = 7T(n/2) + 18n^2/4$$

for matrix multiplication. For Strassen's method to work as claimed n must be a power of two and the base case is for $n = 2$. The number of operations required to multiply two 2×2 matrices is given either by the traditional method (known also as Cannon's method) of 12 operations (8 multiplications, 4 additions) or Strassen's method itself of 25 operations (7 multiplications, and 18 additions).

$$\begin{aligned}
m_1 &= (A_{12} - A_{22}) * (B_{21} + B_{22}) \\
m_2 &= (A_{11} + A_{22}) * (B_{11} + B_{22}) \\
m_3 &= (A_{11} - A_{21}) * (B_{11} + B_{12}) \\
m_4 &= (A_{11} + A_{12}) * B_{22} \\
m_5 &= A_{11} * (B_{12} - B_{22}) \\
m_6 &= A_{22} * (B_{21} - B_{11}) \\
m_7 &= (A_{21} + A_{22}) * B_{11} \\
C_{11} &= m_1 + m_2 - m_4 + m_6 \\
C_{21} &= m_6 + m_7 \\
C_{12} &= m_4 + m_5 \\
C_{22} &= m_2 - m_3 + m_5 - m_7.
\end{aligned}$$

(c) Copyright A. Gerbessiotis. All rights reserved.

Introduction

A set of linear equations with unknowns $x_1, x_2, \ldots x_n$ is

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2, \\ \ldots &= \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n, \end{aligned} \tag{1}$$

A set of values for $x_1, \ldots$ that satisfy all Eq. 1 is said to be a solution to these equations. We can rewrite this system as follows in matrix-vector product form or equivalently letting $A = (a_{ij})$ and $x = (x_i)$ and $b = (b_i)$, we can write $Ax = b$. Then $x = A^{-1}b$, as long as A has an inverse A^{-1}. In that case vector x is unique.

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{pmatrix}$$

A (column) vector x is an $n \times 1$ matrix. A row vector x^T is an $1 \times n$ matrix.

(c) Copyright A. Gerbessiotis. All rights reserved.

LUP Decomposition

The idea behind the LUP decomposition is to find three matrices L, U, P such that $PA = LU$,where P is a permutation matrix of zeroes and ones, U is an upper-triangular matrix and L is a lower triangular matrix. Then solving A reduces to the problem of solving triangular matrices which is simpler.

Suppose we have determined the LUP decomposition of A. Then $Ax = b$ yields an equivalent system $PAx = Pb$, where PA permutes the rows of A. It is easy to see that an 1 in position (i, j) in P has the effect of moving row j of A to row i in PA. Similar observations can be derived for columns of A and AP respectively.

Since $PA = LU$ from the decomposition, we obtain that $PAx = Pb$ can be rewritten as $LUx = Pb$. We solve such a system in two steps. Let us write $Ux = y$. Then $Ly = Pb$.

```
  Solve(A,x,b)
1. Find LUP decomposition of A (i.e. PA=LU).
2. Solve Ly=Pb by forward substitution.
3. Solve Ux=y by backward substitution.
4. return(x)
```

We first solve the system $Ly = Pb$ by forward substitution to find y.

Then we solve the system $Ux = y$ by backward substitution to find x, the solution of $Ax = b$.

This is so because by $Ax = b$ and $PA = LU$, we get $A = P^{-1}LU$.

$$Ax = P^{-1}L(Ux) = P^{-1}Ly = P^{-1}Pb = b,$$

as required.

(c) Copyright A. Gerbessiotis. All rights reserved.

Forward and Backward Substitution

Forward substitution solves a linear system of n unknowns in $O(n^2)$ time. Note that for the system of equations below, we have $b_{j[i]}$ instead of say b_i because of the product Pb: this permutes the rows (i.e. elements) of the column vector b by moving to row i the element previously in row $j[i]$. We also note that the diagonal elements of L are units (i.e. one) where as the diagonal elements of U are not necessarily one.

$$\begin{array}{rrrcrl}
y_1 + & & & \ldots & & = b_{j[1]} \\
l_{21}y_1 + & y_2 + & & \ldots & & = b_{j[2]} \\
l_{31}y_1 + & l_{32}y_2 + & y_3 + & \ldots & & = b_{j[3]} \\
 & & & \ldots & & \\
l_{n1}y_1 + & l_{n2}y_2 + & l_{n3}y_3 + & \ldots & y_n & = b_{j[n]}
\end{array}$$

We first solve for y_1 from the first equation. We have that $y_1 = b_{j[1]}$.
Substituting y_1 into the second equation and solving for y_2 we find that $y_2 = b_{j[2]} - l_{21}y_1$.
Similarly $y_i = b_{j[i]} - \sum_{k=1}^{i-1} l_{ik}y_k$.

Back substitution works upwards by first solving for x_n in $Ux = y$. Then $x_i = \left(y_i - \sum_{k=i+1}^{n} u_{ik}x_k\right)/u_{ii}$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Finding the LUP decomposition

In the absence of P, the LUP decomposition can be found through the LU decomposition algorithm, which is oftern called Gaussian elimination. In Gaussian elimination, we first subtract multiples of the first equation from the others so that the first variable is eliminated from the second third and so on equations. After we accomplish this we then subtract multiples of the second equation from the others so that the second variable is eliminated from the the third fourth and so on equations, and so on. This way we end up with an upper triangular matrix in place of A. Matrix L is made up of the multipliers we used in variable elimination. An LU decomposition description is a recursive procedure.

If $n = 1$, we are done, since we can choose $L = I_1$ and $U = A$.

For $n > 1$ we break A into four pieces.

$$A = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \dots & a_{1n} \\ -- & -- & -- & -- \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{array} \right) = \left(\begin{array}{cc} a_{11} & w^T \\ v & A' \end{array} \right)$$

where v is $n-1$ element column vector, w^T is an $n-1$ element row vector, and A' is an $(n-1) \times (n-1)$ matrix. Then we can verify that

$$A = \left(\begin{array}{cc} a_{11} & w^T \\ v & A' \end{array} \right) = \left(\begin{array}{cc} 1 & 0 \\ v/a_{11} & I_{n-1} \end{array} \right) \left(\begin{array}{cc} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{array} \right)$$

In the latter expression, the 0's in the first row and first column span $n-1$ columns and rows respectively. The term $A' - vw^T/a_{11}$ is an $(n-1) \times (n-1)$ matrix called the **Schur complement of** A with respect to a_{11}.

We now claim if A is nonsingular so is the Schur complement. We prove it by way of contradiction. We thus assume that the Schur complement is singular. Then its row rank is strictly less than $n-1$. Then because the bottom $n-1$ rows have 0 in the first column, the first row is independent of the $n-1$ other rows, i.e. the row rank of the entire matrix is less than $n-1$, and so is that of the product, i.e. rank of A is strictly less than $n-1$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Finding the LUP decomposition (continued)

Since the Schur complement is nonsingular we can recursively establish its own complement by further decomposing $A' - vw^T/a_{11}$. Let the end result of the recursion that LU decomposes the Schur complement of A with respect to a_{11} be $A' - vw^T/a_{11} = L'U'$.

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & U' \end{pmatrix} = LU. \end{aligned}$$

If a_{11} is zero, the whole scheme breaks down. We call such elements (the diagonal elements) the **pivots**. In order to make sure that a pivot is not zero, we can preprocess A, by permuting its rows so that a_{11} is non-zero. The same applies to the Schur complement of A. The row exchanges in pivoting operations are kept track in the permutation matrix P.

```
  LUDecompose(A)
1. for(k=1;k<=n;k++) {
2.    u_kk = a_kk;
3.    for (i=k+1;i<=n;i++) {
4.     l_ik = a_ik / u_kk; //l_ik holds v_i
                                        T
5.     u_ki = a_ki;        //u_ki holds w _i
6.    }
7.    for(i=k+1;i<=n;i++){             // The Schur complement of A
8.      for(j=k+1;j<=n;j++){           // with respect to a_kk
          a_ij = a_ij - l_ik * u_kj ;  // is computed
9.      }
10.   }
11. }
12. return(L=l[ ][ ],U=u[ ][ ]);
```

Because of the triple loop of lines 1, 7, and 8, running time is $O(n^3)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Finding the LUP decomposition (continued)

The pivoting variant is the same except that before an elimination takes place (i.e. the Schur complement is established), we permute rows so that in the k-th iteration the element with largest absolute value is located in a_{kk}.

For example, before we start dividing by the input matrix's a_{11} we exchange the first row with that row l whose a_{l1} has the largest absolute value, and after exchanging the l-th and 1-st rows, then and only then the Schur complement calculation is performed: the a_{11} used is the original a_{l1} because of the row exchange. This pivoting operation requires for the i column $n - i$ comparisons to establish the element with the largest absolute value and $n - i + 1$ swaps for row exchange. The total cost of pivoting over the n columns is thus $O(n^2) = o(n^3)$.

We keep track of row exchanges not in a permutation matrix which is a sparse matrix (only n of n^2 entries are non-zero) and wastes space but in an one-dimensional matrix insteaad to save space.

Initially the i-th entry of this matrix is set to i. If at some point we exchange row k and row l, we also exchange the k-th and l-th entries of the matrix.

(c) Copyright A. Gerbessiotis. All rights reserved.

LUP-based algorithm

The LUP decomposition of A requires $O(n^3)$ operations. This cost can be considered a preprocessing cost if many equations on the same A need to be solved.

After we have established this decomposition, we can solve the system $Ax = b$ in $O(n^2)$ additional time (the cost of forward and backward substitution). If another system $Ax = c$ is given we can solve this system in additional $O(n^2)$ time as A is already decomposed.

If k systems of the form $Ax = D_i$, $i = 1, \ldots, k$ are given, all these systems can be solved in $O(kn^2)$ time given the availability of LUP of A. If we have n such systems the total time to solve these is $O(n \cdot n^2) = O(n^3)$ plus the time to decompose A which is also $O(n^3)$ for a total of $O(n^3)$.

If the n systems are the n columns of I_n, the x column vectors that we calculate are the column vectors of a matrix X such that $AX = I_n$, i.e. we thus can calculate the inverse of A in total $O(n^3)$ time, provided that A exists.

(c) Copyright A. Gerbessiotis. All rights reserved.

Theorem. Matrix multiplication is no harder than inversion as long as the time $In(n)$ it takes to invert an $n \times n$ matrix is $\Omega(n^2)$, and $In(3n) = O(In(n))$.

Proof. Suppose A, B, C are three matrices for which the product $C = A \times B$ needs to be established. We then create the matrix

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}$$

whose inverse is,

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}$$

Forming D from A and B takes $O(n^2)$ time. Finding the inverse of D which is a $3n \times 3n$ matrix takes time $In(3n)$, where $In(n)$ is the time it takes to invert a matrix of size $n \times n$. Subsequently AB can be read off D^{-1} in $O(n^2)$ time. Therefore $M(n)$ the time it takes to multiply two $n \times n$ matrices is given by the following expression

$$M(n) = O(n^2) + In(3n)$$

Since $In(3n) = O(In(n))$ and $In(n) = \Omega(n^2)$, we have than

$$M(n) = O(In(n)).\square$$

Theorem. Inversion is no harder than matrix multiplication (under certain "innocent conditions").

Proof. Perform a similar trick by forming a matrix and then endup with the following recurrence

$$In(n) = 2In(n/2) + \Theta(M(n))$$

The master method (case 3) provides the solution to this recurrence as long as $2M(n/2) < M(n)$. The solution is $In(n) = O(M(n))$.

Relationships (continued)

An easier bound to derive is by noting that an $n \times n$ matrix A can de written in terms of its $n/2 \times n/2$ submatrices $A_{11}, A_{12}, A_{21}, A_{22}$ as follows.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & 0 \\ 0 & D \end{pmatrix} \begin{pmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{pmatrix}$$

where $D = A_{22} - A_{21}A_{11}^{-1}A_{12}$. We denote by M matrix multiplications of $n/2 \times n/2$ matrices and with I inversions of $n/2 \times n/2$ matrices.

In order to compute D we need 2 M and 1 I (of A_{11}^{-1}) plus $O(n^2)$ time for the subtraction.

The triple matrix form of A requires 1 additional M to compute $A_{11}^{-1}A_{12}$ or $A_{21}A_{11}^{-1}$ as the other one is a partial product in D's computation.

Therefore the total count is 3 M's, 1 I, plus $O(n^2)$ work in order to establish the triple matrix representation of A.

After this representation of A has been established, the inverse of A is then given by

$$A^{-1} = \begin{pmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{pmatrix} \begin{pmatrix} A_{11}^{-1} & 0 \\ 0 & D^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{pmatrix}$$

In order to multiply the second and third matrices we need to perform 1 M. In order to multiply the first with the previous result we need 2 M's. The inverse of A_{11} needs also to be established.

The total count therefore is

$$In(n) = 2In(n/2) + 6M(n/2) + \Theta(n^2).$$

This method is slightly better than the one in the textbook where the $6M(n/2)$ term becomes $4M(n)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Appendix
Matrix Inversion through Matrix Multiplication

The transformation that proves that matrix inversion is no harder than matrix multiplication can be made to work without a need of checking whether A_{11} and D are singular or not (as long as we know that A is nonsingular), if instead of finding the inverse of A we find the inverse of A^tA, which is $(A^tA)^{-1}$. It is easy to show that

$$A^{-1} = (A^tA)^{-1}A^t$$

and therefore given the inverse of A^tA we can recover A^{-1} by one extra matrix multiplication operation for a total additional cost of $M(n)$. A^tA is a positive definite matrix and as such is guaranteed that A_{11} and D are nonsingular and positive definite.