Transposes

September 7, 2017

1 Transpose, Permutations, and Orthogonality

One special type of matrix for which we can solve problems much more quickly is a permutation matrix, introduced in the previous lecture on $PA = LU$ factorization.

In [1]:
```
# construct a permutation matrix P from the permutation vector p
function permutation_matrix(p)
    P = zeros(Int, length(p),length(p))
    for i = 1:length(p)
        P[i,p[i]] = 1
    end
    return P
end
```

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

In [2]:
```
P = permutation_matrix([2,4,1,5,3])
```

Out[2]:
```
5×5 Array{Int64,2}:
 0  1  0  0  0
 0  0  0  1  0
 1  0  0  0  0
 0  0  0  0  1
 0  0  1  0  0
```

In [3]:
```
I₅ = eye(5)
```

Out[3]:
```
5×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  0.0  1.0
```

In [4]:
```
P * I₅
```

Out[4]:
```
5×5 Array{Float64,2}:
 0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0
 1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0
 0.0  0.0  1.0  0.0  0.0
```

The inverse of any permutation matrix P turns out to be its transpose P^T: we just swap rows and columns. In Julia, this is denoted `P` (technically, this is the conjugate transpose, and `P.` is the transpose, but the two are the same for real-number matrices where complex conjugation does nothing).

In [5]:
```
P'
```

Out[5]:
```
5×5 Array{Int64,2}:
 0  0  1  0  0
 1  0  0  0  0
 0  0  0  0  1
 0  1  0  0  0
 0  0  0  1  0
```

In [6]:
```
P'*P
```

Out[6]:
```
5×5 Array{Int64,2}:
 1  0  0  0  0
 0  1  0  0  0
 0  0  1  0  0
 0  0  0  1  0
 0  0  0  0  1
```

In [7]:
```
P*P'
```

Out[7]:
```
5×5 Array{Int64,2}:
 1  0  0  0  0
 0  1  0  0  0
 0  0  1  0  0
 0  0  0  1  0
 0  0  0  0  1
```

The reason this works is that $P^T P$ computes the dot products of *all the columns* of P with *all of the columns*, and the columns of P are orthonormal (orthogonal with length 1). We say that P is an example of an "orthogonal" matrix or a "unitary" matrix. We will have much to say about such matrices later in 18.06.

2 Transposes and products

Transposes are important in linear algebra because they have a special relationship to matrix and vector products:

$$(AB)^T = B^T A^T$$

and hence for a dot product (inner product) $x^T y$

$$x \text{ dot } (Ay) = x^T (Ay) = (A^T x)^T y = (A^T x) \text{ dot } y$$

We can even turn the second step around and use this as the *definition* of a transpose: a transpose is *what "moves" a matrix from one side to the other of a dot product.*

In [8]:
```
C = rand(-9:9, 4,4)
D = rand(-9:9, 4,4)
(C*D)' == D'*C'
```

Out[8]:
```
true
```

3 Transposes and inverses

From the above property, we have:

$$(AA^{-1})^T = (A^{-1})^T A^T = I^T = I$$

and it follows that:

$$(A^{-1})^T = (A^T)^{-1}$$

The *transpose of the inverse* is the *inverse of the transpose*.

```
In [9]: A = [4  -2  -7  -4  -8
         9  -6  -6  -1  -5
        -2  -9   3  -5   2
         9   7  -9   5  -8
        -1   6  -3   9   6]
```

```
Out[9]: 5×5 Array{Int64,2}:
  4  -2  -7  -4  -8
  9  -6  -6  -1  -5
 -2  -9   3  -5   2
  9   7  -9   5  -8
 -1   6  -3   9   6
```

```
In [10]: inv(A')
```

```
Out[10]: 5×5 Array{Float64,2}:
  0.0109991   0.131989  -0.235564  -0.301558    0.2044
  0.529789    0.35747   -0.179652  -0.69172     0.678582
 -0.908341   -0.900092   0.370302   1.48701    -1.29667
 -0.635197   -0.622365   0.353804   1.16499    -1.05408
 -0.0879927  -0.055912  -0.11549    0.0791323   0.0314696
```

```
In [11]: inv(A)'
```

```
Out[11]: 5×5 Array{Float64,2}:
  0.0109991   0.131989  -0.235564  -0.301558    0.2044
  0.529789    0.35747   -0.179652  -0.69172     0.678582
 -0.908341   -0.900092   0.370302   1.48701    -1.29667
 -0.635197   -0.622365   0.353804   1.16499    -1.05408
 -0.0879927  -0.055912  -0.11549    0.0791323   0.0314696
```

As expected, they match!

4 Transposes and LU factors

If $A = LU$, then $A^T = U^T L^T$. Note that U^T is *lower* triangular, and L^T is *upper* trangular. That means, that once we have the LU factorization of A, we immediately have a similar factorization of A^T.

```
In [12]: L,U,p = lu(A)
```

```
Out[12]: (
[1.0 0.0 ... 0.0 0.0; 1.0 1.0 ... 0.0 0.0; ... ; -0.111111 0.410256 ... 1.0 0.0; -0.222222 -0.7

[9.0 -6.0 ... -1.0 -5.0; 0.0 13.0 ... 6.0 -3.0; ... ; 0.0 0.0 ... 8.67894 9.95297; 0.0 0.0 ...

[2,4,1,5,3])
```

```
In [13]: L'
```

```
Out[13]: 5×5 Array{Float64,2}:
 1.0  1.0  0.444444   -0.111111  -0.222222
 0.0  1.0  0.0512821   0.410256  -0.794872
 0.0  0.0  1.0         0.582822   0.171779
 0.0  0.0  0.0         1.0        0.0242696
 0.0  0.0  0.0         0.0        1.0
```

In [14]: U'

Out[14]: 5×5 Array{Float64,2}:
```
  9.0   0.0   0.0       0.0       0.0
 -6.0  13.0   0.0       0.0       0.0
 -6.0  -3.0  -4.17949   0.0       0.0
 -1.0   6.0  -3.86325   8.67894   0.0
 -5.0  -3.0  -5.62393   9.95297  -0.771206
```

In particular, suppose we know the $PA = LU$ factorization for A, but we want to solve $A^T x = b$. We can:

- Write $A = P^T LU \implies A^T = U^T L^T P$
- Substitute this in to $A^T x = b$ to obtain $U^T L^T P x = b$
- Parenthesize and solve from the "outside in": $U^T(L^T(Px)) = b$:
  - First solve $U^T c = b$ for c by forward-substitution
  - Then solve $L^T d = c$ by backsubstitution
  - Then solve $Px = d$ for $x = P^T d$ (i.e. just reversing the permutation)

Let's try it:

In [15]:
```
b = [4,2,1,-2,3] # "randomly" chosen right-hand side
A' \ b # correct solution to A[U+1D40]x = b
```

Out[15]: 5-element Array{Float64,1}:
```
  1.28873
  6.07363
-11.9273
 -8.92392
 -0.643141
```

In [16]:
```
c = U' \ b # forward-substitution
d = L' \ c # backsubstitution
permutation_matrix(p)' * d
```

Out[16]: 5-element Array{Float64,1}:
```
  1.28873
  6.07363
-11.9273
 -8.92392
 -0.643141
```

As usual, the `lufact(A)` object (which encapsulates L, U, and P) does all this for you (in a more efficient way because it makes sure to take advantage of the special structure of these matrices, which we didn't above):

In [17]:
```
LU = lufact(A)
LU' \ b
```

Out[17]: 5-element Array{Float64,1}:
```
  1.28873
  6.07363
-11.9273
 -8.92392
 -0.643141
```

5 Symmetric matrices

A *very* important type of matrix that arises frequently in real problems (we will have *much more* to say about this later in the course, after exam 2) is a symmetric matrix: a matrix S that is equal to its transpose $S = S^T$.

Given any matrix A, we can make a symmetric matrix out of it very easily in two ways: * $A + A^T$ (or often we write the "symmetric part" of A as $\frac{A+A^T}{2}$). (For square matrices only.) * $A^T A$ or AA^T. (This even works for *non-square* matrix.)

```
In [18]: S = A' * A
```

```
Out[18]: 5×5 Array{Int64,2}:
          183    13  -166    21  -159
           13   206   -58   148     8
         -166   -58   184   -53   146
           21   148   -53   148    41
         -159     8   146    41   193
```

The ordinary LU factorization of a symmetric S, however, seems to have nothing to do with the symmetry of S. Is there any special relationship between L and U in this case?

```
In [19]: L, U = lu(S, Val{false}) # LU without pivoting
```

```
Out[19]: (
         [1.0 0.0 ... 0.0 0.0; 0.0710383 1.0 ... 0.0 0.0; ... ; 0.114754 0.714408 ... 1.0 0.0; -0.86885

         [183.0 13.0 ... 21.0 -159.0; 0.0 205.077 ... 146.508 19.2951; ... ; 0.0 0.0 ... 40.8852 45.711

         [1,2,3,4,5])
```

```
In [20]: L
```

```
Out[20]: 5×5 Array{Float64,2}:
           1.0         0.0         0.0         0.0      0.0
           0.0710383   1.0         0.0         0.0      0.0
          -0.907104   -0.225319    1.0         0.0      0.0
           0.114754    0.714408   -0.0408412   1.0      0.0
          -0.868852    0.0940872   0.265894    1.11804  1.0
```

```
In [21]: U
```

```
Out[21]: 5×5 Array{Float64,2}:
          183.0   13.0    -166.0      21.0       -159.0
            0.0  205.077   -46.2077  146.508       19.2951
            0.0    0.0      23.0093   -0.939727     6.11804
            0.0    0.0       0.0      40.8852      45.7112
            0.0    0.0       0.0       0.0          0.303428
```

U and L *seem* quite different because L has 1's along the diagonal, but U has some other numbers (the pivots). We can extract these with `diag(U)` in Julia:

```
In [22]: diag(U)
```

```
Out[22]: 5-element Array{Float64,1}:
          183.0
          205.077
           23.0093
           40.8852
            0.303428
```

We could make U look more like L if we *divided each row* of U by these pivots. That corresponds to multiplying $D^{-1}U$, where D is the *diagonal matrix* of the pivots:

```
In [23]: D = diagm(diag(U)) # diagm makes a diagonal matrix from a 1d array
```

```
Out[23]: 5×5 Array{Float64,2}:
          183.0    0.0      0.0       0.0      0.0
            0.0  205.077    0.0       0.0      0.0
            0.0    0.0     23.0093    0.0      0.0
            0.0    0.0      0.0      40.8852   0.0
            0.0    0.0      0.0       0.0      0.303428
```

Since a diagonal matrix just multiplies each row by a single number, the inverse of a diagonal matrix simply *divides* each row by the *reciprocal* of that number:

```
In [24]: inv(D)
```

```
Out[24]: 5×5 Array{Float64,2}:
          0.00546448  0.0         0.0        0.0        0.0
          0.0         0.00487623  0.0        0.0        0.0
          0.0         0.0         0.0434607  0.0        0.0
          0.0         0.0         0.0        0.0244587  0.0
          0.0         0.0         0.0        0.0        3.29568
```

```
In [25]: inv(D) * U
```

```
Out[25]: 5×5 Array{Float64,2}:
          1.0  0.0710383  -0.907104   0.114754   -0.868852
          0.0  1.0        -0.225319   0.714408    0.0940872
          0.0  0.0         1.0       -0.0408412   0.265894
          0.0  0.0         0.0        1.0         1.11804
          0.0  0.0         0.0        0.0         1.0
```

Wait a minute, now the entries look *exactly* like those of L, except above the diagonal rather than below. In fact, this is precisely the *transpose* of L:

```
In [26]: L'
```

```
Out[26]: 5×5 Array{Float64,2}:
          1.0  0.0710383  -0.907104   0.114754   -0.868852
          0.0  1.0        -0.225319   0.714408    0.0940872
          0.0  0.0         1.0       -0.0408412   0.265894
          0.0  0.0         0.0        1.0         1.11804
          0.0  0.0         0.0        0.0         1.0
```

Since $D^{-1}U = L^T$, we have $U = DL^T$, and hence $S = LU = LDL^T$.

This fact is so important that it has its own name: we have constructed the **LDL[U+1D40] factorization** of our symmetric matrix S. This factorization is useful for two reasons:

- It preserves the special structure of a symmetric matrix, which is important if we are to do subsequent algebraic manipulations: $(LDL^T)^T = LDL^T$.
- Clever implementations can save roughly a factor of two in the number of operations by exploiting the symmetry.

6 Cholesky factorization

Finally, we should mention another very important variation on this theme.

Suppose that we have a symmetric matrix S in which **all the pivots are positive**. This is called a positive-definite matrix, and turns out to be the case *whenever* you construct S from $A^T A$ or AA^T (for real A), as above. We will have much more to say about such matrices later in the course.

In that case, we can take the *square roots* of the pivots to write $D = KK$ where K is a diagonal matrix of the square roots of the pivots:

```
In [27]: K = diagm(sqrt.(diag(U)))

Out[27]: 5×5 Array{Float64,2}:
          13.5277   0.0      0.0      0.0      0.0
           0.0     14.3205   0.0      0.0      0.0
           0.0      0.0      4.7968   0.0      0.0
           0.0      0.0      0.0      6.39416  0.0
           0.0      0.0      0.0      0.0      0.550843
```

```
In [28]: K*K - D

Out[28]: 5×5 Array{Float64,2}:
          0.0   0.0           0.0          0.0  0.0
          0.0  -2.84217e-14   0.0          0.0  0.0
          0.0   0.0           3.55271e-15  0.0  0.0
          0.0   0.0           0.0          0.0  0.0
          0.0   0.0           0.0          0.0  5.55112e-17
```

Then we can write $S = LDL^T = LKKL^T = (LK)(LK)^T$. The matrix $\hat{L} = LK$ is also a lower-triangular matrix, it is L with the *columns* scaled by K. So, we can write *any* symmetric positive-definite (SPD) matrix as:

$$S = \hat{L}\hat{L}^T$$

This is called the Cholesky factorization of S, and it usually the most efficient way to solve SPD systems (half the operations, and often half the storage, compared to LU). In Julia, it is computed by `chol` (which returns $\hat{L}^T$) or `cholfact`:

```
In [29]: chol(S)

Out[29]: 5×5 UpperTriangular{Float64,Array{Float64,2}}:
          13.5277   0.960988  -12.2711     1.55236   -11.7536
            ⋅      14.3205     -3.22668   10.2307      1.34738
            ⋅        ⋅          4.7968    -0.195907    1.27544
            ⋅        ⋅           ⋅         6.39416     7.14891
            ⋅        ⋅           ⋅          ⋅          0.550843
```

```
In [30]: (L*K)'

Out[30]: 5×5 Array{Float64,2}:
          13.5277   0.960988  -12.2711     1.55236   -11.7536
           0.0     14.3205     -3.22668   10.2307      1.34738
           0.0      0.0         4.7968    -0.195907    1.27544
           0.0      0.0         0.0        6.39416     7.14891
           0.0      0.0         0.0        0.0         0.550843
```

One interesting fact about Cholesky factorization of SPD matrices is that **row swaps are never required**, even when concerns about roundoff errors are included, so there is no P matrix.