1 Matrix inverses

It is often conceptually convenient to talk about the inverse $A^{-1}$ of a matrix $A$, which exists for any non-singular square matrix. This is the matrix such that $x = A^{-1}b$ solves $Ax = b$ for any $b$. The inverse is conceptually convenient because it allows us to move matrices around in equations almost like numbers (except that matrices don’t commute!).

Another way of defining the inverse of a matrix involves the identity matrix $I$. Here is a $5 \times 5$ identity matrix:

$$I = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} = (e_1 \ e_2 \ e_3 \ e_4 \ e_5)$$

where the columns $e_1 \cdots e_5$ of $I$ are the unit vectors in each component.

The identity matrix, which can be constructed by `eye(5)` in Julia, has the property that $Ix = x$ for any $x$, and hence $IA = A$ for any (here $5 \times 5$) matrix $A$:

```
In [1]: A = [4 -2 -7 -4 -8
6 -6 -1 -5
-2 -9 3 -5 2
9 7 -9 5 -8
-1 6 -3 9 6]
# a randomly chosen 5x5 matrix
```

```
Out[1]: 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 [2]: b = [-7,2,4,-4,-7]
# a randomly chosen right-hand side
```

```
Out[2]: 5-element Array{Int64,1}:
 -7
  2
  4
 -4
 -7
```

```
In [3]: I_5 = eye(Int, 5)
```

```
Out[3]: 5×5 Array{Int64,2}:
 1  0  0  0  0
```
In [4]: I\[3\] * b == b

Out[4]: true

In [5]: I\[3\] * A == A

Out[5]: true

The inverse matrix $A^{-1}$ is the matrix such that $A^{-1}A = AA^{-1} = I$.

Why does this correspond to solving $Ax = b$? Multiplying both sides on the left by $A^{-1}$ (multiplying on the right would make no sense: we can’t multiply vector $\times$ matrix!), we get

$$A^{-1}Ax = Ix = x = A^{-1}b$$

How do we find $A^{-1}$? The key is the equation $AA^{-1} = I$, which looks just like $AX = B$ for the right-hand sides consisting of the columns of the identity matrix, i.e. the unit vectors. So, we just solve $Ax = e_i$ for $i = 1,\ldots,5$, or equivalently do $A \backslash I$ in Julia. Of course, Julia comes with a built-in function `inv(A)` for computing $A^{-1}$ as well:

In [6]: Ainv = A \ I

Out[6]: 5×5 Array{Float64,2}:
0.0109991 0.529789 -0.908341 -0.635197 -0.0879927
0.131989 0.35747 -0.900092 -0.622365 -0.055912
-0.235564 -0.179652 0.370302 0.353804 -0.11549
-0.301558 -0.69172 1.48701 1.16499 0.0791323
0.2044 0.678582 -1.29667 -1.05408 0.0314696

In [9]: Ainv - inv(A)

Out[9]: 5×5 Array{Float64,2}:
1.88738e-15 1.46549e-14 -1.82077e-14 -2.22045e-15 -1.15463e-14
-1.58068e-14 -7.68829e-15 -2.05808e-14 -5.77316e-15 1.0

(The difference is just roundoff errors.)

In [10]: Ainv * A

Out[10]: 5×5 Array{Float64,2}:
1.0 -8.32667e-15 -1.49325e-14 -4.88498e-15 -1.0103e-14
1.46549e-14 1.0 -1.82077e-14 -2.22045e-15 -1.15463e-14
-1.58068e-14 1.4877e-14 2.36478e-14 1.0 1.4877e-14
1.38639e-14 -7.68829e-15 -2.05808e-14 -5.77316e-15 1.0

(Again, we get $I$ up to roundoff errors because the computer does arithmetic only to 15–16 significant digits.)

In [11]: A * Ainv
Normally, $AB \neq BA$ for two matrices $A$ and $B$. Why can we multiply $A$ by $A^{-1}$ on either the left or right and get the same answer $I$? It is fairly easy to see why:

$$AA^{-1} = I \implies AA^{-1}A = IA = A = A(A^{-1})$$

Since $A(A^{-1}) = A$, and $A$ is non-singular (so there is a unique solution to this system of equations), we must have $A^{-1}A = I$.

Matrix inverses are funny, however:

- Inverse matrices are very convenient in *analytical* manipulations, because they allow you to move matrices from one side to the other of equations easily.

- Inverse matrices are almost never computed in “serious” numerical calculations. Whenever you see $A^{-1}B$ (or $A^{-1}b$), when you go to implement it on a computer you should read $A^{-1}B$ as “solve $AX = B$ by some method.” e.g. solve it by $A \backslash B$ or by first computing the LU factorization of $A$ and then using it to solve $AX = B$.

One reason that you don’t usually compute inverse matrices is that it is wasteful: once you have $PA = LU$, you can solve $AX = B$ directly without bothering to find $A^{-1}$, and computing $A^{-1}$ requires much more work if you only have to solve a few right-hand sides.

Another reason is that for many special matrices, there are ways to solve $AX = B$ much more quickly than you can find $A^{-1}$. For example, many large matrices in practice are sparse (mostly zero), and often for sparse matrices you can arrange for $L$ and $U$ to be sparse too. Sparse matrices are much more efficient to work with than general “dense” matrices because you don’t have to multiply (or even store) the zeros. Even if $A$ is sparse, however, $A^{-1}$ is usually non-sparse, so you lose the special efficiency of sparsity if you compute the inverse matrix.

### 1.1 Inverses and products

Inverses have a special relationship to matrix products:

$$(AB)^{-1} = B^{-1}A^{-1}$$

The reason for this is that we must have $(AB)^{-1}AB = I$, and it is easy to see that $B^{-1}A^{-1}$ does the trick. Equivalently, $AB$ is the matrix that first multiplies by $B$ then by $A$; to invert this, we must *reverse the steps*: first multiply by the inverse of $A$ and then by the inverse of $B$.
2 Complexity of Matrix Operations

With a little effort, we can figure out that the number of arithmetic operations for an $n \times n$ matrix scales proportional to (for large $n$):

- $n^2$ for: matrix $\times$ vector $Ax$, or solving a triangular system like $Ux = c$ or $Lc = b$ (back/forward substitution)
- $n^3$ for: matrix $\times$ matrix $AB$, LU factorization $PA = LU$, or solving a triangular system with $n$ right-hand sides like computing $A^{-1}$ from the LU factorization.

(In computer science, we would say that these have “complexity” $\Theta(n^2)$ and $\Theta(n^3)$, respectively.

Let’s see how these predictions match up to reality:

```julia
In [15]: Pkg.add("BenchmarkTools") # a useful package for benchmarking
    using BenchmarkTools

INFO: Nothing to be done
INFO: METADATA is out-of-date | you may not have the latest version of BenchmarkTools

Measure the time for LU factorization of $10 \times 10$, $100 \times 100$, $500 \times 500$, $1000 \times 1000$, and $2000 \times 2000$ random real (double precision) matrices:

```julia
In [16]: n = [10,100,500,1000,2000]
    t = [@belapsed(lufact($(rand(n,n))), evals=1) for n in n]

Out[16]: 5-element Array{Float64,1}:
    1.048e-6
    0.000157875
    0.00277716
    0.0110866
    0.0571334
```

Now let’s plot it on a log-log scale to see if it is the expected $n^3$ power law:

```julia
In [17]: using PyPlot
    loglog(n, t*1e9, "bo-")
    loglog(n, n.^3, "k--")
    xlabel("matrix size n")
    ylabel("time (ns)")
    legend(["time", L"n^3"])
    title("time for LU factorization")
```
It's pretty close! For large $n$, you can see it starting to go parallel to the $n^3$ line.

The reason it is initially better than $n^3$ (i.e. it is faster than the $n^3$ dependence would suggest) is probably that there are computational tricks that one can do for large matrices that don’t work for small matrices. e.g. for large matrices the code is probably using multiple cores (multiple processors) on my laptop, but for small matrices the problem is too small to exploit parallelism.

Let’s also look at the time to solve $LUX = b$ when we are given the LU factors, which we predict should grow $\sim n^2$:

In [18]: ts = [@belapsed($(lufact(rand(n,n))) \ $(rand(n))) for n in n]

Out[18]: 5-element Array{Float64,1}:
3.25424e-7
0.000174126
0.000993936
0.00215818
0.00535709

In [19]: loglog(n, t*1e9, "bo-")
loglog(n, n.^2, "k--")
xlabel("matrix size n")
ylabel("time (ns)")
legend(["time", L"n^2"])
title("time for LU solve")
Yup, it’s pretty close to the $n^2$ growth! The key point is that, unless you have many ($\gtrsim n$) right-hand sides, most of the effort is spent in Gaussian elimination (finding L and U), not in the back/forward-substitution to solve $LUx = b$.

If we believe this scaling, how long would it take for my laptop to solve a $10^6 \times 10^6$ system of equations?


Out[20]: 11 weeks, 5 days, 15 hours, 47 minutes, 50 seconds

In fact, we usually run out of memory before we run out of time:

In [21]: println((1e6)^2 * sizeof(Float64) / 2^30, " GiB for a 10^6 \times 10^6 matrix")

7450.580596923828 GiB for a $10^6 \times 10^6$ matrix

In practice, people do regularly solve problems this large, and even larger, but they can do so because real matrices that big almost always have some special structure that allows you to solve them more quickly and store them more compactly. For example, a common special structure is sparsity: matrices whose entries are mostly zero. We will learn some basic ways to take advantage of this later in 18.06, and sparse-matrix methods are covered more extensively in 18.335.
3 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 [22]: # 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[22]: permutation_matrix (generic function with 1 method)

In [23]: P = permutation_matrix([2,4,1,5,3])

Out[23]: 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 [24]: P * I

Out[24]: 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 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 [25]: P'

Out[25]: 5×5 Array{Int64,2}:
        0 0 1 0 0
        1 0 0 0 0
        0 0 0 1 0
        0 1 0 0 0
        0 0 0 1 0

In [26]: P'*P

Out[26]: 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 [27]: P*P'
The reason this works is that $P^TP$ 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.

3.1 Transposes and products

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

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

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

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

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 [29]: C = rand(-9:9, 4,4)
   D = rand(-9:9, 4,4)
   (C*D)' == D'*C'

Out[29]: true

3.2 Transposes and inverses

From the above property, we have:

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

and it follows that:

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

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

In [30]: inv(A')

Out[30]: 5×5 Array{Float64,2}

In [31]: inv(A')

Out[31]: 5×5 Array{Float64,2}

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 triangular. That means, that once we have the LU factorization of $A$, we immediately have a similar factorization of $A^T$. 